Widening the Remote Visualization Bottleneck

Simon Stegmaier Joachim Diepstraten Manfred Weiler Thomas Ertl
Visualization and Interactive Systems Group, University of Stuttgart
Universitätsstraße 38, D-70569 Stuttgart[*]

Abstract

With the advent of generic remote visualization solutions, accessing distant graphics resources from the desktop is no longer restricted to specially tailored applications. However, current solutions still suffer from poor interactivity due to limited network bandwidth and high latency. In this paper we present several improvements to our remote visualization system in order to overcome these drawbacks. Different image compression schemes are evaluated with regard to applicability for scientific visualization, showing that image compression alone cannot guarantee interactive frame rates. Therefore, we furthermore present different techniques for alleviating the bandwidth limit, including quality reduction during user interaction or animations. Latency aspects are addressed by employing multi-threading for asynchronous compression and utilizing features of modern programmable graphics adapters for performing the image compression entirely on the graphics card. The presented ideas are integrated in our remote visualization system without violating universality.

1. Introduction

Many scientific simulations calculated these days produce datasets of sizes that cannot be handled with PCs or workstations typically available at the desks of the analyzing researchers. However, speaking in Richard Hamming's words, the purpose of computing is insight, not numbers; thus, if analyzing the data becomes increasingly difficult, the value of the simulation decreases.

A highly effective way to analyze datasets is scientific visualization. However, scientific visualization not only requires a fast CPU and huge amounts of memory (both main memory and secondary storage) but also high rendering performance provided by modern graphics adapters. Obviously, these resources are not always available and it is impossible—especially for publicly-funded research institutions like universities—to replace the computing equipment each time the dataset sizes double. And even if this was possible a central solution might still be favorable.

First, if the datasets remain on the host where they were generated, there is no need for transferring the data each time the simulation parameters have changed. This may be a great advantage, given that many wide-area network connections are still very slow compared to LANs. Second, replicating the datasets on the hosts of every researcher working with the data multiplies the overall memory and disk requirements and unnecessarily raises the costs for desktop computers. Third, raw simulation data is often confidential and a transfer via insecure network connections may be unacceptable.

To overcome these drawbacks of decentralized data storage, the concept of remote visualization was introduced. The basic idea is simple: The datasets are visualized on the high-performance computer–the server–and only the rendered images are transmitted to the researcher. This minimizes the resources required at the client. However, albeit compression techniques can be employed to reduce the size of the images to be transmitted, the data rates necessary for interactive frame rates usually restrict the applicability of remote visualization solutions to high-speed networks, usually LANs.

In this paper, several methods for achieving interactive frame rates even in low-bandwidth environments are discussed and evaluated using our own generic remote visualization solution as a testing platform. The paper is organized as follows: Related work is discussed in Section 2. Our remote visualization solution is described in Section 3, including a revised architecture required for incorporating the optimizations. Software-based optimization methods are discussed in Section 4, followed by the presentation of a hardware-based image compression in Section 5. Recommendations and a conclusion are given in Section 6.

2. Related Work

Engel et al. [3] developed a framework for providing remote control to OpenInventor and Cosmo3D applications. The system transmits compressed images from the server to a Java-based client and returns events generated at client side via CORBA requests. The system requires substantial modifications of the visualization application and shows a very low overall performance.

Ma and Camp [8] developed a solution for remote visualization of time-varying data over wide area networks. It involves a dedicated display daemon and a custom transport method which enables them to employ arbitrary compres-

[*] {stegmaier|diepstraten|weiler|ertl}@informatik.uni-stuttgart.de

sion techniques. Again, the presented system is not generic but targeted at a parallel volume rendering application.

Bethel [1] presented Visapult, a system that combines minimized data transfers and workstation-accelerated rendering. Like the solutions developed by Engel, Ma, and Camp, Visapult requires modifications of the application in order to make it “network aware”. As another drawback, the system relies to some extent on the existence of hardware graphics acceleration on the local display.

A truly generic solution was first described in detail by Stegmaier et al. [15]. The system takes advantage of the transport mechanisms part of the X Window System [10] leading to an extremely compact implementation and (in combination with VNC [12]) support for a very large variety of platforms for the client. Due to the abandonment of a proprietary data channel, no custom compression of images is supported.

Wide-spread use of generic solutions came with the advent of SGI’s commercial OpenGL Vizserver [14]. OpenGL Vizserver allows a wide range of clients to access the rendering capabilities of remote graphics servers. However, due to certain design decisions, the servers are restricted to SGI workstations. Amongst other compression algorithms, Vizserver includes an implementation of *Color Cell Compression* [2].

3. Status Quo

The remote visualization system presented in this work is based on the system described in [15]. The core idea can be easily recognized by considering the way an OpenGL-based application generates images when the display is redirected to a remote host[1] (left side of Fig. 1). All OpenGL commands are encoded using the GLX protocol and transmitted to the remote host as part of the well-known X11 protocol stream. The stream is then decoded by the receiving X server and the OpenGL commands are executed exploiting the resources available at the destination host. This is exactly what is *not* desired for hardware-accelerated remote visualization since we cannot benefit from the graphics resources of the host running the application. Thus, the transmission of OpenGL commands must be suppressed (to obtain the scenario on the right side of Fig. 1) and only the rendered images should be transferred to the host where the user interaction occurs. In the remainder of this paper, we will use the terms *render server* and *interaction server* to refer to the two hosts involved in the visualization process.

Surprisingly, the required modifications of the GLX architecture can be obtained through a single shared object linked to the visualization application at program start-up.

3.1 Dynamic Linking

Almost all applications developed these days make use of functionality provided as libraries. Usually, library functionality is provided in two formats: as *static* libraries and as *shared* libraries. It is up to the programmer to select a suitable format.

[1]We will only address remote visualization systems based on the X Window system in this paper.

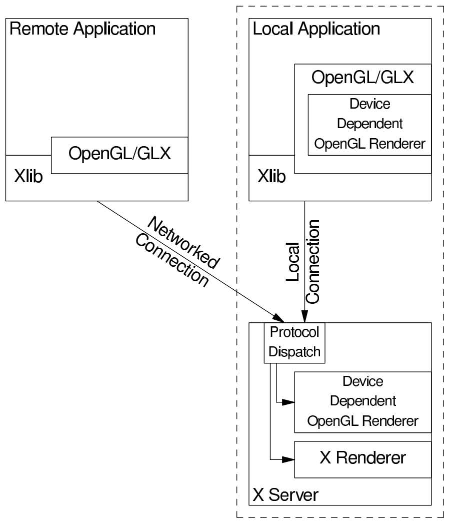

Figure 1. GLX architecture as of Kilgard [6].

For shared libraries only a link to the actual functionality is included in the application’s executable. Thus, *dynamically linked* applications allow the replacement of library functionality without relinking. Therefore, shared objects are the only suitable choice for library functionality that is not portable between different target platforms, e.g. the GPU-dependent OpenGL libraries.

Most systems that support load-time linking also support a second mechanism sometimes referred to as *preloading*. Preloading allows to manipulate the stack that is used to resolve function references in dynamically linked applications, usually by setting an environment variable to an object file that redefines the functions to be overridden. Therefore, preloading presents an easy method for non-invasively customizing library behavior. However, completely redefining library functions is not trivial—in particular if the functions perform complex tasks.

To cope with this problem, many systems also support *run-time* linking. With run-time linking, symbol resolution is performed after the program has started instead of during the loading phase. The original functionality can then be recovered by opening the original shared object (usually with `dlopen`) and determining the address of the relevant function from its name (with `dlsym`).

3.2 Library Architecture

Using the described dynamic linking functionality, a generic remote visualization solution can be obtained by preloading library functions from two groups.

- Set-up functions like `XOpenDisplay`, `glXChooseVisual`, and `glXMakeCurrent`
- Trigger functions like `glXSwapBuffers`, `glFlush`, and `glFinish`

The set-up functions make sure that the stream of OpenGL commands is directed to the render server, the trigger functions determine the point of time that the rendering has finished and an image update must be sent to the interaction server. If an image needs to be sent, the image is read from the framebuffer with `glReadPixels` and transmitted via `XPutImage` with the display set to the interaction server. The technical details can be found in [15].

3.3 Evaluation of the Original Library

The described library architecture is simple, can be implemented using a single software component, and reduces the coding efforts to a minimum since the handling of the user interface is done completely by the X Window system. In addition, since the library is able to work with any kind of X server, combinations with LBX [4] and VNC [12] are possible.

However, the approach also has disadvantages. The main drawback is that the transmission of the image data cannot be controlled without modifying the X server, i.e., no user-specified compression schemes can be applied to the image data besides those already incorporated in VNC or LBX; thus, the solution may be suboptimal both in environments of very low network bandwidth (the compression ratio may to too low to allow for interactive work) and very high network bandwidth (where the compression times may be the limiting factor).

3.4 Revised Architecture

A remote visualization architecture suitable for all kinds of network environments must be able to use arbitrary compression algorithms. This is most easily accomplished by setting up a dedicated data channel (e.g. a TCP connection) for the image transfer and by providing a client application capable of decompressing the received data (Fig. 2). Compared to the original architecture, the required modifications are only minor—basically, the only difference is that now `XPutImage` is called by the client application instead of the remote visualization library. However, the original architecture's one-component approach and ease of use are major benefits from a user's point of view; thus, care should be taken not to sacrifice these benefits for improved performance. Considering an invocation of an OpenGL application with the original implementation:

```
$ export DISPLAY=salsa:0.0
$ LD_PRELOAD=$PWD/librv.so.0 opengl_app
```

it becomes apparent that all the information required for addressing the image data can be determined from the value of the `DISPLAY` environment variable. Therefore, no arguments need to be specified by the user when starting the application on the render server.

Once the image data has been received by the client program, the data must be decompressed and drawn into the appropriate window. Since this is already accomplished in the original implementation, the window identifier is known and can be sent to the client as header information. As a result, the client, too, can be started without passing any arguments. Thus, the two-component revised architecture is as user friendly as the original implementation but provides a basis flexible enough for evaluating the optimizations proposed in this paper.

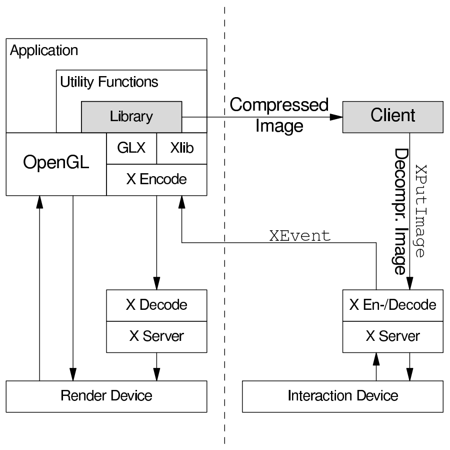

Figure 2. Revised two-component architecture of generic remote visualization system with custom image compression.

4. Widening the Bottleneck

From the results given in [15] it is noticeable that already at screen sizes beyond 512 $\times$ 512 the frame rates drop rapidly. The situation gets even worse when dealing with slow networks like Wireless LAN 802.11b (11 Mbps) or DSL (2 Mbps). There is only one option to solve this problem: Reducing the amount of data which has to be transmitted over the network.

However, the quality of the network connection is not the only parameter having influence on the achievable frame rates. Another issue is latency, the time that passes between the start of the rendering and the sending of the final image. This server-side latency is even increased when compression algorithms are applied. Therefore, methods for handling both latency and data reduction are necessary to achieve increased frame rates on the client.

4.1 Data Rate Reduction

A basic idea for reducing the data rate without much effort is to simply reduce the amount of data *created* on the server. For some applications, having a true-color visual is not necessary—16 bit or even 8 bit color depths might be sufficient. However, this is not always desired (e.g. in medical applications) and only helps to reduce the data amount by a factor of two to three which is not enough in most cases. Therefore, avoiding redundant information (from a coding theory's point of view) is more appropriate. Exploiting redundancies for creating a more compact version

Table 1. Frame rates for different scenarios. Top: 802.11b WLAN, bottom: Fast Ethernet.

Compression	WLAN Ratio	WLAN PSNR	WLAN FPS	Ethernet Ratio	Ethernet PSNR	Ethernet FPS
none	–	–	0.6	–	–	10.2
LZO	17	–	9.0	33	–	170.0
ZLIB	40	–	9.4	79	–	72.0
CCC	8	37	7.5	8	42	138.0
CCC_LZO	52	37	10.0	89	42	138.4
BTPC	32	17	3.5	61	18	20.0

of the original data is the approach of data compression algorithms.

4.1.1 Software Compression

Many general compression algorithms and also many special image compression algorithms can be found in the literature [16]. However, an algorithm suitable in a remote visualization scenario has to fulfill certain criteria: 1. High interactivity, meaning both short compression and decompression times for not increasing latency more than necessary; 2. Good image quality, especially in the case of medical (volume) visualization. Compression artifacts could otherwise lead to misinterpretation of data with crucial consequences; 3. High compression rates for fast image transmission even over slow network connections.

Since remote visualization systems basically have to deal with synthetic images, it has to be taken into account that some compression algorithms (e.g. JPEG) do not behave too well for these kind of images.

Our revised remote visualization system includes five software compression algorithms based on three different groups of compression techniques: ZLIB [5] and LZO [11] for lossless compression, *Color Cell Compression* (CCC) [2] exploiting spatial redundancy, a combination of CCC with LZO (CCC_LZO), and *Binary Tree Prediction Coding* (BTPC) [13]. These five different compression methods are evaluated by examining the compression ratio, the achievable frame rates with different network and client configurations, and the image quality according to the peak signal to noise ratio (PSNR).

Table 1 shows the results measured for two different client/server and network configurations. The slow network is a 11 Mbps 802.11b WLAN, the fast network is a 100 Mbps Fast Ethernet. For both network configurations, a PC with an Intel Pentium III 1 GHz, 256 MB Memory, and an NVIDIA Geforce2 was used as server. The client for the slow network was a Compaq iPAQ 3850 with a 211 MHz StrongArm CPU. For the fast network, a PC with an AMD Athlon 1.5 GHz processor was used. One of the applications from [15] was used for the measurements: "gears". For the WLAN, the window was set to 240×320 pixels (the display size of the iPAQ) at 16 bit color depth, and to 512×512 pixels at 24 bit color depth for the PC client.

The WLAN frame rates are lower than expected requiring further explanation: First, the maximum measured throughput was only 197 KBytes/s even with best possible reception. Second, the iPAQ only supports 16 bit colors whereas the special image compression algorithms assume and encode only 24 bit color images. This means that the same amount of data (24 bit) is passed to the compression algorithms and that we, therefore, cannot profit from a reduced color depth regarding compression/decompresssion and transmission times.

4.1.2 Motion Handling

To reduce the image data even further, it is possible to exploit the human eye's insensitivity to details of moving objects by adjusting the quality parameter of the compression algorithms or by downsampling the image to a quarter of its original resolution during user interaction or animations. The server can employ a heuristic to automatically detect these situations by considering the average number of buffer swaps per time unit. The image is scaled back to its original resolution by the client. Of course, downsampling leads to a definite loss in image quality. Therefore, when the user stops to interact with the application or the animations stops, a high quality version of the last generated image is sent to the client. This approach reduces both the rasterization load of the server and the raw image data to a quarter. Applications that are rasterization bound will, therefore, profit from this solution in double respect. To test the efficency of the downsampling, another experiment was carried out using the Fast Ethernet scenario. It was possible to increase the frame rates with LZO compression from 170 fps to over 260 fps. The frame rates could not be increased as expected (about a factor of 4) since the client has both to rescale and to mirror the image in software. This takes about 3.5 ms in total for a 512×512 image, yielding a maximum frame rate of about 286 fps.

4.2 Latency Reduction

Remote visualization using the described library involves phases of high CPU utilization as well as phases of high GPU utilization. Four major phases can be identified: image rendering, framebuffer read-out, image compression, and data transmission. Provided that the rendering phase makes extensive use of advanced OpenGL features like display lists and assuming that the framebuffer read-out can be done with a DMA operation, these two phases can be classified as *GPU-dominated* work. The latter two *CPU-dominated* phases (compression and transmission) are using the CPU exclusively, potentially consuming resources required by the graphics card driver and consequently slowing down the rendering. This can be alleviated given a suitable parallel architecture as depicted in Fig. 3. The solution is based on POSIX pthreads and uses two semaphores for synchronizing the exchange of image data.

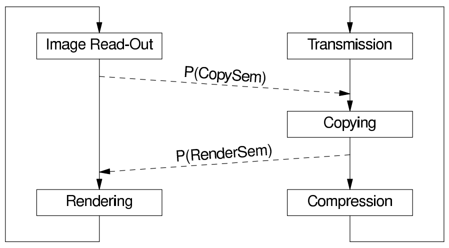

Figure 3. Parallel processing in the remote visualization library.

Using a dual-processor AMD Athlon MP 2200+ PC as render server, a frame rate increase of about 42% (195 vs. 137 fps) could be experienced. However, parallel processing is just one means for improving latency. An alternative is to reduce the compression time.

5 Hardware Compression

The graphics adapters of many modern PCs include a powerful programmable processing unit. This GPU is capable of even outperforming modern CPUs for certain highly parallelizable tasks. Some image compression algorithms, e.g. the *Color Cell Compression* proposed by Campbell et al. [2], match this criterion, which suggests a CCC implementation on graphics hardware. Besides an expected reduction of the compression time, a hardware-based compression has the advantage that only a fraction of the original data has to be read from the framebuffer, making expensive framebuffer reads much more bearable.

In a hardware-based decompression algorithm the GPU can also benefit from the reduced image data by utilizing a texture map with the encoded image and reconstruct the image on the fly without performance penalties compared to rendering a textured quadrilateral. If OpenGL is available on the interaction server this can be a significant performance gain compared to writing the decoded image to the framebuffer with `XPutImage`.

Understanding our hardware compression requires knowledge of the CCC algorithm. The CCC algorithm first decomposes the source image given in a 3 bytes/pixel RGB format into independently processed cells of 4 × 4 pixels. For each cell, a mean luminance $\mathcal{L}_{mean}$ is calculated using the well-known luminance equation $\mathcal{L} = 0.3R + 0.59G + 0.11B$. Next, all pixels of the cell are divided into pixels with luminance less than or equal to $\mathcal{L}_{mean}$ and pixels with luminance greater than $\mathcal{L}_{mean}$. A 16 bit bitmask is then created in which pixels with luminance values greater $\mathcal{L}_{mean}$ are marked with 1. Additionally, two RGB color values are stored per cell which are the arithmetic mean RGB values of the pixel groups. The image is reconstructed from the bitmask and the two colors by setting the first color where the corresponding cell's bitmask is 0, and the second color where the bit is 1. Using this approach, a cell can be encoded with $2+3+3 = 8$ bytes. Additionally a quantization of the group colors is applied storing only the 5, 6, and 5 most significant bits of the red, green, and blue channel respectively, yielding an overall compression ratio of $1 : 8$.

Obviously, the CCC algorithm allows all cells to be processed independently. This matches the architecture of modern graphics adapters with several pixel pipelines operating in parallel very well. The following description relates to an implementation on an NVIDIA GeForceFX graphics card that—with slight modifications—has also been successfully adapted to the ATI Radeon 9700.

5.1 Encoder

The hardware-based CCC algorithm renders a quadrilateral with a size of one fourth of the source image in each dimension. Thus, each pixel generated from this quadrilateral maps to one cell of the source image. The color data of the source image is provided by a texture map that is updated in each frame. Since OpenGL allows to efficiently copy data directly from the framebuffer to a texture map, this step hardly involves any overhead compared to the expensive framebuffer read of a software solution. The alternative of using NVIDIA's `render_texture_rectangle` OpenGL extensions [7] that allows to directly define a texture from the framebuffer without copying was discarded due to a negligible performance benefit.

To map rendered pixels to cells, a fragment shader is applied that performs 16 lookups in the source image texture for acquiring the cell's color information and for generating the bitmask and the two group colors. The shader implementation is almost a 1:1 mapping of the software encoder since NVIDIA's high level shading language *Cg* [9] was utilized for the hardware implementation.

The bitmask and the two group colors ideally should be written with 16 bit precision to the red, green, and blue component of the framebuffer. Unfortunately, only 8 bit framebuffer precision is currently available. A possible solution would have been to use an additional output target and to write the bitmask to the red and green components of the target and the colors to the red/green and blue/alpha channels of the framebuffer, respectively. However, currently no multiple render targets are available for the NVIDIA card. Another alternative, 16 bit pbuffers, had to be discarded due to huge performance penalties caused by context switches.

For these reasons, three rendering passes are applied. The first pass computes the bitmask, whereas the second and the third passes compute the two colors. Each value is computed with 16 bit, split into a high and a low byte and written to the red and alpha component of the fragment. After each pass, the pixels covered by the rendered quadrilateral are read from the framebuffer. Actually, a two-pass solution is also sufficient: one pass for the bitmask and one pass for both colors. However, this requires a fragment shader for the decoding to separate the colors on the graphics card. On an AMD Athlon XP 2200+ PC the three-pass encoder took 20.6 ms for compressing a 512 × 512 image. Compressing the same image with two passes only required 14.2 ms.

5.2 Decoder

Whereas the encoder requires the extended fragment processing features of the latest graphics hardware generation, a hardware-based decoding can be achieved basically with OpenGL 1.0 functionality. The reconstruction requires three passes. First, the bitmask data is written into the stencil buffer. In the second pass, a screen-sized quadrilateral is rendered with the *second* color mapped as a texture. The OpenGL stencil-test is exploited to write only pixels with their corresponding bit in the bitmask set. In the third pass using the *first* color as a texture only pixels with a corresponding unset bit are written by inverting the stencil-test. This solution took 5.4 ms for decoding a 512 $\times$ 512 image on the test system. The reconstruction can be reduced to a single rendering pass if the interaction server offers multi-textures and extended fragment color combination (as provided since NVIDIA's GeForce256 chip via the `register_combiners` extension [7]). In this case, the bitmask and the two group colors are bound as three textures and the combiner setup routes either the first or the second color to the fragment color depending on the value of the bitmask texture. This decoding required 5.3 ms which implies that the texture setup is more expensive than the rendering itself.

The color textures can directly be defined from the CCC image stream, provided that the encoded data for the bitmask and the colors are stored consecutively and not interleaved as in the original CCC description. OpenGL intrinsically supports the 5/6/5 quantized color format. All that is necessary is to provide OpenGL with the right offset in the data block. Unfortunately, the bitmask requires more effort. OpenGL does not allow to specify a texture from bit data; therefore, the bitmask first has to be converted to a byte stream with one byte per bitmask bit. Using a fragment shader for the reconstruction, the bitmask texture can also be defined directly from the image stream since the individual bits per pixel can be extracted from the fragment shader on the fly. Thus, as programmable fragment processing becomes a common feature for each desktop PC in the near future, the need for unpacking the bitmask data is eliminated. The fragment shader implementation required 6.8 ms when using three individual textures for the bitmask, the first and the second color. This is slightly slower than using an already unpacked bitmask. However, on systems with slower processors, a significant speed-up from 20.3 ms to 8.9 ms could be measured.

6. Results and Conclusion

Our experiences suggest the use of lossless image compression algorithms for both scenarios (WLAN/PDA and Fast Ethernet/PC). In addition, parallel processing can increase the frame rates significantly if the render server provides multiple CPUs. On the contrary, hardware compression currently has no considerable impact since carefully hand-optimized software implementations achieve equal or even higher performance (our reference CCC implementation took 12.5 ms and 5.4 ms for encoding and decoding, respectively). However, the pure processing power of modern GPUs already beats general CPUs and projecting recent developments of graphics adapters we expect the gap to even widen. Extended features will remove the necessity for work-arounds, further accelerating hardware compression. For motion handling, we see little potential unless networks with bandwidths much below WLAN are used.

Acknowledgments

We would like to thank our student Gueorgui Ovtcharov for integrating some of our ideas into the system.

References

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

[2] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, and L. A. Leske. Two bit/pixel full color encoding. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 215–223. ACM Press, 1986.

[3] 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.

[4] J. Fulton and C. K. Kantarjiev. An update on low bandwidth X (LBX). In *The X Resource*, number 5, pages 251–266, January 1993.

[5] J.-L. Gailly and M. Adler, 2002. http://www.gzip.org/zlib.

[6] M. J. Kilgard. *Programming OpenGL for the X Window System*. Addison-Wesley, 1996.

[7] M. J. Kilgard, editor. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001.

[8] K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Supercomputing*, 2000.

[9] NVIDIA Corp. Cg Language Specification, 2002. Available at http://developer.nvidia.com/cg.

[10] A. Nye, editor. *Volume 0: X Protocol Reference Manual*. X Window System Series. O'Reilly & Associates, 4th edition, January 1995.

[11] M. Oberhumer, 2002. http://www.oberhumer.com/opensource/lzo.

[12] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[13] J. A. Robinson. Efficient General-Purpose Image Compression with Binary Tree Predictive Coding. In *Procceedings of IEEE Transactions on Image Processing '97*, pages 601–607. IEEE, 1997.

[14] Silicon Graphics, Inc. OpenGL Vizserver 3.0 – Application-Transparent Remote Interactive Visualization and Collaboration, 2003. http://www.sgi.com/.

[15] S. Stegmaier, M. Magallón, and T. Ertl. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '02*, 2002.

[16] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 1999.