Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2002)
D. Ebert, P. Brunet, I. Navazo (Editors)

A Generic Solution for Hardware-Accelerated Remote Visualization

Simon Stegmaier, Marcelo Magallón and Thomas Ertl

Visualization and Interactive Systems Group, IfI, University of Stuttgart
Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany
(stegmaier|magallon|ertl)@informatik.uni-stuttgart.de

Abstract

This paper presents a generic solution for hardware-accelerated remote visualization that works transparently for all OpenGL-based applications and OpenGL-based scene graphs. Universality is achieved by taking advantage of dynamic linking, efficient data transfer by means of VNC. The proposed solution does not require any modifications of existing applications and allows for remote visualization with different hardware architectures involved in the visualization process. The library's performance is evaluated using standard OpenGL example programs and by volume rendering substantial data sets.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Distributed/network graphics, C.2.4 [Distributed Systems]: Distributed applications

1. Introduction

Scientific simulations tend to produce ever-growing amounts of data. Analyzing these data on the researcher's local host poses two problems: first, local analysis is impossible if the resources needed to visualize the data exceed the local hardware capabilities; second, transferring the data each time new simulation data is available is tedious if the network-bandwidth is low. These problems lead to an increased interest in remote visualization.

The common approach for remote visualization employs the OpenGL remote rendering facilities. In this approach the application is run remotely while the local host does the actual rendering. This solution is application-independent and therefore very popular, but it does not benefit from special graphics hardware features available at the remote host such as 3D textures and large amounts of texture memory. As a concrete example, in the system presented by Magallón et al[11], a cluster of Linux PCs equipped with graphics accelerators renders and composes images which are presented to the user. In this situation, traditional OpenGL remote rendering performs poorly because of the high bandwidth requirements it imposes on the network between the cluster and the user's workstation. Several solutions to tackle this shortcoming have been developed (e.g., Bethel[1], Engel[3, 4, 2], Ma[10], among others), but all of these lack application or architecture independence.

In most cases generic solutions perform worse than specific solutions for the same problem, so why should one care about universality? Specific solutions require the programmer to know the internal workings of the application to be adapted. This often requires source code availability, thus commercial applications do not come into question for specific solutions. And even if the source is available, it may be hard for a programmer to modify an application originaly implemented by someone else. So in this case, too, a generic solution might be preferable.

For these reasons this paper presents an application- and architecture-independent solution for hardware-accelerated remote visualization. It is based around the well known concept of *dynamic linking*[6] and exploits characteristics of the *X Window System* and its associated protocol. The idea is to redirect OpenGL rendering requests to the hardware where the program is running. Once the rendering is done, the resulting image is read off the framebuffer and sent over the network to the display together with all other GUI elements. Since all the OpenGL rendering happens on the remote display, it is possible to use an existing OpenGL-based application without requiring the availability of OpenGL-capable hardware locally, while taking advantage of advanced graphics features as well as hardware acceleration by the *remote* hardware. For example, this solution can be used to provide remote access to distributed rendering facilities such as those presented by Magallón et al. By leveraging already existing remote-access applications, our proposed solution also makes it possible to access remote visualization resources from operating environments that do not provide implementations for the required protocols by default.

The rest of this paper is organized as follows: first, other solutions for remote access of graphics hardware are analyzed; then the necessary background regarding dynamic linking and OpenGL rendering is visited; afterwards, the architecture of our solution is explained; in the last part, our results are presented.

2. Previous Work

Silicon Graphics, Inc. provides a commercial solution called OpenGL VizServer[16], that enables lightweight clients such as O_2 and PC workstations to access the rendering capabilities of SGI Onyx servers. Because of design decisions, other architectures cannot be used as servers for this application. Similarly to our solution, the VizServer relies on dynamically linked executables in order to be able to implement its functionality without modifying the target application.

Ma and Camp[10] developed a solution for remote visualization of time-varying data over wide area networks. It involves a dedicated display daemon and display interface and display daemon. The first receives data from a render process, compresses it and passes it to the second, which in turn decompresses the data and presents it to the user. By using a custom transport method, they are able to employ arbitrary compression techniques. Bethel[1] presented Visapult, a prototype system developed at Lawrence Berkeley National Laboratory that combines minimized data transfers and workstation-accelerated rendering. Visapult also requires modifications of the application in order to make it "network aware" and relies to some extent on the existence of hardware graphics acceleration on the local display. Engel and Ertl[2] developed a solution for remote collaborative volume visualization which exploits the characteristics of the application domain to reduce latency as well as required network bandwidth. Engel et at[3] further developed this approach and implemented a hybrid rendering mechanism to obtain better framerates.

Another solution for remote rendering is included with the Open Inventor 3.0 toolkit offered by TGS (http://www.tgs.com). The solution is very similar to our work in that it addresses GLX and X requests to two different X servers and utilizes the VNC (virtual network computing) client-server infrastructure for image transmission. Based on the available information, the solution seems to be restricted to Open Inventor.

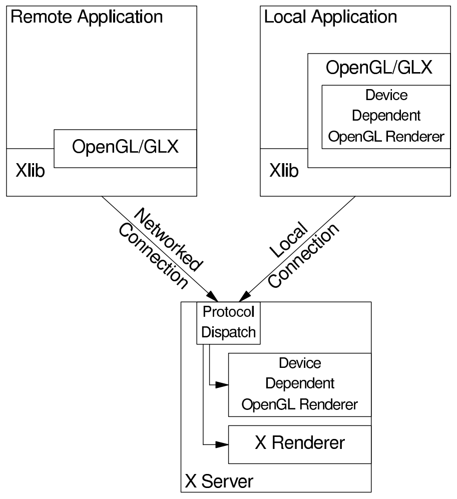

Figure 1: *GLX architecture as presented by Kilgard*[7]

3. Background

3.1. OpenGL Graphics with the X Window System

GLX[17] is an extension to the X protocol[12] that allows clients to create a so-called "GLX context" which can be used to issue OpenGL calls that can be executed using either a hardware-accelerated rendering engine or a software-based one. By sitting on top of X, network transparency is obtained for free. A GLX context can operate in either *direct* or *indirect* mode. In indirect mode, the client sends requests to the X server which propagates them to the hardware. In direct mode, the X server only functions as a marshal making sure that the OpenGL state of each client is kept consistent. Since the X protocol is bypassed in direct mode, OpenGL rendering can achieve the maximum performance of the hardware[8]. Direct rendering implies that the process is running on the same machine as the X server and not over the network. In the case of remote rendering, only indirect mode is possible, if at all. Figure 1 shows both cases.

3.2. Dynamic Linking

In modern systems, programmers have the choice of *statically* or *dynamically* linking programs during compilation. In static linking, all the object references of a program are resolved during the link phase of the compilation. In contrast, with dynamic linking, the executable file contains references to, but not the actual code of the required library functions. In this case, symbol resolution is carried out during load-time. The dynamic linker is in charge of finding these references in the executable and *resolving* them using a list of *shared object names* (libraries) that the executable contains. One of the advantages of load-time linking is that it facilitates code reuse, it simplifies the task of fixing bugs and reduces the memory requirements imposed on systems, since code pages can be shared among unrelated processes. By its very nature, it also enables users to replace libraries with custom versions designed to modify the behavior of a program. The only requirement in this case is to keep the application binary interfaces unmodified. Since re-implementing all the interfaces of a library can be cumbersome and tedious, some systems offer the possibility to load arbitrary lists of shared objects code before the required libraries are loaded, with the purpose of selectively overriding functions in other shared objects. This process is called *preloading*. As will be explained later, this simplifies the implementation of our solution.

In addition to load-time linking, it is also possible to perform *runtime linking*, as first described by Ho and Olsson[6] and later explained in the Linux/GCC case by Lu[9]. In this approach, additional objects can be opened at run-time and symbols can be selectively added to the running program. The most widely used interface for this purpose is `dlopen(3)`, available on systems such as IRIX, Linux, and Solaris, among others. This can be used, in concurrence with the preloading feature mentioned before, to wrap code around the original library functions: first the function is overridden using preloading, then its original code is recovered using dynamic linking and it is used by the customized version of the function to provide the original behavior if necessary.

4. Implementation

By taking advantage of the dynamic linking facilities explained above, it is possible to modify the behavior of any given program for the X Window System without changing its source code or that of the libraries it uses. In our approach, there are two X servers involved: one that supports GLX and a second one, which takes care of the user interaction, and which does not necessarily support GLX. In the following discussion, the first X server will be called *render server* and the second one the *interaction server*. *Display* will be used in the same way the X Window System defines it.

The application is started locally to the render server but its environment is configured to have it displayed on the interaction server. It is loaded in such a way that a custom version of every GLX function is used, whose job is to redirect GLX requests to the render server. Since the display is part of the GLX context's properties, OpenGL calls are automatically redirected to the render server. In a sense, a *new library* is inserted between the application and the system's OpenGL library, as depicted in Figure 2. Once the application requests a buffer swap, the contents of the framebuffer is read and written to an `XImage` structure, which is sent to the interaction display via a `XPutImage` request. User interaction works transparently since events are transported between the interaction X server and the render server without modification. A more detailed description of customizations required is provided in the rest of this section.

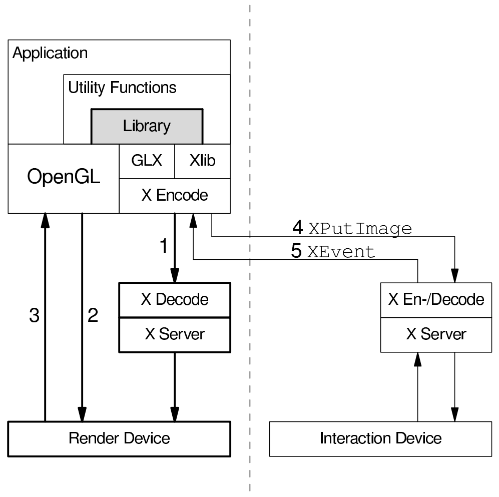

Figure 2: *System architecture. (1) The application issues a GLX request which is send to the render server. (2) The application issues OpenGL calls, which are handled by the render device. (3) The library reads the contents of the framebuffer and (4) sends it to the interaction server using a* `XPutImage` *request. (5)* `XEvent`*s are sent from the interaction server to the application.*

4.1. Customizations for `Xlib` Functions

Whenever the application calls `XOpenDisplay` to open the interaction display, the custom version of this function opens it as well as a second one on the render host. A `Display` structure pointing to the requested display is returned, which ensures that the normal operation of the program is not disrupted. The render display is closed when the number of calls to `XCloseDisplay` matches the number of calls to `XOpenDisplay`.

4.2. Customizations for GLX Functions

For most GLX functions the only required change is redirecting the request from the interaction server to the render server. Only a few functions have to be treated in a special way. In this discussion, it is assumed that the client is using GLX 1.2. For later versions a similar implementation applies.

`glXChooseVisual` is used to select a visual that matches the attribute set specified by the application. The application is neither required to use the visual returned by `glXChooseVisual`, nor is it prevented from calling it multiple times. The visual `glXChooseVisual` returns has to be valid on the *interaction server*, since it will be used to create a widget there. The custom version of the function matches visuals across the two X servers and returns the best visual that is compatible with the given attribute set. In the current implementation color depth is used as the metric, but this is not the optimal approach in the general case. If no visual is found in these conditions, it is up to the application to cope with the situation. The attribute set passed to `glXChooseVisual` is saved for upcoming calls of `glXCreateContext`.

When the library creates a drawable for rendering, it tries to use in-hardware off-screen preserved buffers (called *preserved PBuffers*) and falls back to regular windows if these are not available. PBuffers are preferred because they are not obscured by other windows. The use of PBuffers has to be specified upon context creation using `glXCreateContext` but the size of the PBuffer is specified later with a call to `glXCreatePBuffer`. This later call might fail because of insufficient resources. Since the library cannot obtain the size of the drawable preferred by the application until `glXMakeCurrent` is called, a situation is possible where a PBuffer context is created but is later unusable. This situation can be avoided by noticing that the value returned by `glXCreateContext` is a pointer to an opaque data type. This allows the library to generate its own value when the application calls `glXCreateContext` and defer the actual creation of the context to the moment when the application calls `glXMakeCurrent`.

When the application calls `glXSwapBuffers`, buffers are swapped, the rendered image is read from the framebuffer and transmitted to the interaction server as an `XImage`. The normal image transport method, `XPutImage`, incurs a high overhead because the data is read from the client's memory space and is copied to the X server memory space which hinders performance significantly. If the MIT Shared Memory Extension is available, `XShmPutImage` is used instead. The XShm extension cannot be used if shared memory is not available, as it is normally the case when the client and the server run on different hosts. If the application does not use double buffering, it is necessary to use a heuristic to determine when to send the images to the interaction X server. A first approximation would be to use one of `glXWaitGL`, `glFlush` or `glFinish`, but in practice this has proven to work unreliably: several tested clients do not issue `glXWaitGL` calls and others issue too many synchronization calls.

5. Discussion

5.1. Optimizations

Reading the rendered images from the framebuffer and sending them over the network are expensive operations and reduce the maximum achievable frame rates of the applications that use the library. This is especially undesirable for interactive applications where frame rates are low even when run locally, e.g. volume renderers. Reading the framebuffer can be considered an atomic operation that cannot be optimized. Therefore, frame rate increases must be achieved by optimizing image transmission and generation.

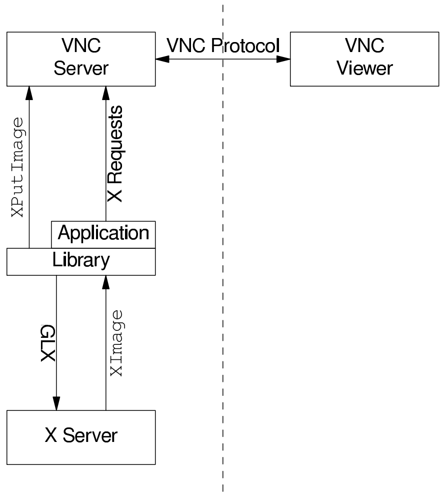

Figure 3: *System architecture when VNC is included*

The core X protocol does not include any form of image compression. This has been implemented via an extension oriented towards low bandwidth environments called LBX[5]. Using LBX on a local area network we experienced reduced network traffic but no performance gains. Another way of obtaining stream compression on top of X is VNC[14]. VNC is a free multi-platform client-server application for displaying and interacting with remote desktops. The protocol underlying VNC is only capable of sending rectangular framebuffer updates to the client. VNC provides a variety of specifically designed compression algorithms to make this transmission as efficient as possible. The Unix variant of the VNC server is based upon a standard X server, which means that on the one hand it can communicate with clients using the X protocol but on the other hand that it does not implement any X protocol extensions, especially GLX. Figure 3 shows how VNC can be used with our library. Given this configuration we measured a frame rate increase of up to 70 percent when compared to the results measured using display redirection for image transmission. Still the experiments show that use of VNC's hextile is only advantageous if the image to be transmitted has large areas of uniform color.

To take advantage to the VNC's hextile compressor characteristics, the image can be downscaled to one fourth of its original size and then it can be sent to the X server with each pixel quadrupled. This allows to hextile compressor to achieve higher ratios but compromises image quality. From the user's point of view, loss of image quality might be acceptable while modifying the scene, but not while analyzing the image on the screen. To implement this idea, a custom version of `XNextEvent` is supplied which takes note of mouse activity in the OpenGL rendering area. When the interaction starts, downsampled images are sent to the server and once it stops, a normal image is sent. Figure 4 shows

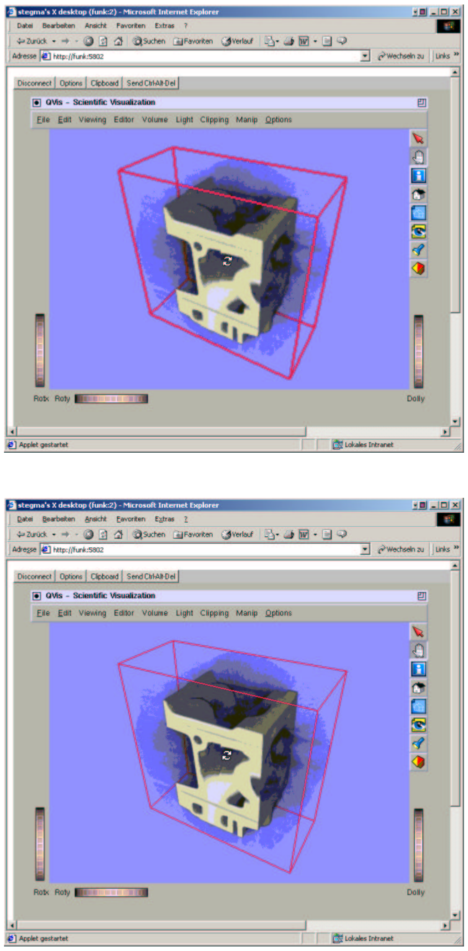
stegma's X desktop (funk:2) - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Verlauf
Adresse http://funk:5802
Wechseln zu Links
Disconnect Options Clipboard Send Ctrl-Alt-Del
QVis - Scientific Visualization
File Edit Viewing Editor Volume Light Clipping Manip Options
Rotx Roty
Dolly
Applet gestartet
Lokales Intranet

Figure 4: *Remote visualization using VNC for image compression and a web browser as VNC client. The effects of down-sampling can be seen in the upper image.*

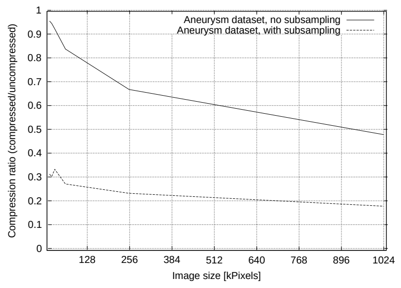

Figure 5: *VNC compression ratio for medical image data with and without downsampling*

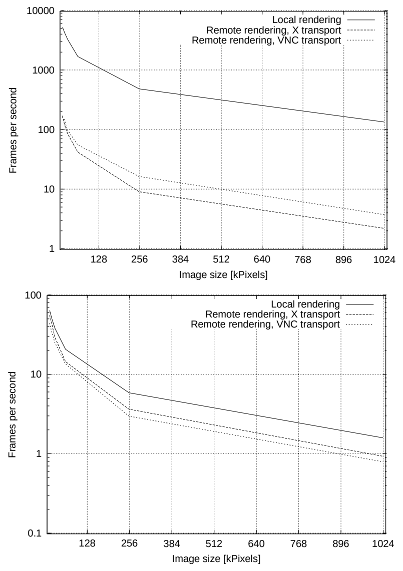

Figure 6: *Measured frame rates for "gears" (upper plot) and a volume renderer (lower one)*

a comparison of image quality using full and reduced resolution (see color plates). As long as the image does not contain areas of high frequency the image quality is good enough for normal interaction. Figure 5 shows a comparison of compression ratios using the VNC hextile compressor. This approach is still work in progress since a better way of detecting "user interaction" has to be found.

5.2. Results

The upper diagram in Figure 6 compares the library's performance with Brian Paul's GLX port of gears over a Fast Ethernet connection. The results are shown as a function of window dimensions (all windows had the same width and height) for different transport mechanisms (X display redirection and VNC). The application's frame rate in the local case is provided for comparison.

The lower diagram shows the results of the same measurements using a texture-based volume renderer[13] and a $256 \times 256 \times 128$ volume data set of an aneurysm. Again the frame rates are shown for different transport mechanisms and several window dimensions. In addition, a characteristic for the frame rate when using reduced resolution during object movement is shown.

As can be seen frame rates using our framework are quite acceptable for CPU-bound applications but drop sharply when applied to an applications with high communication demands. This suggests that the frames rates mainly depend on the bandwidth of the available network.

The render server that was used for all measurements was equipped with a 900 MHz AMD Athlon CPU, 256 MB RAM and a GeForce 2 from NVIDIA. The interaction server was equipped with a 1.2 GHz AMD Athlon CPU, but since the interaction server was rarely operated at more than half capacity a much slower processor would also have sufficed. Both systems were running a Linux 2.4.10 Kernel and were connected using 100 Mbps Fast Ethernet.

For the VNC experiments we used VNC 3.3.3r1 and the VNC native viewer (instead of a web browser) for displaying the rendered image. The viewer was run with the VNC standard data encodings (`hextile` and `copyrect`) enabled. In addition, the option `-truecolor` to request true color visuals was specified. Accordingly, to provide the desired color values, the VNC server was run using a color depth of 24 bits. Deferred updates were disabled.

The frame rates reported in this section are the frame rates as reported by the applications on the render server. Using VNC these frame rates do not exactly match those measured by the interaction server, but since the error is very small (the frame loss is about 0.1 percent at 600 fps and zero at 20 fps) validity of the results can still be assumed.

In conjunction with VNC, the library permits remote visualization and user interaction on a pocket PC (Figure 7, see color plates). Using the well-known engine data set ($256 \times 256 \times 110$ voxels) we measured about 3 fps during with 8 bit colors and an image size of about 120 pixels squared.

5.3. Limitations

Our approach is not as generally applicable as we would have liked it to be. The first problem is that visuals have to be matched across X servers with potentially incompatible sets of visuals. An X server that does not support the GLX extension will not present as many visuals to the applications as one that does. When the GLX extension is not present, each visual has a class (e.g. direct or truecolor) and a color depth. The GLX extension adds properties such as bits per pixel for the framebuffer and the ancillary buffers. If the application relies on `glXChooseVisual` to select the visual it will use, there is no problem. Since the application is not aware of the existence of the render X server when it tries to study the properties on each of the available visuals, our library has to decide which subset of visuals on the render server it will present to the application. When using VNC the problem becomes more evident, since the VNC server presents only one visual to the application. This problem can be solved by telling the library explicitly which visuals to use. In any case, this only emphasizes the fact that our library uses the GLX API for something that is not part of its design and there might be legal uses of the API which we have not considered.

The second problem is the reliance on the availability of dynamic linking. Some Unix variants provide only a subset of the required features, in particular they lack library preloading. Even without this feature, it is still possible to implement the functionality by providing a custom version

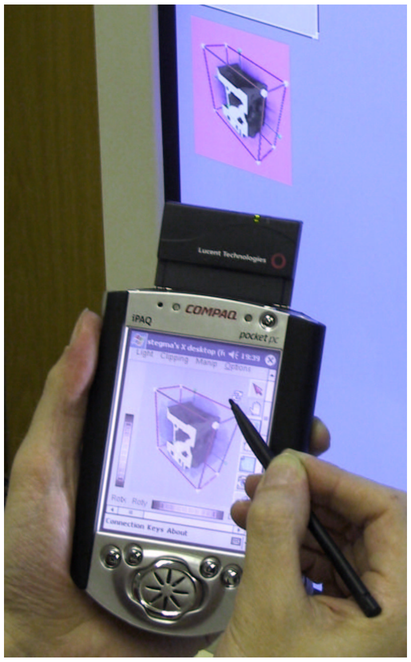

Figure 7: *Interaction with a volume dataset on a pocket PC connected to the render server via 11 Mbps wireless LAN. As can be seen in the background, the render server displays only the GLX drawable and not any of the application's widgets.*

of the OpenGL library that uses `dlopen` to link to the system's version of it. This introduces the overhead of one extra function call per OpenGL function. Another solution, in case a `dlopen`-like interface does not exist at all, could be implemented as a proxy for the X server. It would have to decode the X stream, extract the GLX commands out of it and reencode the remaining ones. There would be a measurable overhead associated with this, mostly because the stream would have to be kept in a consistent state.

Even when using VNC, there is a major performance degradation because of the required network bandwidth, as we discussed before.

The standard VNC applet client quantizes colors to 8 bits, which produces artifacts as seen in figure 8 (see color plates). The shown application uses the terrain visualization algorithm from Röttger et al[15].

Security is also a problem. For this approach to work, the X connection to the render server has to be authorized. For practical purposes, this means either that users are trusted or that exclusive use of the remote display is granted. Standard authentication features of modern Unix variants can be used or extensions to the X security mechanisms can be developed to overcome this problem, but it is not completely solvable since the granularity of the X security model is coarse. This

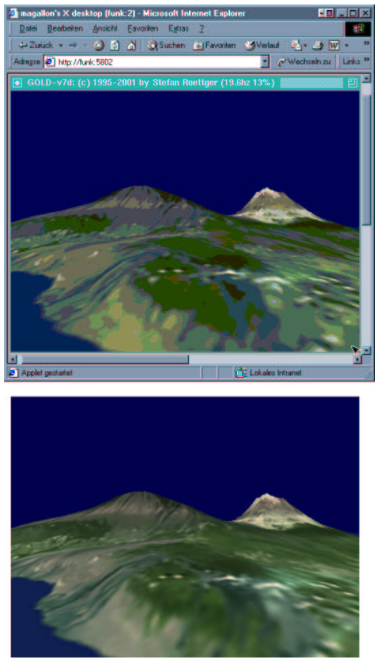
magallon's X desktop (funk:2) - Microsoft Internet Explorer
http://funk:5802
GOLD-v7d: (c) 1995-2001 by Stefan Roettger (19.6hz 13%)
Applet gestartet
Lokales Intranet

Figure 8: *Remote terrain visualization. The effects of color quantization can be seen in the upper image.*

problem has to be handled via site policy. Similarly, connections from X server to X server, or between the VNC client and server are not encrypted and are subject to eavesdropping. Possible methods to solve this include the use of technologies such as IPsec and SSH.

6. Conclusions

In this paper we propose a generic solution for hardware-accelerated remote visualization. The presented library serves the visualization community in two ways. First, it shows that any OpenGL application can be used for remote visualization—whether source code is provided or not—as long as the required visuals can be matched. Second, it demonstrates how cross-platform remote visualization can be done without the need to re-invent suitable image transmission and compression algorithms. Bearing in mind that frame rates are limited by network bandwidth, we expect our solution to become even more significant as soon as modern high-speed interconnects find their way into our daily life.

Acknowledgements

We would like to thank Klaus Engel for providing us with a volume rendering application and access to a pocket PC which served as test bed for our framework and Stefan Röttger for making available his terrain rendering application to us.

References

1. Wes Bethel. Visualizaton dot com. *Computer Graphics and Applications*, 20(3):17–20, May/June 2000. 1, 2

2. K. Engel and T. Ertl. Texture-based Volume Visualization for Multiple Users on the World Wide Web. In Gervautz, M. and Hildebrand, A. and Schmalstieg, D., editor, *Virtual Environments '99*, pages 115–124. Eurographics, Springer, 1999. 1, 2

3. K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl. Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications. In *Procceedings of IEEE Visualization '00*, pages 449–452. IEEE, 2000. 1, 2

4. K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 167–177,291, May 2000. 1

5. Jim Fulton and Chris Kent Kantarjiev. An update on low bandwidth X (LBX). In *The X Resource*, number 5, pages 251–266, January 1993. 4

6. W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software, Practice and Experience*, 21(4):375–390, 1991. 1, 3

7. Mark J. Kilgard. *Programming OpenGL for the X Window System*. Addison-Wesley, 1996. 2

8. Mark J. Kilgard, David Blythe, and Deanna Hohn. System support for OpenGL direct rendering. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 116–127. Canadian Human-Computer Communications Society, 1995. 2

9. H. Lu. ELF: From the programmer's perspective, 1995. ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz. 3

10. Kwan-Liu Ma and David M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Supercomputing*, 2000. 1, 2

11. M. Magallón, M. Hopf, and T. Ertl. Parallel Volume Rendering using PC Graphics Hardware. In *Pacific Conference on Computer Graphics and Applications*, pages 384–389, 2001. 1

12. Adrian Nye, editor. *Volume 0: X Protocol Reference Manual*. X Window System Series. O'Reilly & Associates, 4th edition, January 1995. 2

13. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–

118,147. Addison-Wesley Publishing Company, Inc., 2000. 5

14. Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998. 4

15. S. Röttger, W. Heidrich, Ph. Slusallek, and H.-P. Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. In *Procceedings of WSCG '98*, pages 315–322, 1998. 6

16. Silicon Graphics, Inc. Vizserver, November 2001. http://www.sgi.com/software/vizserver/. 2

17. Paula Womack and Jon Leech. OpenGL graphics with the X Window System, version 1.3, 1998. http://www.opengl.org/. 2

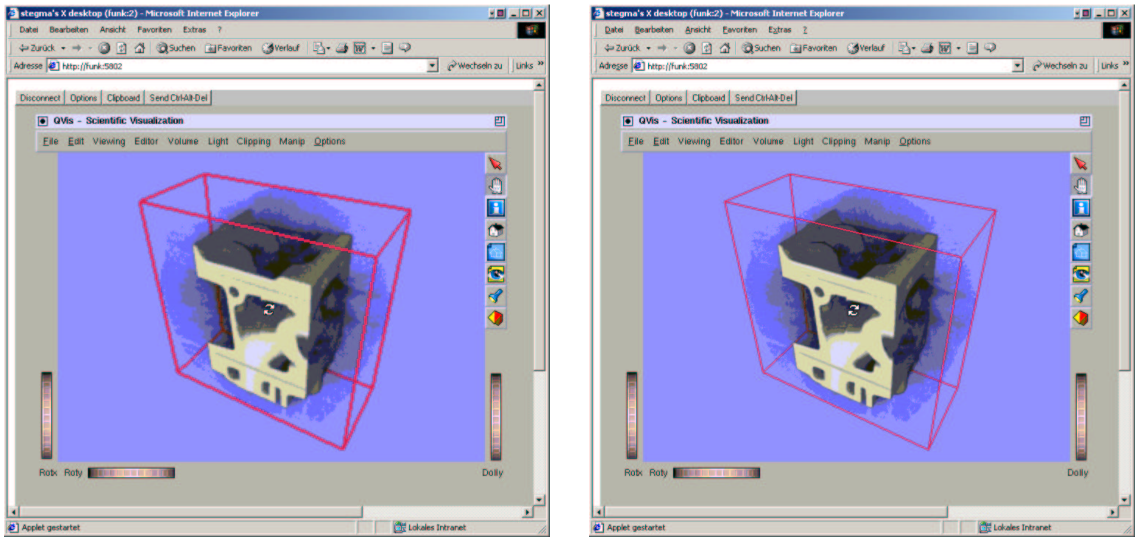

Figure 4: *Remote visualization using VNC for image compression and a web browser as VNC client. The effects of down-sampling can be seen in the left image.*

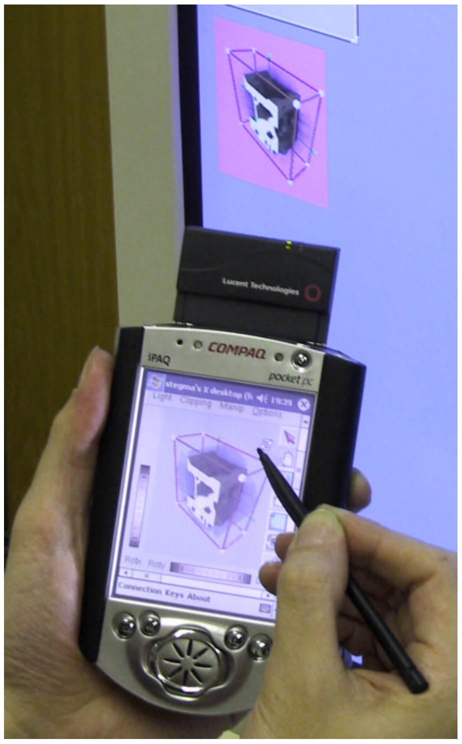

Figure 7: *Interaction with a volume dataset on a pocket PC connected to the render server via 11 Mbps wireless LAN. As can be seen in the background, the render server displays only the GLX drawable and not any of the application's widgets.*

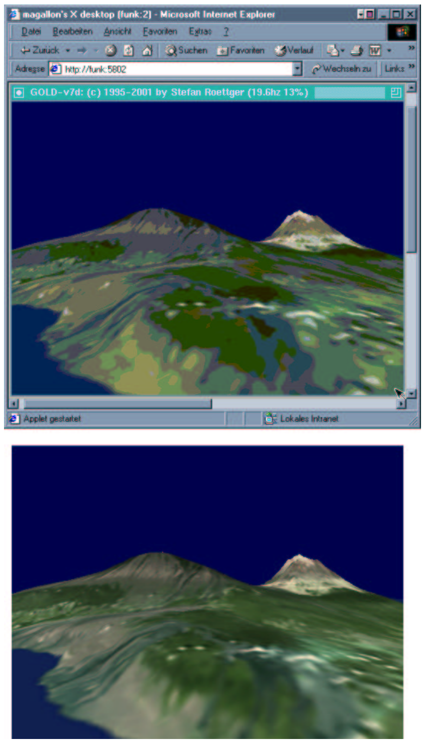

Figure 8: *Remote terrain visualization. The effects of color quantization can be seen in the upper image.*