Distributed Visualization with OpenGL Vizserver

Hans-Christian Hege, André Merzky, Stefan Zachow

August 14, 2001

Abstract

The increasing demand for distributed solutions in computing technology does not stop when it comes to visualization techniques. However, the capabilities of todays applications to perform remote rendering are limited by historical designs legacys. Especially the popular X11 protocol, that has been proven to be extremely flexible and useful for remote 2D graphics applications, collapses in case of remotely rendering complex 3D scenes. In this paper, we give a short overview of *generic* remote rendering technologies available today, and we compare their performances and usabilities to the recently released OpenGL Vizserver by Silicon Graphics, Inc. (SGI): a network extension to the SGI OpenGL rendering engines. In this report we limit ourselves to remote techniques compatible to the OpenGL standard.

1 Introduction

From the early years of computing, server oriented infrastructures are widely used in many application domains. Ever since we know about remote computing, remote resources, and, in fact, the 'net'. In almost the same manner a demand for remote visualization exists. With nowadays increasing immersive and interactive character of even large scale scientific application [1, 2, 3], this demand is still not satisfied - High end 3D visualization almost always depends on *local* graphics equipment.

This paper reviews various existing approaches to close this gap. The next section gives a survey of possible distribution scenarios for visualization systems. Afterwards a number of technologies supporting these scenarios is listed, and their underlying concepts as well as their performance is compared. Finally our investigations concerning the OpenGL Vizserver will be presented in detail. Several performance measurements have been done and the results will be given at the end of this paper.

2 Distributed Rendering and Visualization

All visualization systems known to us are based on a similar design principle. Simple speaking they form a pipeline transforming data (in a very broad sense) into images. Such a visualization pipeline usually consists of several distinct stages. This section will describe the common architecture in some detail, and thereof we will derive possible scenarios for the distribution of a visualization pipeline.

2.1 The Visualization Pipeline

Data sources for visualization tasks are manifold. Huge amounts of data are produced in scientific experiments, or scientific simulations. Business processes like trading, brokering and banking produce much different kinds of data, on various scales of size. Environmental studies as well as earth and astronomical observations produce huge amounts of image data in various formats and frequency ranges as well [4]. Governmental processes as censuses or public transport plannings result in a multitude of statistical information.

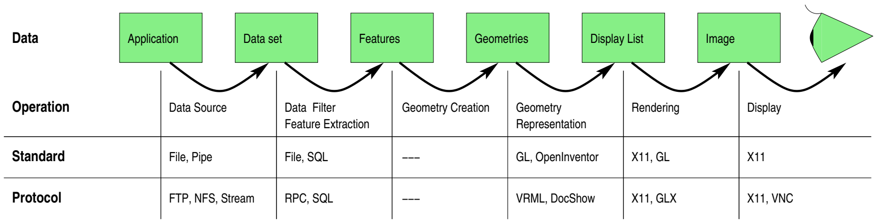

Figure 1: *Scheme of visualization pipeline.*

For all these different data sources, data formats vary greatly. Some data (eg. astronomical pictures) may be displayable directly, but in many cases this won't be possible: How would you *display* a dataset containing the number of inhabitants of all European capitals for example? Although various sensible possibilities will enter your mind immediately, they all do probably need to alter the data set in some way.

For many data sets and visualization tasks, the first processing step is to **extract interesting features** out of the data [6] Interesting are simply those features to be visualized. This step often just performs some filtering of data (eg. for images), but can also include more or less complicated preprocessing steps (eg. computing an isosurface out of a 3D scalar field). Various data sets may be combined at this step (eg. population information with a geographical map). However, user interaction has a strong influence on this process.

Once these features are available, they are **transformed into graphical representatives** (primitives). Usually these primitives are geometries (point sets, lines, surfaces) and images (textures). This step interfaces between the interpretation of the original data and their graphical presentation. Again, this process can be very straightforward (eg. translate a set of geographical coordinates into points), but can also base on complex computational algorithms (eg. create a bundle of slices and images with transparency information from 3D data sets for texture based volume rendering).

The primitives obtained in this step are often very close to the representation used in low level graphic libraries. In fact, the data and geometry types, used there, are often inspired by the data types resulting in the geometry/image creation phase. Visualization systems now **translate geometry and image primitives** into internally used primitives. These are typically point sets, polygons, images etc. This process is usually independent of any user interaction, but it affects to the programmer of a visualization system.

Examples for common libraries and toolkits are X11 [8, 7], vtk [9], Open Inventor [10] and Open GL [11]. These visualization toolkits and libraries do closely interact with the visualization hardware. They perform data abstraction to match the hardware representation of visualization primitives, and initiate the **rendering** process on hardware level[1].

The result of the rendering process is a partial or complete image, which is to be **displayed** on the user's display hardware (monitor, printer...). Most common computer displays are monitors. For a smooth and steady presentation on CRT's or LCD displays, a frame rate of about 70 images per second is required. This is easily achieved by presenting the same image over and over again. For animated visualization though, a minimal frame rate of about 25 frames per second must be achieved in order to produce smooth animations[2].

2.2 Distributing the Visualization Pipeline

A distribution of the rendering pipeline can be realized in *any* of its layers, or even at several stages at the same time. This is done by either partitioning the pipeline and distribution of the resulting stages over a network, or by splitting a single stage for the sake of parallel execution of specialized tasks. An example for the first category may be the network filesystem (NFS), that allows to access a data source as input to a visualization pipeline, located anywhere in the net. An example for parallel execution is *wireGL*, that allows to utilize multiple hardware renderer for parallel execution of the image rendering phase.

[1]Sometimes, if hardware is not available, the rendering is emulated via software. This has certain performance penalties, but frees the renderer from any hardware dependencies.

[2]We want to emphasise the fact that the term 'rate' has a different meaning at every processing step: data could arive at any rate, (from n events per milisecond to one events per day), but it is still possible to achieve smooth animations on the display, depending on what data the data set contains, what vizualization techniques are used to extract the visualization primitives and to render/display them. We need to distinguish between these rates in our later discussion about performance of visualization techniques. Their distribution can be performed at any of its layers, and hence different types of frame rates need to be considered.

The type and amount of data to be transported over the network does heavily depend on the pipeline stage the distribution is applied to. It should be noted that in general the amount of data handled by the visualization pipeline is decreasing from data source to the geometry representation, but is increasing again at the last step, due to the size and number of images/frames produced at rendering level. This is to be taken into consideration by designing a distributed visualization pipeline.

Further it should be noted, that an early[3] distribution of the visualization pipeline often reduces to remote data handling techniques. Various research areas do investigate on this field, and often general approaches are easy to adopt for the visualization case. That is why this paper barely pays attention to distribution techniques applied before the stage of geometry creation or even before feature extraction – they are too numerous to be covered completely and in detail. Thus the following section focuses on techniques providing remote geometry operations, remote rendering and/or display techniques. We evaluate these approaches in terms of **generality**, **usability** and **performance**.

Generality: covers the scope of supported graphics standards (X11, PeX, OpenGL, Inventor...), *and* the generality of the software architecture (library replacement, library wrapper, external tool, proprietary program extension...).

Usability: covers the ease of use, number and complexity of prerequisites, supported platforms and software, as well as general requirements.

Performance: gives estimates for 2D and 3D graphics performance, and for event transport. Graphics performance in the distributed case is often limited by network bandwidth; event performance is important for interactive use, and usually affected by latency. Performance is estimated in a rather inaccurate fashion, and usually varies greatly for different situations. Our performance evaluation results from experienced case studies, and reflects a rather personal estimation. Please keep that in mind! The performance measurements for the OpenGL Vizserver as described in section 5 follows a rather accurate procedure.

3 Remote Visualization Techniques

As described before, the distribution of the visualization pipeline can be realized in *any* of its layers, or even at several stages at the same time. The remote and distributed visualization techniques described in this section are classified in order to reflect this situation. We start with the lowest layer techniques (remote data sources) and end with the highest layers (distributed display technologies). For the sake of completeness a short introduction of local visualization will be given first. A comparison of different visualization systems, for instance can be found in [5].

[3] Early: close to to the data source.

3.1 Local Visualization

Local visualization is the most common method, where visualization software, rendering hardware and output devices are located at the same place. No network load is produced, the limiting factors with respect to visualization performance and quality are:

- performance of data storage/retrieval, memory bandwidth
- software performance (preparation of geometry primitives)
- hardware performance (primitives/second, pixel fillrate, frame rate)
- display capabilities (resolution, frequency, stereo)

Most of these constraints do also apply to all remote techniques, since they utilize the same components as local visualization does – but with additional network interconnects in between. Only approaches which allow to perform certain pipeline operations distributed *in parallel* may overcome these limitations (eg. WireGL), and hence may perform better than equally equipped local visualization environments.

In terms of our evaluation defined in the last section, we can therefore assign highest values for local visualization:

Generality	very general	10
Usability	widely used, flexible, compatible	10
Performance	very fast 2D/3D, immediate events	10

All following distributed/parallel visualization techniques and their properties are compared to local visualization. We start with techniques utilizing remote and distributed data sources.

3.2 Remote and Distributed Data Sources

As mentioned before, visualization pipelines including remote and distributed data sources very often benefit from more general techniques for remote data access. These may range from remote data bases and networking file systems, to the integration of well known and standardized network data transport protocols like FTP and HTTP. We would like to mention NFS as a typical and widely used network file system, and ftp as a well known network data transport protocol.

3.2.1 Network File System – NFS

The NFS [12] makes remote disk storage systems look like local disk space. This allows applications to access remote data via the same API as normal local files. Hence, an application working on locally stored data will see no difference, and can work unchanged. That explains the high marks for generality.

NFS is quite simple to set up, but needs administrative permissions on both data source and visualization side, and involves a certain amount of agreement on organizational level. This decreases the flexibility of the NFS setup significantly. NFS also involves some security risks that lower the usability. In LAN environments, NFS is surely the most commonly used way of distributing a rendering pipeline, and very often it is not even noticed by the users.

The performance of NFS environments depends on numerous variables, like network latency, network throughput, data size and for instance the NFS version and implementation. Experiences show good performance for local and higher bandwidth networks, but rather bad results for long distances and small bandwidths (< 100 MBit). One has to keep in mind, that remote data access techniques always operate on the big end of the visualization pipe, and very often have to transport large amounts of data.

Generality	very general	10
Usability	setup not trivial, usage simple	6
Performance	good on LANs and highspeed WANs	6

3.2.2 File Transfer Protocol – FTP

The *File Transfer Protocol*, FTP, exists since 1971 [13], and is a very well known and established protocol standard for remote file access. It is a Client-Server oriented protocol, and many implementations exist on various software levels (servers, clients, tools, libraries). This makes implementation and usage of FTP based remote data access comparably simple and allows interoperation with existing environments.

To use the remote data access features of FTP, the visualization environment needs to explicitly know about the protocol or its implementation (library). In general, it can't be transparently used as a file system replacement. FTP can be used by normal users – no special system privileges are required.

The main limitations of FTP are caused by its restriction to the transfer of complete files only. Unlike with NFS, a visualization system processing only a subset of a large data set has to transfer the *complete* data set to the local host. This can decrease the performance of the system significantly[4]. Network bandwidth also influences FTP performance more than latency does.

Generality	quite general	7
Usability	simple, compatible, external	5
Performance	good, but only complete files	4

[4]There exist extensions to the FTP protocol that allow partial file transfers [14]. These extensions are well in scope with the original FTP standard, and will hopefully be included in future FTP implementations. This is increasing the usability of FTP for remote visualization significantly.

3.3 Remote and Distributed Data Filters

3.3.1 Advanced Visualization System – AVS

We list a single but very general and well known example of remote and distributed data filtering architectures, the *Advanced Visualization System*[5] [15]. AVS is a complete visualization environment for (mainly) scientific data, that lets the user adapt the early part of the visualization pipeline by freely arranging predefined and customized modules[6]. Thus, the AVS environment follows a data flow concept. Most interesting is that these modules can be distributed over a network, which results in an arbitrary distributed and potentially parallelized visualization pipeline. This modular distribution is very flexible, modules are quite easy to write and to adopt, and can perform various tasks. Backflow data channels allow data flow loops, and steering interactions with all modules. Integration of existing software is very often (but not always) quite simple. The distribution of modules does not require system privileges, and can even span wide area networks (WAN). This results in relatively high marks for generality. Usability is somewhat reduced by the fact that the rendering engine of AVS is not very efficient, and the resulting images are of poorer quality than of competing engines.

The performance of AVS network communication is comparably good for low update rates and not too large data sets. Anyway, the user does not have much opportunities to influence this performance or to reliably estimate it, since most of the mechanisms are completely hidden within the software. For large data sets and larger latencies, we experienced significant performance leaks, even in highspeed LANs.

Generality	quite general, modular, customizable	8
Usability	easy, proprietary, poor rendering	6
Performance	not optimal for large data sets	4

3.4 Remote Geometry Creation

To remotely *create* geometries and *spread* them over the network seems to be a reasonable solution: the amount of data to be transported is extremely small, and due to the potential of geometric scenes to represent basically everything (possibly with added textures) gives hope for very generic solutions.

Alas, we know only of one general approach, VRML, that is self limiting its scope to WWW based technologies, without much reason. Additionally we know about quite a number of more specialized and proprietary techniques. Exemplarily, we will describe DocShow–VR.

[5] We only consider AVS version 4 here, since we have no experiences with the knowingly very different follow up versions.

[6] Most often, these modules perform data filtering, thus AVS is placed in this section – though it covers all other stages of a visualization pipeline as well.

The reason *why* there are only such a few examples of remote geometry generation techniques are not clear. Surely the obvious necessity to additional agreement on a compatible and generic transport technology, as well as on a well defined description language and API adds to this. Nevertheless, the large number of non generic but successfully used approaches do clearly reveal the potential of this approach.

3.4.1 VRML

The *V*irtual *R*eality *M*arkup *L*anguage [16] is designed to describe geometric scenes in a platform independent manner, and to transport (stream) this scene description over the network. By defining a generic file layout, an arbitrary application can relatively easy create VRML scenes; the transport is guaranteed by the well established HTTP protocoll, or in fact by any file transfer mechanism (NFS, FTP, ...); and the visualization of the scene is possible with a number of capable VRML viewers. These viewers usually map VRML geometry descriptions to low level rendering libraries as OpenGL, and than utilize local rendering hardware for optimal rendering speed.

One fact decreasing the generality of this approach is that to our knowledge there exist no libraries to integrate VRML in existing visualization environments. Also, the language definition does not completeley cover well established standards as OpenGL, thus limiting the scope and quality of VRML scenes.

Due to its coding scheme, network transport of scene graph based data streams, a in VRML, is comparably fast. On both ends of the pipeline, specialized local hardware can be used to perform the pre and post processing steps of the rendering pipeline. However, the usage of an ASCII based data stream, as generic and useful as it may be in many aspects, adds some latency due to its preprocessing (parsing).

Generality	very general, arbitrary geometries	8
Usability	simple, standardized, arb. transport	9
Performance	very good, small data, some latency	8

3.4.2 DocShow-VR

DocShow-VR [17] comes as a binary reincarnation of VRML. As it is very young and (not yet) standardized, it combines the basic ideas of geometry description from VRML with a binary real time streaming protocol and some server and caching infrastructure. This makes it more suitable for larger scenes with high update rates, but limits its usage to quite special and custom environments right now. A netscape viewer plugin enhances the scope of usage, though.

Generality	quite general, non standard (yet?)	5
Usability	API, simple setup, good rendering	7
Performance	very good, binary transport	9

3.5 Remote and Distributed Rendering

Distributed rendering is often seen as the future of high end or high performance computer graphics – the movie industry is already deploying this approach in high throughput scenarios in a very successful manner[7]. Although quite a number of issues are still unsolved, watching computer graphics hardware vendors like Sun, SGI and HP, we can expect a number of related solutions in the near future.

Usually, distributed rendering is implemented in a proprietary way, often on hardware level. The high demands on data throughput, caused by the number and size of the resulting images, well justifies this. With WireGL we discuss one more general solution here, which has the potential to serve for many common scenarios.

3.5.1 WireGL

WireGL aims at the utilization of parallel rendering hardware. For achieving this, it distributes the scene to be rendered to the resources by issuing respective GLX calls. The resulting multi image output can be recombined to a single image, or displayed by special multi resolution devices as power walls.

The approach to provide a replacement for the OpenGL-library allows arbitrary applications and visualization systems to fully utilize WireGL. It runs on single CPU systems and scales very well to moderate sized clusters of PCs. Measurements show peak rates of 70 Mega-Triangles per second on commodity 32-PE PC-Cluster! WireGL's limitation of availability on various operating systems (only Linux and Microsoft Windows right now) will hopefully be overcome in the near future.

Generality	limited to GL, very general	9
Usability	new, experimental, IRIX, Win, Linux	4
Performance	very fast 2D/3D, immediate events, scales!!	12

On the other hand, **Remote rendering** techniques are well established by now, and serve various scenarios. But they are all limited by their low 3D performance. We describe some of these techniques, including the OpenGL Vizserver, which is promising to overcome this limitation.

3.5.2 X11-based distribution

The X11 protocol inherits its networking 'capabilities' from its earliest design. All XLib low level calls are able to perform graphic and event actions on remote displays. Security is optionally taken care of by the XAuthority extension. **Every** program utilizing the XLib is automatically network transparent!

[7] i.e., in so called *rendering farms*

Remote X11 performs well for simple (2D) XLib calls. For complex scenes, as well as 3D scenarios, the library overhead becomes significant, not only limiting performance by the required network bandwidth, but also by its need for computing resources on both, the local and the remote host. Additionally, the graphics hardware of the server system is in no means used for rendering, but all XLib calls are evaluated and rendered on the client system[8]. It has to be noted, that most modern visualization systems utilize OpenGL and similar rendering infratructures, and do not rely on X11 only. This limits the usability of remote X11 severely.

Generality	non GL, very general	10
Usability	widely used, flexible, compatible	5
Performance	fast events, decreases with complexity	5

3.5.3 VNC

There are a number of approaches to overcome the remote X11 limitations. The most simple but most successful ones transports the remotely rendered X11 image to the user. Network load is thereby reduced by applying sophisticated techniques for partly updates and online image compression. Others reimplement the remote rendering with customized X-Servers. A famous example of those is the Virtual Network Computer (VNC). VNC [19] comes with an own XServer implementation, that uses the REMOTE FRAME BUFFER protocol (RFB) to render into a remote graphics display. The XServer is based on XFree86, a popular free XServer implementation, which is widely used in the Open Source (e.g. Linux) community. Unfortunately, the minimal VNC XFree version is not a complete one: it is missing several X-Extensions (which should be simple to add), and, even more important, it does not support rendering via the OpenGL network interface (GLX), but limits itself to XLib calls only.

VNC can handle a wide variety of servers, and supports numereous clients: stand alone as well as browser embedded. It performs pretty well even in WAN environments, but highly interactive usage is depreciated by low (but yet smooth) frame rates.

Generality	non GL, very general	8
Usability	widely used, very flexible	7
Performance	fast 2D, fast events	8

3.5.4 GLX

GLX as an extension to the GL standard implements remote rendering functionality similar to remote X11. It allows remote applications to transport GL graphic primitives and rendering instructions to be transported to the local host and to be interpreted by the local graphics hardware. This requires compatible

[8]Strictly spoken, this is not remote rendering

GLX installations on both ends, not being available for all platforms. Due to its standardization and inclusion into the native GL libraries, this approach is quite general and simple to use.

Generality	very general	10
Usability	transparent, not always available	7
Performance	fast 3D, fast events	9

3.5.5 OpenGL Vizserver

The OpenGL Vizserver [20, 21] overcomes several of the limitations of the remote X11 and GLX protocols, but at the same time introduces new limitations. An advantage is that the graphics hardware of the *server* is used in the same effective way as it is for local visualization. It's protocol also includes the possibility to compress images for faster network transfer, just as VNC does (tradeoffs are CPU utilization and quality degradation). Finally, also GL calls can make use of the servers specialized graphics hardware.

The handling of the service is quite convenient on the client side: the user opens a remote session by using some very simple GUI, specifying server host, network interface, and compression parameters. On the server side, the flexible use of the OpenGL Vizserver requires an unmanaged graphics pipe[9]. If one cannot afford leaving one of the pipes unmanaged all the time, some administrative tasks have to be performed every time the OpenGL Vizserver is supposed to be used. The startup files for the display manager must be modified, so that no xdm process will be restarted for a certain pipe after shutdown of the appropriate X server. Afterwards the xdm daemon must get a signal to reread the database. Finally the xdm including X server can be terminated for the desired graphics board.

The OpenGL Vizserver performs very well, and seems to be only limited by the fillrate of the client's hardware. The performance severely suffers from high network latencies, especially for large rendering areas. The following section will describe the OpenGL Vizserver technology in detail, followed by an extensive performance analysis. The OpenGL Vizserver server is available for IRIX only. However, Vizserver clients are currently available for IRIX, Linux and Solaris systems.

Generality	very general	10
Usability	client simple, server tricky	5
Performance	quite fast 2D/3D, immediate events	7

3.6 Remote Display Techniques

Remote Display techniques are in fact remote rendering techniques seen from the 'other' side. They usually concentrate on the fact that a data display and the

[9] i.e., no X display manager (xdm) is running for that board.

user may be remote from the source of the images to be displayed, and therefore optimize the data transfer. Additionally to the remote rendering techniques we present a hardware solution from Lightwave as a representative example of such techniques.

3.6.1 LightWave

Lightwave [22] and similar companies offer commercial hardware solutions to allow to remotely access the display of high end graphics servers. The basic idea is to transport the raw display device signal over 'long' distances. Input events are transported in opposite directions. The distance is hereby limited by signal loss and synchronization problems, and is usually limited to below 100 meters. Image quality loss occures by introducing shadow images, color shifts and image noise. These effects are neglectable for short and medium distances.

Generality	very general	10
Usability	limited range, lowers quality, expensive	3
Performance	as local	10

4 The Silicon Graphics OpenGL Vizserver

The OpenGL Vizserver is a remote rendering software, that allows users to take advantage of the graphics perfomance of an SGI Onyx Infinite Reality system on conventional desktop workstations. The OpenGL Vizserver enables users to view and interact with large data sets from a desktop system like for instance a Silicon Graphics Octane, O2, or even other UNIX desktop systems (Linux, Solaris). A client for Microsoft Windows NT is announced but not yet available.

With OpenGL Vizserver, graphics processing is handled entirely on the Onyx system. The only fundamental requirement of the client will be a true color display. The OpenGL Vizserver transmits either uncompressed or compressed images from the Onyx frame buffer via standard network connections to a Vizserver client running on a desktop workstation. The OpenGL Vizserver is almost transparent to the user, and it can also be used via an API for remote visualization software development.

OpenGL Vizserver offers the following features:

- Ability to transmit images directly from the Onyx frame buffer
- Ability to run over standard networks
- Application transparency

4.1 How does the OpenGL Vizserver work?

The OpenGL Vizserver concept bases on a server software running on a high performance graphics computer like an Onyx II or higher, and a client software running on any desktop computer system with a true color graphic subsystem.

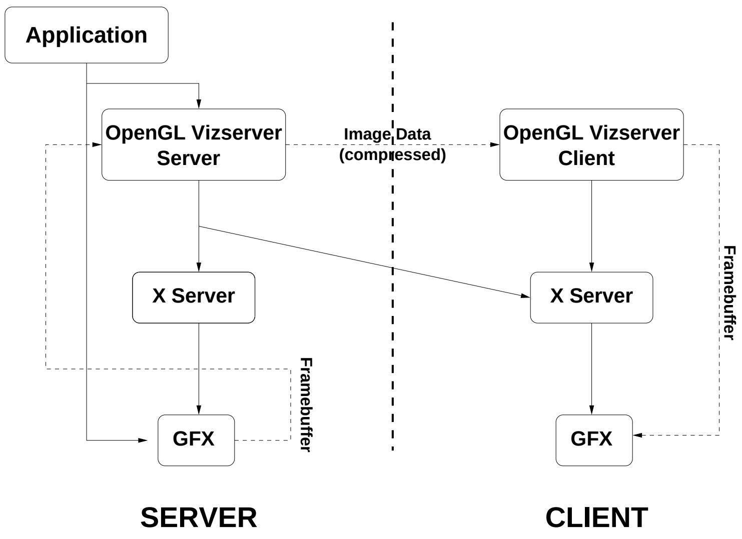

Figure 2: X11 application using OpenGL Vizserver [21]

The rendering takes place on the Onyx system and the visual result, usually displayed as the frame buffer's content on a monitor directly connected to the Onyx pipe, is now transferred via network under control of the OpenGL Vizserver software. The frame buffer contents can be compressed for low speed network connections or remote visualization via wide area networks (WAN).

4.1.1 Data Compression

Two compression techniques are currently supported by the OpenGL Vizserver. Either Color Cell Compression (CCC) or Interpolated Color Cell Compression (ICC). CCC has a compression ratio of 8:1 and ICC is 4:1. Both algorithms reduce the size of the image by reducing the color space of small regions within the image and therefore result in a lossy compression scheme. The two schemes are derived from the Block Truncation Coding (BTC) algorithm which compresses a 4×4 pixel block down to two colors plus a 4×4 pixel mask [27, 28].

The variation between the CCC and ICC compressors lies in the number of colors identified by the pixel mask. In the case of CCC, the mask identifies two colors. The ICC mask identifies four colors. In both cases, the colors components are converted to 5 bits of red, 6 bits of green and 5 bits of blue for 16 bits per pixel. The compression rate for these two schemes is 8:1 for CCC and 4:1 for

ICC. This fixed compression rate has the advantage that compression time is also fixed, offering deterministic, image independent latency. This yields better behavior for interactive applications.

On the client side the images are decompressed if necessary by the OpenGL Vizserver client software and then directly copied into the frame buffer again. This way the rendering of complex scenes with texture mapping, anti aliasing and so forth is done using dedicated graphics hardware and just the rendering result is transferred to where it can be displayed. Stereo rendering, a convenient feature of Onyx systems, is currently not supported by the OpenGL Vizserver.

4.1.2 Frame Triggering

Exactly four low level GL calls are triggering a new image readout and transport: `glXSwapBuffers(), glFinish(), glFlush() and glXWaitGL()`. If one of these events occures, the rendered image is read back from the framebuffer into main memory, encoded, compressed and finally sent to the client. There it goes via decompression and decoding from main memory into the framebuffer.

This process is throughput limited by several components. First, the pure frame rate of the server hardware will define the maximum number of triggering calls to occur. This is mainly limited by scene complexity and rendering area. The image readout, its encoding and compression is independent of scene complexity, but depends on the fill area. As far as we understand, the minimum of rendering rate and readout/preprocessing rate is finally used – both processes are in synch to each other.

The network transport is clearly limited by throughput, and hence depends on the fill area (data size) and the chosen compression rate. If network congestions occur, and the network image transport rate drops below the servers image processing rate, the server drops processed images until the network buffers are free.

On client side, the decoding and decompression are limited by the fill area again, as well as the final image display. It is not clear to us what happens if the client cannot handle incoming images fast enough (due to high CPU load for example). We expect that the resulting network congestion on receiver side triggers a similar one on sender side, and agains leads to the dropping of images on server side.

Adjusting `tcp` buffer sizes (increasing) and the `ack` timeouts (decreasing) should help here, although we could not measure significant differences.

4.2 Installation and Setup of the OpenGL Vizserver

Using the OpenGL vizserver is almost transparent to the user but requires a little configuration work for the systems administrator. The major requirement is a free graphics pipe, i.e. not managed by any display manager. Therefore all boards designated for being under the control of the OpenGL Vizserver must not be listed within /usr/X11/lib/xdm/Xservers or any similar `X11` configuration or startup file.

4.2.1 Installation of the server software

We did install the server software on our OnyxII and OnyxIII systems via SGI's software manager `swmgr`. The software goes into the directories /usr/vizserver/ and /var/vizserver. In a first step at least one configuration file /var/vizserver/users has to be edited to make the OpenGL Vizserver publicly available. Details on giving users permission to connect to the vizserver or the assignment of graphics boards can be found within the manual page `man vsserver`[10].

To start the server, the vizserver option must be enabled via chkconfig vizserver on. This will cause the server to startup automatically when the system is transitioned to multiuser mode (usually at system boot time). The startup script for the OpenGL Vizserver server manager is located in the file /etc/init.d/vizserver. This script takes a single argument which indicates whether to start, stop, restart or update the server manager.

Having at least one of the Onyx's graphics boards not managed by the X11 display manager (xdm), and the X11 server as well as the OpenGL vizserver running, enables all users listed in /var/vizserver/users to connect to the OpenGL Vizserver with the appropriate client software.

4.2.2 Installation of the client software

After installation and configuration of the server software, we did install the OpenGL Vizserver client software, that can be downloaded from SGI's web site[11], on different machines. The client software is currently available for IRIX, Solaris and Linux. A Microsoft Windows NT/2000 client is highly appreciated. On IRIX systems the client software will be located within the directory /usr/sbin. It can be started with the command vizserver.

4.2.3 Using the OpenGL vizserver client

At startup of the OpenGL Vizserver client software, the main window is presented requesting the hostname of the server system. In a second window an authentication dialog for remote login on the server system appears. After succesful login[12] a session can be started or stopped from the main window. This window can be left active on the desktop and users can login to use the vizserver or logout to release the vizserver for other user's connections.

[10] vsserver is responsible for starting and managing server sessions.

[11] `www.sgi.com/products/evaluation/index.html#gl_viz`

[12] Users have to have an account on the server system as well as an entry in the vizserver's config file.

An OpenGL Vizserver session is initiated after some configuration parameters have been set in a third dialog window. One can choose the pipe, a network interface and a compression mode. The latter can be changed on the fly, even during an active session, the other parameters remain constant. A session simply pop-ups a terminal window. The only difference between any other terminal window, remotely started on this server, is the content of the `DISPLAY` variable, indicating that one gets display results directly from the Onyx's graphics adapter. Now any application can be started from this window using the graphics hardware, like running this software directly on the Onyx console. Thus higher frame rates can be achieved and hardware acceleration, harware antialiasing as well as the texture buffer can be used, on ordinary and inexpensive workstations.

5 Evaluation of the OpenGl Vizserver

Our goal was to determine the maximum frame rates for remotely rendering 3D graphics with our visualization software Amira [29]. The basic reason for using the OpenGL Vizserver is to provide high perfomance graphics to remotely connected users in a distributed visualization environment. Generally speaking the OpenGL vizserver only makes sense when one can afford leaving an (expensive) graphics pipe of an Onyx computer system reserved for distributed use. Having more than one user occasionally but not simultaneously needing high performance graphics is the ideal prerequisite.

We tested the visualization performance on several SGI workstations (Indigo, O2 and Octane) with differences in memory, graphics and network hardware, connected via TCP/IP networks (LAN, WAN). For optimal performance, SGI recommends that the OpenGL Vizserver network support 100 Mb/s bandwidth. We tested LAN and WAN connections ranging from 10 and 100 Mb/s up to 1 Gb/s. We performed measurements with varying visualization hardware and networking connections. Additionally, the compression and resolution parameters of the visualization have been varied. We measured performance and network load. All results were compared to performance results for local visualization on the Onyx system (see. section 6).

5.1 Network Configurations

Measuring remote visualization performance naturally depends heavily on available network bandwith and latency. We tried to evaluate the different remote visualization techniques listed above on a variaty of network technologies, including: 10/100 MBit Ethernet, and OC12/OC48 ATM.

Our local network configuration enables us to dedicate the available network bandwith to the application in many cases. As a real world example, we also measured performance on a WAN, whereby no control over network QoS was given.

5.2 Graphic Configuration

Our test configuration consisted of an Onyx II with Infinite Reality-2 graphic on server side. The client side varied – we used an Indigo-2 Maximum Impact with MIPS R4/250, several O2 (MIPS R5/180 – R12/300) with O2-Graphics, and an Octane MXI R10/195. Since the client side is only displaying 2D-Images, the overall system performance (CPU, memory) turned out to be more important than the graphics hardware.

6 Performance Measurements and Outlook

We made a number of tests to measure the performance of the OpenGL Vizserver in detail, and estimate its dependencies from various components as available graphic hardware, network infrastructure, scene complexity, fill area and data compression. The results are not included in this *extended abstract*, but will be delivered soon, replacing this document without notice.

7 Acknowledgements

This work has been greatly supported by Silicon Graphics Inc., which provided us with the newest versions of the Open GL Vizserver, and also provided numerous detailed information about its internals. Especially we would like to thank Michael G. Brown for his insight into the vizserver's frame triggering pipeline.

Several of our measurements have been performed between the Konrad Zuse Institut Berlin and the 'Klinikum Rechts der Isar' in Munich. We'd also like to thank Dr. Robert Sader for providing access to their equipment, and Dr. Victor Apostolescu (Leibniz-Rechenzentrum in Munich) for his support in networking questions and measurements. We are also grateful to the German Research Network (DFN) for providing the fast WAN infrastructure that is of utmost importance for our work.

References

[1] Ian Foster, Carl Kesselman (eds.): *"The Grid: Blueprint for a Future Computing Infrastructure."* Morgan-Kaufmann (1999)

[2] *"Advanced Collaborative Environments - Global Grid Forum Working Group."* `http://calder.ncsa.uiuc.edu/ACE-grid/`

[3] Werner Benger, Hans-Christian Hege, Andre Merzky, Thomas Radke, Edward Seidel: *"Efficient Distributed File I/O for Visualization in Grid Environments."* PDC Proceedings, "Simulation and Visualization on the Grid",

Lecture Notes in Computational Science and Engineering, PDC'99, Springer, (2000)

[4] IEEE Computer Graphics & Applications, Special Issue on Large Scale Data Visualization, Vol 21(4), July/August (2001)

[5] Mustafa Demirtas, Stefan Zachow: *Comparison of Visualization Software for Medical Image Data.* Technical Report: TR-MKG-SRL 001/97, Surgical Robotics Lab, Charité; - Campus Virchow, Berlin (1997) `http://www.charite.de/rv/mkg/srl/reports`

[6] Heidrun Schumann, Wolfgang Müller: *"Visualisierung - Grundlagen und allgemeine Methoden."* Springer, (2000)

[7] *"The X Windows System."* `http://www.x.org`

[8] *"The XFree86 Project, Inc."* `http://www.xfree.org`

[9] *"The Visualization ToolKit."* `http://public.kitware.com`

[10] *"Open Inventor."* `http://oss.sgi.com/projects/inventor`, SGI

[11] *"OpenGL – High Performance 2D/3D Graphics."* `http://www.opengl.org`

[12] *"The Network File System."* `http://globecom.net/ietf/rfc/rfc1094.html`, IETF

[13] *"The File Transfer Protocol."* `http://globecom.net/ietf/rfc/rfc172.html`, IETF

[14] *"The GridFTP Protocol and Software."* `http://www.globus.org/datagrid/gridftp.html`, GGF

[15] *"Advanced Visual Systems."* `http://www.avs.com`

[16] *"The Virtual Reality Markup Language"* `http://www.web3d.org/`

[17] Stephan Olbrich: *"Ein leistungsfähiges System zur Online-Präsentation von Sequenzen komplexer virtueller 3D-Szenen."* Fachbereich Elektrotechnik und Informationstechnik der Universität Hannover, Dissertation (2000)

[18] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett and Pat Hanrahan: *"WireGL: A Scalable Graphics System for Clusters."* Proceedings of SIGGRAPH (2001)

[19] *"The Virtual Network Computer."* `http://www.uk.research.att.com/vnc`

[20] *"The OpenGL Vizserver."* `http://www.sgi.com/software/vizserver`

[21] C. Ohazama: *"OpenGL Vizserver."* White Paper, Silicon Graphics Inc., (1999) http://www.sgi.com/software/vizserver/whitepapers.html

[22] *"Lightwave Communications."* http://www.lightwavecom.com

[23] Gabrielle Allen, Werner Benger, Hans-Christian Hege, Joan Massó, Andre Merzky, Thomas Radke, Edward Seidel, John Shalf: *"Solving Einstein's Equations on Supercomputers."* IEEE Computer, 32:12 pp. 52-59 (1999)

[24] Werner Benger, Hans-Christian Hege, Andre Merzky, Thomas Radke, Edward Seidel: *"Efficient Distributed File IO for Visualization in Grid Environments."* in: L. Johnsson, M. Hammill, F. Short (eds.), *Simulation and Visualization on the Grid*, Lecture Notes in Computational Science and Engineering, Vol. 13, Springer, pp. 1-16 (2000)

[25] K. Engel, O. Sommer, and T. Ertl. *"An Interactive Hardware Accelerated Remote 3D-Visualization Framework."* Proc. of EG/IEEE TCVG Symposium on Visualization (VisSym), May (2000)

[26] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl. *"Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications."* To appear in: Procceedings of IEEE Visualization (2000)

[27] G. Campbell, TA. DeFanti, J. Frederiksen, SA. Joyce, and LA. Leske: *"Two bit/pixel full color encoding."* Proceedings of the 13th annual conference on Computer Graphics (Siggraph), pp. 215–223 (1986)

[28] G. Knittel, A. Schilling, A. Kugler, and W. Straßer: *"Hardware for Superior Texture Performance".* Computers and Graphics, Vol. 20, No. 4, pp. 475–481 (1996)

[29] *Amira: An Advanced 3D Visualization and Volume Modeling System.* http://amira.zib.de (2001)