A Study of the Performance of VirtualGL 2.1 and TurboVNC 0.4

Version 1d, 8/20/2008 -- The VirtualGL Project

This report and all associated illustrations are licensed under the Creative Commons Attribution 3.0 License. Any works which contain material derived from this document must cite The VirtualGL Project as the source of the material and list the current URL for the VirtualGL web site.

This report attempts to characterize the performance of VirtualGL 2.1 and TurboVNC 0.4 across the range of supported server and client platforms, across different types of networks, and using a wide variety of configuration options. It also attempts to compare, apples to apples, the performance of the current versions of VirtualGL and TurboVNC with their predecessor versions and with current versions of other remote display software.

1 The Tools

1.1 GLXSpheres

The GLXSpheres benchmark, which is included in the VirtualGL 2.1 server packages and which is described in detail in the VirtualGL User's Guide, is designed to emulate the image output of the old nVidia SphereMark demo. It was discovered, quite by accident, that the SphereMark demo is a good testbed for studying the performance of VirtualGL's image pipeline. The images generated by SphereMark, and by its open source look-alike GLXSpheres, contain a realistic proportion of solid background and smooth-shaded geometry which simulates the workload of images generated by real visualization and CAD applications. The benchmark also contains very few polygons, so when it is run in a VirtualGL environment, the performance will always be limited by VirtualGL and never by the server's 3D graphics card.

Whether or not GLXSpheres is reflective of the performance of any real application is left as an exercise for the reader. To put this another way, Your Mileage May Vary (YMMV.)

1.2 CPUStat

CPUStat is a simple Linux tool which reads the `/proc` filesystem to determine the percentage of time for which the server CPUs are active. `vmstat` on Solaris serves the same purpose.

1.3 NetTest

NetTest, which is included in the VirtualGL server and client packages, was extended for the purposes of this study to include a bandwidth measurement tool. This new mode, enabled by invoking `nettest -bench`, measures the aggregate bandwidth usage on a given network interface over a given period of time. Adding this mode to NetTest was necessitated by accuracy issues which were discovered in other open source bandwidth measurement solutions, such as IPTraf and NICStat.

1.4 NISTNet

Developed by the National Institute of Standards and Technology, NISTNet is a network performance limiter. It can be used to artificially add latency to or subtract bandwidth from a Linux network connection, simulating a slower network.

2 The Methodology

Benchmarking VirtualGL, or any other complex system, requires a high degree of vigilance. There is always a risk that a transient phenomenon may introduce unwanted variables which affect the consistency or reproducibility of the tests. The general strategies for benchmarking this system are: (a) run an application which is known to produce steady performance in a local display environment, (b) measure this performance over a relatively long time period, and (c) where possible, sanity check results against other, independent metrics.

For each client, server, and X server combination, a baseline level of performance was obtained, and one parameter at a time was altered and compared to the baseline in order to ascertain that parameter's effect on the performance of the system. The baseline was obtained by running the 64-bit version of GLXSpheres in VirtualGL with no frame spoiling:

```
/opt/VirtualGL/bin/vglrun -sp /opt/VirtualGL/bin/glxspheres64
```

If the VGL Image Transport was used, then the baseline was measured with the default `vglconnect` options (X11 tunneled through SSh but with no encryption or tunneling of the VirtualGL image stream.)

For each test, GLXSpheres was run with its default window size of 1240 x 900 pixels. Care was taken to ensure that the benchmark window was never obscured during any of the tests. The client screen was set to a resolution of 1280 x 1024 pixels to accommodate the benchmark window without clipping. When running GLXSpheres in an X proxy environment, such as TurboVNC, the proxy desktop resolution was set to 1240 x 900 pixels, and GLXSpheres was run using full-screen mode:

```
/opt/VirtualGL/bin/vglrun -sp /opt/VirtualGL/bin/glxspheres64 -fs
```

This caused GLXSpheres to occupy the entire client area of the X proxy window, which resulted in the same coverage of the client desktop as if the benchmark had been run using the VGL Image Transport.

The benchmark's frame rate was measured from the client's point of view by averaging the output of TCBench over a 60-second period (two separate 30-second runs.) This was sanity checked, where possible, with the profiling output of VirtualGL as well as the frame rate reported by GLXSpheres itself. As long as the VGL Image Transport is being used with frame spoiling disabled, then the benchmark's reported frame rate is a valid measure of client/server frame rate The frame rate reported by GLXSpheres is additionally valid when using TurboVNC 0.4, as long as frame spoiling is disabled in VirtualGL and as long as the network and the TurboVNC viewer are able to process frames as fast as the server can compress and send them. In cases where the frame rate was somewhat high (generally 30+ frames/second), it was necessary to increase the sampling rate of TCBench (`tcbench -s100`) to obtain accurate results.

While the benchmark was running, the average CPU usage across all server CPUs was measured by running `/opt/VirtualGL/bin/cpustat` (Linux) or `vmstat 5` (Solaris) on the server. These programs were allowed to run until the average CPU usage converged to a consistent value.

While the benchmark remained running, the average bandwidth usage for the server's network device was measured by running `/opt/VirtualGL/bin/nettest -bench <device> 25` on the server, and the average of the second and third results were taken (this equates to the average network bandwidth for a 50-second period.)

3 The Metrics

While frame rate is a useful metric, it does not tell the whole story. The reason is that the frame rate in any thin client system is often limited by the client's (in)ability to decompress and draw the frames. If two tests produced the same frame rate, it would be impossible to tell whether one test used more CPU or network resources than the other. As long as the resource consumption was within the limits of what the server and the network could handle, then the difference in resource consumption would not reveal itself in a simple comparison of client/server frame rates.

One must instead look at the average CPU and network usage on the server to ascertain which modes of operation made more efficient use of these resources. Why this is important is that VirtualGL servers are usually provisioned for more than one simultaneous user. It is important to understand how many VirtualGL users can potentially co-exist on a given server platform and network connection without bogging down either. In this study, four metrics were used to ascertain this: “CPU-limited frame rate”, “encoding time”, “network-limited frame rate”, and “frame size.”

The CPU-limited frame rate is, simply put, the frame rate at which VirtualGL could theoretically deliver frames if the server CPUs were the only bottleneck. It is computed by dividing the actual client/server frame rate (in frames/second) by the average server CPU utilization. For instance, if the actual frame rate was 22 frames/second and the server's CPUs were, on average, 60% busy, then the CPU-limited frame rate was 22 / 0.60 = 37 frames/second. This tells us that, if a second user were to start using VirtualGL on the same server, there is a possibility that the first user's performance would slow down a bit (assuming that both users were actively manipulating a 3D scene at the same time, and assuming that both users were connected to the server via similar clients and networks.) If the actual

frame rate equals the CPU-limited frame rate, then the server's CPUs are maxed out, and the server cannot accommodate additional users without a proportional drop in performance. The CPU-limited frame rate defines how efficiently the server CPUs were used when compressing and sending each frame. A higher value means that the CPUs took less time to compress each frame and were thus used more efficiently.

Encoding time is the reciprocal of the CPU-limited frame rate. It defines the amount of CPU time (usually expressed in milliseconds) that it took the server, on average, to encode each frame for transmission. This document usually reports absolute figures in terms of CPU-limited frame rate but will sometimes refer to their relative differences using encoding time, since encoding time is an intuitively easier concept to grasp.

The network-limited frame rate is the frame rate at which VirtualGL could theoretically deliver frames to the client if the network was the only bottleneck. Network-limited frame rate is computed by dividing the theoretical network bandwidth (in Megabits/second) by the actual network usage and then multiplying by the actual frame rate. For instance, if a test generated an average frame rate of 20 frames/second using 60 Megabits/second of bandwidth on a 100 Megabit/second network, then the network-limited frame rate was 100 / 60 * 20 = 33 frames/second. If two users had to share this 100 Megabit/second interconnect, then there is a good chance that they would both observe performance degradation when simultaneously manipulating a 3D scene in VirtualGL.

The last metric, frame size, is closely related to network-limited frame rate. Frame size is simply the number of megabits of data, on average, that are required to represent a single frame of the 3D animation on the network. It is computed by dividing the average network usage by the frame rate. For instance, (60 Megabits/second) / (22 frames/second) = 2.73 Megabits/frame. It is easy to derive the compression ratio from this figure if you know the size of the image. In the case of GLXSpheres, the image contains 1240 x 900 = 1,116,000 pixels. Each pixel initially contains 24 bits, so the compression ratio for the above example is (1.12 Megapixels/frame) * (24 bits/pixel) / (2.73 Megabits/frame) = 9.8.

Returning to the definition of network-limited frame rate, it is now easy to see that it is simply the theoretical network bandwidth (in Megabits/second) divided by the frame size (in Megabits/frame.) So, for instance, if a particular protocol had a frame size of 2 Megabits, 50 frames/second could be accommodated on a 100 Megabit/second link, assuming no other performance bottlenecks.

For the purposes of this document, 1 Megabit = 1,000,000 bits, not 1,048,576 bits. This is the common usage when referring to the speed of networks.

4 The Systems

The test systems consisted of "typical" workstation platforms that might be used with VirtualGL:

SPARC server and client:

- Dual-processor (1.6 GHz UltraSPARC III) Sun Ultra45
- 2 GB memory
- Sun XVR-2500 framebuffer
    - Driver patch 120928-22
- Sun OpenGL v1.5, patch 120812-22
- Solaris 10 6/06 + all patches up to 10/31/07
    - Sun mediaLib 2.5

x86 server:

- Sun Ultra20 with 2.4 GHz dual-core AMD Opteron 180
- 2 GB memory
- nVidia QuadroFX 1400
    - Driver version 100.14.19
- Solaris 10 Update 4
    - Sun mediaLib 2.5
- CentOS Enterprise Linux 5.0

x86 client:

- Sun w1100z with single 1.8 GHz AMD Opteron 144
- 512 MB memory
- nVidia QuadroFX 3000
    - Driver version 100.14.19 (Linux) and 81.67 (Windows)
- Windows XP Service Pack 2
    - Hummingbird Exceed 3D 2008
- CentOS Enterprise Linux 5.0

5 The Results

5.1 VirtualGL 2.1 vs. VirtualGL 2.0.1 (VGL Image Transport)

No significant differences were observed between the performance of VirtualGL 2.0.1 and VirtualGL 2.1 when running on Linux or Windows systems. However, there was marked improvement between VirtualGL 2.0.1 and VirtualGL 2.1 on Solaris/SPARC systems. This increase in performance was due to optimized Huffman coding routines, which were contributed to TurboJPEG/mediaLib by the mediaLib authors.

Comparison of VGL Image Transport Performance Between VirtualGL 2.0.1 and VirtualGL 2.1 Servers (Connecting to VirtualGL 2.1 Client)

	Actual Frame Rate (VGL 2.0.1)	Actual Frame Rate (VGL 2.1) *[Baseline]*	CPU-Limited Frame Rate (VGL 2.0.1)	CPU-Limited Frame Rate (VGL 2.1) *[Baseline]*
SPARC → Windows	18.62	23.75 **(+28%)**	29.09	38.93 **(+34%)**
SPARC → Windows (`VGL_NPROCS=2`)	23.51	23.17 **(-1.4%)**	28.66	37.37 **(+30%)**
Solaris/x86 (32-bit) → Windows	20.08	23.04 **(+15%)**	34.62	39.04 **(+13%)**
Solaris/x86 (64-bit) → Windows	23.09	23.01 **(-0.35%)**	47.12	46.96 **(-0.34%)**

Comparison of VGL Image Transport Performance Between VirtualGL 2.0.1 and VirtualGL 2.1 Clients (Connecting to VirtualGL 2.1 Server)

	Actual Frame Rate (VGL 2.0.1)	Actual Frame Rate (VGL 2.1) *[Baseline]*
Linux → **SPARC**	22.57	33.19 **(+47%)**

With VGL 2.0.1, it took two compression threads on the SPARC server to drive the client at full capacity. With VGL 2.1, this could be achieved with only one compression thread, thanks to the 34% more efficient use of the server's CPUs.

For the Solaris/x86 server, the optimized Huffman routines only improved the performance of the 32-bit VirtualGL server components. However, the Solaris/x86 server realized a huge performance improvement across the board by upgrading from mediaLib 2.4 (the version included with Solaris 10) to mediaLib 2.5 (available from http://www.sun.com/processors/vis/mlib.html). See below for more details.

5.2 TurboVNC 0.4 vs. TurboVNC 0.3.3

In order to discuss what has changed between TurboVNC 0.3.3 and 0.4, it is first necessary to discuss how VNC works. The RFB protocol, on which VNC is based, is a client-driven protocol. Most other remote display protocols (including the VGL Image Transport, X11, RDP, etc.) are server-driven protocols. With a server-driven protocol, the server pushes image updates to the remote display client, usually unprompted. With VNC, on the other hand, the server will not send pixels to a VNC viewer unless the viewer explicitly requests a framebuffer update.

After initialization, the VNC viewer sends an initial framebuffer update request to obtain the entire contents of the server's virtual framebuffer. After that, it can send incremental update requests to obtain only modified regions of the framebuffer. When a framebuffer update request is received by the VNC server, the server first checks to see if there are any modified pixels to be sent. If so, then those pixels are immediately compressed and sent to the viewer as a framebuffer update. The viewer then decompresses and draws the update before sending a request for another update. Meanwhile, the VNC server processes X11 requests from applications while it is waiting for the next update request from the viewer.

This works OK on low-latency networks, but on high-latency connections, it's problematic. Since the server is waiting on the viewer and the viewer is waiting on the server, each framebuffer update requires a round trip from server to viewer. As the latency of the network increases, it begins to take as much time to transmit each frame as it does to encode or decode it, and this causes the frame rate to drop severely. In the VGL Image Transport and other streaming image protocols, network latency is "hidden" by pipelining the various stages of image transmission. The client can be decoding and drawing one frame while it is receiving another and while the server is encoding yet another. With a client-driven protocol, however, such pipelining is impossible, because the next frame cannot be encoded by the server until the previous one has been decoded by the client.

Beginning with TurboVNC 0.3.2, an attempt was made to pipeline at least part of the RFB protocol. This was accomplished by modifying the TurboVNC viewer such that it sent a new framebuffer update request before decoding the previous update, rather than after. This allowed at least part of the network latency to be hidden, which significantly improved TurboVNC's performance on high-latency networks. However, it was discovered that this optimization slowed performance on low-latency networks. The reasons why were not well understood at the time, and in TurboVNC 0.3.3, a "High-Latency Network" switch was introduced as a compromise. This switch allowed one to turn off the pipelined update request optimization when using TurboVNC on low-latency networks.

One of the goals for TurboVNC 0.4 was to eliminate this "High-Latency Network" switch, since it was often a source of confusion for users. So we set out to uncover the reason why enabling pipelined update requests decreased performance on low-latency networks. As it turns out, the reason has to do

with a feature of the VNC server called “deferred updates.” As with many X servers, the VNC X server is single-threaded. The server is essentially an infinite loop that continuously polls for requests from either an X application or a connected VNC viewer. If the VNC server receives a drawing request from an X application, the server will check to see if there is an unprocessed framebuffer update request from a VNC viewer. If so, it will trigger a deferred framebuffer update and return to its main loop. The loop continues to check the deferred update timer, and once the timer has elapsed, then the deferred update will be immediately sent to the viewer. If an X drawing request is received and no framebuffer update request is pending, then the VNC server marks the drawing region as modified. Modified regions will then be sent immediately the next time an update request is received from the viewer. The deferred update timer is meant to act as a sort of caching mechanism, allowing many small framebuffer updates to be combined into one larger update.

Whether or not a framebuffer update is deferred or sent immediately depends on when the framebuffer update request is received from the viewer. If a new update request is received very soon after the previous update is sent, then chances are that the framebuffer will not yet have been modified relative to the previous update. Thus, the update request will not actually be processed until the next time an X11 drawing command occurs, at which point a deferred update request will be triggered. If, however, a new framebuffer update request arrives somewhat after the previous update has been sent, then chances are that the framebuffer will have been modified, and thus a new update will be sent immediately.

To make a long story short, if the pipelined update request feature (the “High-Latency Network” switch in TurboVNC 0.3.3) was turned on when connecting over a low-latency network, it caused almost all of the framebuffer update requests to be deferred in the VNC server. This is a problem, because the default value of the deferred update timer in VNC is 40 ms. Thus, an additional 40 ms delay was being introduced into every frame. TurboVNC 0.4's solution was to reduce the deferred update timer to 1 ms. This allowed pipelined update requests to be used on low-latency networks with no performance penalty. Thus, the “High-Latency Network” switch could be hard-wired to on and removed from the TurboVNC Viewer interface.

Note that turning off the deferred update mechanism altogether (setting the timer to 0) produced undesirable performance effects, such as a perceivable lag when typing text into an xterm window. It seems that the deferred update mechanism works as designed, but the default timer value was simply too high to perform well with TurboVNC's other optimizations.

Additional performance improvements were realized in the SPARC version of TurboVNC 0.4 due to the same TurboJPEG/mediaLib Huffman coding optimizations described earlier.

The following tests compared the end-to-end performance of TurboVNC 0.3.3 and TurboVNC 0.4 when using optimal settings for a low-latency network.

Performance Comparison Between TurboVNC 0.3.3 Server/Viewer ("High-Latency Network" Switch Turned Off) and TurboVNC 0.4 Server/Viewer

	Actual Frame Rate (TurboVNC 0.3.3)	**Actual Frame Rate (TurboVNC 0.4)** *[Baseline]*	**CPU-Limited Frame Rate (TurboVNC 0.3.3)**	**CPU-Limited Frame Rate (TurboVNC 0.4)** *[Baseline]*
Linux → Linux	23.90	28.44 **(+19%)**	36.59	42.45 **(+16%)**
SPARC → Windows	14.60	17.21 **(+18%)**	23.17	25.69 **(+11%)**
Solaris/x86 → Windows	19.37	22.70 **(+17%)**	30.26	34.92 **(+15%)**

No significant improvement was observed (or expected) in the Windows or Linux TurboVNC viewers. The performance improvement observed above was due to the combination of reducing the deferred update timer to 1 ms (from 40 ms) and pipelining the framebuffer update requests from the client. Additionally, the SPARC server realized some improvement from the Huffman coding optimizations.

The following test compared the TurboVNC 0.3.3 and TurboVNC 0.4 viewers, in an attempt to isolate the performance improvement due to the Huffman coding optimizations. The TurboVNC 0.4 server was used in both cases, and the "High-Latency Network" switch was turned on in the TurboVNC 0.3.3 viewer to simulate the behavior of the TurboVNC 0.4 viewer.

	Actual Frame Rate (TurboVNC 0.3.3)	**Actual Frame Rate (TurboVNC 0.4)** *[Baseline]*
Linux → **SPARC**	21.13	24.26 **(+15%)**

In general, the performance improvements in TurboVNC 0.4 were only observed on low-latency network connections. The performance improvements between TurboVNC 0.3.3 and TurboVNC 0.4 were barely measurable on high-latency connections.

5.3 mediaLib 2.4 vs. 2.5

When the Solaris/x86 server was upgraded from mediaLib 2.4 (the version included with Solaris 10) to mediaLib 2.5, the performance of VirtualGL and TurboVNC improved dramatically.

Comparison of VGL Image Transport Performance When Using mediaLib 2.4 and mediaLib 2.5

	Actual Frame Rate (mediaLib 2.4)	Actual Frame Rate (mediaLib 2.5) *[Baseline]*	CPU-Limited Frame Rate (mediaLib 2.4)	CPU-Limited Frame Rate (mediaLib 2.5) *[Baseline]*
Solaris/x86 (32-bit) → Windows	11.66	23.04 **(+98%)**	21.20	39.04 **(+84%)**
Solaris/x86 (64-bit) → Windows	16.95	23.01 **(+36%)**	29.73	46.96 **(+58%)**

Comparison of TurboVNC Performance When Using mediaLib 2.4 and mediaLib 2.5

	Actual Frame Rate (mediaLib 2.4)	Actual Frame Rate (mediaLib 2.5) *[Baseline]*	CPU-Limited Frame Rate (mediaLib 2.4)	CPU-Limited Frame Rate (mediaLib 2.5) *[Baseline]*
Solaris/x86 → Windows	13.46	22.70 **(+69%)**	22.81	34.92 **(+53%)**

The vast improvement in 32-bit performance went a long way toward closing the severe gap that existed between the performance of 32-bit and 64-bit Solaris/x86 versions of mediaLib 2.4

There was no significant difference observed between the performance of mediaLib 2.4 and mediaLib 2.5 on SPARC systems.

5.4 32-bit vs. 64-bit

The only platform on which 32-bit code performed significantly differently from 64-bit code was Solaris/86. While the performance of 32-bit code improved vastly in the Solaris/x86 version of mediaLib 2.5 (as shown above), the 64-bit version was still found to be 20% more efficient in its use of the server's CPUs (a 20% higher CPU-limited frame rate) than the 32-bit version. Both 32-bit and 64-bit versions, however, were able to drive all of the client platforms to full capacity, so the efficiency differences would only be observable in a multi-user environment.

5.5 Will That Be One CPU or Two?

The question of whether or not a second CPU benefits VirtualGL is complex. Certainly, multiple CPUs are tremendously beneficial when multiple users are sharing the VirtualGL server. But whether a multiple-CPU system will improve a single user's performance depends on the speed of the client and the efficiency of the server. The following table demonstrates the general effectiveness of adding a second CPU on the test servers:

Improvement in VGL Image Transport Performance Due to Adding a Second CPU

	Actual Frame Rate (1 CPU)	Actual Frame Rate (2 CPUs) *[Baseline]*	CPU-Limited Frame Rate (1 CPU)	CPU-Limited Frame Rate (2 CPUs) *[Baseline]*
Linux → Windows	22.76	24.01 **(+5.5%)**	29.91	54.82 **(+83%)**
SPARC → Windows	19.96	23.75 **(+19%)**	19.96	38.93 **(+95%)**
Solaris/x86 (32-bit) → Windows	20.30	23.04 **(+13%)**	20.30	39.04 **(+92%)**
Solaris/x86 (64-bit) → Windows	23.01	23.01 **(+0.0%)**	23.72	46.96 **(+98%)**

As you can see, in a single-user environment, the second CPU is only beneficial if the server is inefficient enough to require more than one CPU to drive the client. Otherwise, the frame rate is limited by the client, and the second CPU has little benefit to the single user. These figures also demonstrate that allocating one CPU per active user is a good rule of thumb, since a single client can easily use all or nearly all of the resources of a single server CPU whenever the client is running at full capacity.

Improvement in VGL Image Transport Performance Due to Adding a Second Compression Thread

	Actual Frame Rate (`VGL_NPROCS=1`) *[Baseline]*	Actual Frame Rate (`VGL_NPROCS=2`)	CPU-Limited Frame Rate (`VGL_NPROCS=1`) *[Baseline]*	CPU-Limited Frame Rate (`VGL_NPROCS=2`)
Linux → Windows	24.01	23.04 **(-4.0%)**	54.82	45.70 **(-17%)**
SPARC → Windows	23.75	23.17 **(-2.4%)**	38.93	37.37 **(-4.0%)**
Solaris/x86 → Windows	23.01	22.60 **(-1.8%)**	46.96	46.12 **(-1.8%)**

Adding a second compression thread might be beneficial if the client was faster or if each of the individual server CPUs was slower. In Section 5.1, adding a second compression thread was also shown to be of some benefit in cases where the server's CPUs were being used inefficiently (such as in the SPARC version of VirtualGL 2.0.x.) However, with VirtualGL 2.1 and with these specific test machines, it was observed that one compression thread was sufficient on all server platforms to drive

the client to full capacity. Thus, adding a second compression thread only decreased the efficiency of the server without producing any increase in actual frame rate.

5.6 OpenGL vs. X11 Drawing

The VirtualGL client can draw images using either OpenGL or X11 commands. The use of OpenGL drawing is automatic (and necessary) when the VirtualGL client draws quad-buffered stereo images. Otherwise, OpenGL is used by default on Solaris/SPARC clients with 3D accelerators, and X11 is used by default on other platforms.

Comparison Between the Performance of OpenGL and X11 Drawing in the VirtualGL Client

	Actual Frame Rate (X11 Drawing)	**Actual Frame Rate (OpenGL Drawing)**
Linux → **Linux**	23.51 *[Baseline]*	26.41 **(+12%)**
Linux → **SPARC**	18.54 **(-44%)**	33.19 *[Baseline]*
Linux → **Windows**	24.01 *[Baseline]*	17.64 **(-27%)**

This validates VirtualGL's choice of default drawing methods on Windows clients (X11) and Solaris clients (OpenGL.) SPARC systems can send packed (3-byte) pixels to the framebuffer when using OpenGL, which is a large reason why OpenGL drawing is so much faster on these systems than X11 drawing (which is forced to use 4-byte pixels.) Most other platforms use 4-byte pixels for both OpenGL and X11.

The decrease in performance on Windows seemed to be due to overhead incurred by the `glDrawPixels()` operation in Exceed 3D. Low-level benchmarks showed that, in Exceed 3D 2008, drawing an image using `glDrawPixels()` was only about 1/5 as fast as drawing the same image using `XShmPutImage()`.

OpenGL performed better than X11 for this specific Linux client configuration, but there is a tremendous amount of variability in Linux graphics adapters. Many graphics adapters that one might find on a Linux PC client might not have accelerated OpenGL at all, so X11 is the safer default choice (and it performs well enough in this case as well.)

5.7 Software Gamma Correction

When an OpenGL application is displayed locally on a Solaris/SPARC workstation, the default behavior of Sun OpenGL is to gamma correct the output of the OpenGL application. VirtualGL simulates this behavior when remotely displaying from a Solaris/SPARC server to any type of client. If the client machine is also a SPARC machine, then VirtualGL will try to use a gamma-corrected X visual (thus allowing the graphics hardware on the SPARC client to perform the gamma correction.)

Otherwise, VirtualGL will perform gamma correction internally. VirtualGL's internal gamma correction mechanism can also be manually enabled when remotely displaying from non-SPARC servers. The following table shows the effect that VirtualGL's internal gamma correction mechanism has on the overall performance of the system when using the VGL Image Transport.

The Effect of Software Gamma Correction on the Performance of the VGL Image Transport

	Actual Frame Rate (no gamma)	**Actual Frame Rate (S/W gamma)**	**CPU-Limited Frame Rate (no gamma)**	**CPU-Limited Frame Rate (S/W gamma)**
Linux → Windows	24.01 *[Baseline]*	23.95 **(-0.25%)**	54.82 *[Baseline]*	43.15 **(-21%)**
SPARC → Windows	23.50 **(-1.1%)**	23.75 *[Baseline]*	43.52 **(+12%)**	38.93 *[Baseline]*
Solaris/x86 → Windows	23.01 *[Baseline]*	23.34 **(-1.4%)**	46.96 *[Baseline]*	43.22 **(-8.0%)**

Although enabling gamma correction did increase the encoding time, the effect on actual frame rate was negligible. The increase in encoding time was greater on the Linux server because the Linux version of VirtualGL uses straight C code to perform software gamma correction. The Solaris version of VirtualGL uses mediaLib to accelerate this task, and thus the encoding time increased less on Solaris servers when enabling software gamma correction.

Enabling gamma correction decreased the frame size by 4-7%. This makes intuitive sense, since gamma correction linearizes the color gradients used in Goraud shading and thus decreases the high frequency components in the image, allowing the JPEG compression algorithm to compress more efficiently.

5.8 Linux vs. Windows (Client)

Generally, no significant differences were observed between the performance of the Linux and Windows versions of VirtualGL when running on the same x86 client machine. There were a few notable exceptions, however:

- OpenGL drawing performed about 50% faster on the Linux client, due to the performance limitations of Exceed 3D which were described above.
- Quad-buffered stereo was 59% faster on the Linux client due to the same Exceed 3D performance limitations.
- SSh tunneling of the VGL Image Transport performed 34% faster on the Linux client due to performance limitations of PuTTY. SSh tunneling of the TurboVNC connection was 58% faster on the Linux client due to the same performance limitations.
- RGB image decoding was 19% faster when displaying to the Linux client (over a gigabit connection.)

- The transparent overlay test in GLXSpheres (`/opt/VirtualGL/bin/vglrun -sp /opt/VirtualGL/bin/glxspheres64 -o`) performed about 25% slower on the Linux client. The Linux nVidia drivers have historically not provided good performance when drawing pixels to a transparent overlay, and this seemed to be the portion of the test that caused the slow-down.

These disparities all involve limitations in the performance of one or more client software components, so they would not be likely to reveal themselves if the server or the network was the primary performance bottleneck (which would be the case when running on a low-bandwidth network, for instance.)

5.9 Linux vs. Solaris (Server)

Generally, no significant differences in actual frame rate were observed when remotely displaying from the x86 server running Linux and from the same x86 server running Solaris. However, for the baseline tests, the following differences in encoding time were observed:

- When using the VGL Image Transport, the 64-bit VirtualGL libraries on Solaris/x86 required 17% more time to encode each frame than the equivalent VirtualGL libraries on Linux (in other words, VirtualGL/Linux had a 17% higher CPU-limited frame rate.)
- When using the VGL Image Transport, the 32-bit VirtualGL libraries on Solaris/x86 required 41% more time to encode each frame than the equivalent VirtualGL libraries on Linux.
- TurboVNC on Solaris/x86 required 16% more time to encode each frame than TurboVNC on Linux.

While this efficiency gap has closed dramatically with the introduction of mediaLib 2.5, mediaLib 2.5 still uses more CPU cycles than the Intel Performance Primitives to compress the same JPEG images on the same hardware. This disparity affected the overall availability of the server's CPUs, but it did not cause any difference in actual frame rate. However, such a difference might reveal itself if multiple users were sharing the server.

The Solaris/x86 TurboJPEG codec produced frames that were about 10-15% larger across the board than the Linux version of the codec. This disparity in frame size would be unlikely to affect actual performance unless the network were the primary bottleneck.

5.10 SPARC vs. x86 (Server)

The following table lists several performance comparisons between the SPARC and x86 test servers, both running Solaris 10 and displaying to the Windows client:

	Actual Frame Rate (SPARC Server)	**Actual Frame Rate (x86 Server)**	**CPU-Limited Frame Rate (SPARC Server)**	**CPU-Limited Frame Rate (x86 Server)**
[Baseline] (VGL Image Transport)	23.75	23.01 **(-3.1%)**	38.93	46.96 **(+21%)**
32-bit (VGL Image Transport)	23.85	23.04 **(-3.3%)**	39.74	39.04 **(-1.8%)**
RGB (VGL Image Transport)	27.58	29.36 **(+6.5%)**	49.25	88.97 **(+81%)**
Anaglyphic Stereo (VGL Image Transport)	11.79	21.34 **(+81%)**	25.63	33.34 **(+30%)**
[Baseline] (TurboVNC)	17.21	22.70 **(+32%)**	25.69	34.92 **(+36%)**
RGB (TurboVNC)	15.29	18.34 **(+20%)**	23.89	49.57 **(+108%)**

Although both servers were able to drive the client to full capacity in most cases, the SPARC server generally required more CPU time to do so. One may wonder why the x86 server performed 81% faster when rendering anaglyphic stereo than the SPARC server, when it was only 30% more efficient. This reveals a bottleneck in the XVR-2500 framebuffer and drivers. When VirtualGL performs anaglyphic stereo rendering, it reads back individual color channels (GL_RED, GL_GREEN, and GL_BLUE) from the 3D graphics card to build the anaglyph. On nVidia hardware, reading back GL_RED, GL_GREEN, and GL_BLUE separately performs the same as if the entire image had been read back at once using GL_RGB. On the XVR-2500, however, reading back an individual color channel is significantly slower than reading back the entire framebuffer as RGB, and reading back all three color channels in sequence is slower still.

While it is true that comparing SPARC and x86 servers is not truly an “apples to apples” comparison, both of the workstations used in this test were current offerings in Sun's workstation product line at the time of publication. In fact, the SPARC workstation was the highest-end SPARC workstation that Sun produced at the time of this writing, and the x86 workstation was a lower-end model.

5.11 RGB vs. JPEG

When using the VGL Image Transport over the baseline gigabit link, RGB encoding generally produced faster results than JPEG encoding. RGB also required significantly less CPU time on the server, but it required significantly more network bandwidth:

Comparison of the Performance of JPEG and RGB Encoding When Using the VGL Image Transport

	Actual Frame Rate (JPEG) *[Baseline]*	Actual Frame Rate (RGB)	CPU-Limited Frame Rate (JPEG) *[Baseline]*	CPU-Limited Frame Rate (RGB)	Network-Limited Frame Rate (JPEG) *[Baseline]*	Network-Limited Frame Rate (RGB)
Linux → Linux	23.51	32.51 **(+38%)**	56.10	95.62 **(+70%)**	510.7	40.48
Linux → SPARC	33.19	25.15 **(-24%)**	60.57	90.13 **(+48%)**	512.5	40.41
Linux → Windows	24.01	27.27 **(+14%)**	54.82	90.28 **(+65%)**	499.4	40.60
SPARC → Windows	23.75	27.58 **(+16%)**	38.93	49.25 **(+26%)**	499.1	39.36
Solaris/x86→ Windows	23.01	29.36 **(+28%)**	46.96	88.97 **(+89%)**	432.3	39.01

The conclusion one can draw from this is that RGB encoding might be beneficial for point-to-point (single user) remote display in cases where the server CPU resources are more precious than the network resources and there is gigabit switched Ethernet between the server and the client. However, in a multi-user environment, RGB encoding quickly becomes an untenable proposition. A single gigabit link could not have even supported two concurrent clients using RGB encoding at the above frame rates. RGB is also not a tenable proposition for anything slower than gigabit. For these specific test cases, RGB encoding required about 25 Megabits to represent each frame. This frame size is significantly smaller than that of the X11 Image Transport, since RGB encoding transfers only 24 bits per pixel instead of 32. RGB encoding can also take advantage of VirtualGL's inter-frame differencing mechanism to further reduce the average frame size. However, 25 Megabits/frame is still far too large to achieve decent frame rates on even a 100 Megabit link.

Comparison of the Performance of JPEG and RGB Encoding When Using the VGL Image Transport (Quad-Buffered Stereo)

	Actual Frame Rate (JPEG) *[Baseline]*	Actual Frame Rate (RGB)	CPU-Limited Frame Rate (JPEG) *[Baseline]*	CPU-Limited Frame Rate (RGB)	Network-Limited Frame Rate (JPEG) *[Baseline]*	Network-Limited Frame Rate (RGB)
Linux → Linux	12.67	16.45 **(+30%)**	26.83	34.85 **(+30%)**	235	19.35
Linux → Windows	8.00	8.11 **(+1.4%)**	22.59	44.81 **(+98%)**	243	19.11

When used with the Linux client, RGB encoding provided significantly better stereo performance on a gigabit link than JPEG encoding. However, this was at the expense of even more network bandwidth usage. As one can see from the table above, driving the client with a decent frame rate required almost all of the available bandwidth.

Comparison of the Performance of JPEG and RGB Encoding When Using TurboVNC

	Actual Frame Rate (JPEG) *[Baseline]*	Actual Frame Rate (RGB)	CPU-Limited Frame Rate (JPEG) *[Baseline]*	CPU-Limited Frame Rate (RGB)	Network-Limited Frame Rate (JPEG) *[Baseline]*	Network-Limited Frame Rate (RGB)
Linux → Linux	28.44	25.66 **(-9.8%)**	42.45	64.62 **(+52%)**	463.8	29.53
Linux → SPARC	24.26	11.17 **(-54%)**	37.37	43.13 **(+15%)**	471.2	26.77
Linux → Windows	26.08	18.28 **(-30%)**	40.62	42.60 **(+4.9%)**	454.3	25.50
SPARC → Windows	17.21	15.29 **(-11%)**	25.69	23.89 **(-7.0%)**	472.5	26.59
Solaris/x86→ Windows	22.70	18.34 **(-19%)**	34.92	49.57 **(+42%)**	415.1	25.41

With TurboVNC, the results were mixed. The RGB codec in TurboVNC is essentially the same as the Raw codec in TightVNC, with few or no optimizations. Prior to sending the pixels, this encoding scheme converts them into the format preferred by the client's display, and this format usually contains 32 bits per pixel and not 24. This, combined with the lack of inter-frame differencing, produced frame sizes 40-60% larger than the RGB encoding implementation in the VGL Image Transport. At these frame sizes, a single user came close to saturating the entire gigabit link.

In general, the combination of larger frame sizes and pixel conversion overhead caused the frame rates to drop across the board, when compared to the JPEG baseline tests. In some cases, such as the SPARC client, this drop in performance was dramatic. As the frame rate dropped, more and more frames began to be spoiled on the server, which decreased its efficiency. In the SPARC → Windows case, this efficiency decrease caused the RGB test to perform less efficiently than even the JPEG test.

In certain cases, RGB encoding in TurboVNC might still be advantageous, but the advantages are much less clear than in the VGL Image Transport. For those desiring a lossless compression solution in TurboVNC, the Lossless Refresh feature might be a better option than RGB encoding.

5.12 Gigabit vs. 100 Megabit

When driving the client to full capacity, VirtualGL generally used less than 50 Megabits/second of network bandwidth in the baseline configuration. Thus, if the server had a 100 Megabit interconnect, it could have accommodated two users before saturating the interconnect.

There were, however, some features in VirtualGL which required gigabit connectivity in order to achieve decent performance:

- RGB encoding required generally about 25 Megabits/frame when using the VGL Image Transport and 35-40 Megabits/frame when using TurboVNC, and thus it was limited to 2-4 frames/second on a 100 Megabit link.
- Since it uses the X11 Image Transport, synchronous mode (`VGL_SYNC=1`) required 36-38 Megabits/frame when displayed to a remote X server. It was thus limited to less than 3 frames/second on a 100 Megabit link.
- Since it uses the X11 Image Transport but transfers only 1 byte/pixel, color index rendering required about 9 Megabits/frame when displayed to a remote X server. This produced usable performance on a 100 Megabit link, but gigabit was required to drive the client to full capacity.

The conclusion one can draw is that gigabit is a good idea for the server network, but 100 Megabit switched Ethernet to each client (or even wireless) should provide more than enough bandwidth for VirtualGL or TurboVNC when using the default (perceptually lossless JPEG) encoding setting.

5.13 TurboVNC vs. the VGL Image Transport

Comparison of Baseline Performance Between the VGL Image Transport and TurboVNC

	Actual Frame Rate (VGL Image Transport)	Actual Frame Rate (TurboVNC)	CPU-Limited Frame Rate (VGL Image Transport)	CPU-Limited Frame Rate (TurboVNC)
Linux → Linux	23.51	28.44 **(+21%)**	56.10	42.45 **(-24%)**
Linux → SPARC	33.19	24.26 **(-27%)**	60.57	37.37 **(-38%)**
Linux → Windows	24.01	26.08 **(+8.6%)**	54.82	40.62 **(-26%)**
SPARC → Windows	23.75	17.21 **(-28%)**	38.93	25.69 **(-34%)**
Solaris/x86→ Windows	23.01	22.70 **(-1.3%)**	46.96	34.92 **(-26%)**

Although greatly improved vs. TurboVNC 0.3.3, TurboVNC 0.4 on SPARC servers still has performance issues which bear further investigation. TurboVNC on Linux was generally observed to be faster than the VGL Image Transport. On Solaris/x86, the two systems performed at parity, although the VGL Image Transport was significantly more efficient in its use of the server CPUs (as was the case on all server platforms.)

The VGL Image Transport produced 4-10% smaller frames than TurboVNC, due primarily to the VGL Image Transport's inter-frame differencing mechanism.

5.14 SSL and SSh Encryption

Comparison of the Performance of SSL Encryption (`vglrun +s`) and SSh Encryption (`vglconnect -s`) When Using the VGL Image Transport

	Actual Frame Rate (Un-encrypted) *[Baseline]*	Actual Frame Rate (SSL Encryption)	Actual Frame Rate (SSh Tunneling)	CPU-Limited Frame Rate (Un-encrypted) *[Baseline]*	CPU-Limited Frame Rate (SSL Encryption)	CPU-Limited Frame Rate (SSh Tunneling)
Linux → Linux	23.51	18.58 **(-21%)**	20.67 **(-12%)**	56.10	41.84 **(-25%)**	42.44 **(-24%)**
Linux → SPARC	33.19	32.56 **(-1.9%)**	32.57 **(-1.9%)**	60.57	52.26 **(-14%)**	53.48 **(-12%)**
Linux → Windows	24.01	19.85 **(-17%)**	*PuTTY:* 15.44 **(-36%)** *OpenSSH:* 19.22 **(-20%)**	54.82	44.91 **(-18%)**	*PuTTY:* 40.51 **(-26%)** *OpenSSH:* 41.78 **(-24%)**
SPARC → Windows	23.75	19.85 **(-16%)**	16.45 **(-31%)**	38.93	33.08 **(-15%)**	32.25 **(-17%)**
Solaris/x86 → Windows	23.01	19.22 **(-16%)**	15.44 **(-33%)**	46.96	41.78 **(-11%)**	39.58 **(-16%)**

Tunneling the VGL Image Transport through PuTTY (using `vglconnect -s` on the Windows client) generally performed about 20% slower than using VirtualGL's built-in SSL encryption feature (`vglrun +s`). However, SSL and SSh performed at parity on the SPARC client, and SSh was actually a bit faster on the Linux client. Using Cygwin OpenSSH instead of PuTTY (`set VGLCONNECT_OPENSSH=1`) on the Windows client resulted in performance which almost matched that of VGL's built-in SSL encryption feature.

In any sense, these results show that, if the performance issue in PuTTY could be eliminated, then VirtualGL's built-in SSL encryption feature would no longer be necessary. It should be made clear that these performance differences would only reveal themselves on a high-speed network. On low-bandwidth and high-latency links, which are the primary environments in which SSh and SSL would be used, there was no observed difference between the performance of SSh and SSL.

There was also no significant difference observed between the frame sizes generated by SSL and SSh tunneling.

Comparison of the Performance of SSh Encryption and Unencrypted Transmission When Using TurboVNC

	Actual Frame Rate (Un-encrypted) *[Baseline]*	**Actual Frame Rate (SSh Tunneling)**	**CPU-Limited Frame Rate (Un-encrypted)** *[Baseline]*	**CPU-Limited Frame Rate (SSh Tunneling)**
Linux → Linux	28.44	24.44 **(-14%)**	42.45	36.97 **(-13%)**
Linux → SPARC	24.26	22.59 **(-6.9%)**	37.37	33.42 **(-11%)**
Linux → Windows	26.08	*PuTTY:* 15.46 **(-41%)** *OpenSSH* 21.22 **(-19%)**	40.62	*PuTTY:* 21.68 **(-47%)** *OpenSSH:* 30.23 **(-26%)**

As with the VGL Image Transport, PuTTY's performance lagged significantly behind that of OpenSSH.

The numbers above also reveal one of VNC's dirty little secrets. As explained in Section 5.2, the VNC protocol (RFB) is a client-driven protocol. Image updates are generally only sent to the client if it requests them. Since the VNC X server is single-threaded, it divides its time between processing X requests and processing RFB requests. If the VNC viewer and the network link are fast enough that each frame can be transmitted and decoded in equal or less time than it took to encode the frame, then the VNC server will always have a pending framebuffer request, and thus every frame that VirtualGL draws into the X server will be sent to the client. However, if the network or the viewer is too slow to keep up with the server, then the server has plenty of time to process multiple `PutImage` requests from VirtualGL while it is waiting for the next framebuffer update request from the client. Since these interim `PutImage` requests alter the virtual framebuffer but do not result in pixels being sent to the viewer, VNC is effectively spoiling those frames.

In general, this shouldn't be an issue when running interactive applications, because the frame rate of interactive applications is gated by mouse movement or other factors. However, the introduction of spurious frame spoiling at low frame rates makes TurboVNC difficult to benchmark accurately, since the spoiled frames add to the average encoding time of each transmitted frame. To put this in perspective, though, other VNC solutions are worse, since they will generally never be able to achieve sufficient frame rates to avoid spoiling. At least with TurboVNC, there is a reasonable expectation that it will not spoil frames in the baseline configuration.

Further investigation of this topic is definitely warranted.

5.15 X11 Forwarding vs. Direct X11

When using a remote X display connection, there was generally no observable difference between the performance of VirtualGL when using SSh X11 Forwarding (`vglconnect` with no arguments) and when using a direct X11 connection (`vglconnect -x`). The exceptions were modes which forced the use of the X11 Image Transport over the remote X display connection. Specifically, synchronous mode (`vglrun +sync`), color index rendering (`glxspheres -c`), and proxy compression mode (`vglrun -c proxy`) were quite a bit slower when tunneling the X11 protocol through SSh. SSh effectively acted as a bandwidth limiter, limiting the bandwidth as follows:

Effective Bandwidth Limit of SSh X11 Forwarding

	Effective Bandwidth Limit (Synchronous Mode)	Actual Frame Rate (Synchronous Mode)	Effective Bandwidth Limit (Color Index)	Actual Frame Rate (Color Index)
Linux → Linux	274 Mbits/s	7.35	288 Mbits/s	30.74
Linux → SPARC	177 Mbits/s	4.75	129 Mbits/s	13.93
Linux → Windows	68.1 Mbits/s	1.84	68.6 Mbits/s	7.26

In all of the above cases, the frame rate improved dramatically (2-10x) when using a direct X11 connection. Even with a direct X11 connection, however, synchronous mode from the Linux server to the SPARC client was still slow due to pixel conversion overhead.

5.16 Stereographic Rendering

Comparison Between the Performance of Stereographic and Monographic Rendering in the VGL Image Transport

	Actual Frame Rate (Mono) *[Baseline]*	**Actual Frame Rate (Quad-Buffered Stereo)**	**Actual Frame Rate (Anaglyphic Stereo)**	**CPU-Limited Frame Rate (Mono)** *[Baseline]*	**CPU-Limited Frame Rate (Quad-Buffered Stereo)**	**CPU-Limited Frame Rate (Anaglyphic Stereo)**
Linux → Linux	23.51	12.67 **(-46%)**	21.71 **(-7.7%)**	56.10	26.83 **(-52%)**	37.68 **(-33%)**
Linux → SPARC	33.19	14.20 **(-57%)**	25.99 **(-22%)**	60.57	28.79 **(-52%)**	40.29 **(-33%)**
Linux → Windows	24.01	8.00 **(-67%)**	21.25 **(-11%)**	54.82	22.59 **(-59%)**	36.95 **(-33%)**
SPARC → Windows	23.75	7.98 **(-66%)**	11.79 **(-50%)**	38.93	18.56 **(-52%)**	25.63 **(-34%)**
Solaris/x86 → Windows	23.01	7.78 **(-66%)**	21.34 **(-7.3%)**	46.96	21.61 **(-54%)**	33.34 **(-29%)**

Most of these figures make intuitive sense. To support quad-buffered stereo rendering, VirtualGL must compress and send twice the data over the network. So quad-buffered stereo should be about half as fast and half as efficient as mono, which it generally was when displaying to the Linux and SPARC clients. However, quad-buffered stereo incurred an additional penalty on the Windows client due to the 27% slow-down in OpenGL drawing vs. X11 drawing in Exceed (described in Section 5.6.) This compounded the slow-down and made quad-buffered stereo only about 1/3 as fast as mono on the Windows client, rather than only half as fast. Anaglyphic stereo generally performed a lot better and was only 1/3 less efficient than mono. The exception here was the SPARC server, which was limited by the readback performance issue described in Section 5.10.

In terms of frame size, quad-buffered stereo produced frames that were about twice the size of mono (which is intuitively obvious), and anaglyphic stereo produced frames that were 30-40% larger than mono. The latter was due primarily to the additional window coverage caused by the overlapping stereo images, which reduced the amount of solid background in each frame.

In TurboVNC, the situation was similar. Anaglyphic stereo produced frames 25-45% larger than mono, used the server's CPUs 25-30% less efficiently, and reduced the actual frame rate by 5-15% (except when displaying from the SPARC server, which was also limited to about 11 frames/second.)

5.17 Interactive Performance

In order to simulate a realistic "worst-case" scenario for interactive applications, GLXSpheres includes a mode in which it will wait for a mouse event before rendering each frame. Since the image workload this generates is very similar to the workload generated by GLXSpheres in its default, non-interactive mode, results from the two modes can be reasonably compared. This allows one to quantify the performance differences between a benchmark environment (with frame spoiling disabled) and a realistic interactive environment (with frame spoiling enabled.)

The reason why frame spoiling exists in VirtualGL is because the X server or proxy generally samples the mouse at a much faster rate (40-60 Hz) than VirtualGL is able to deliver frames to the client. Thus, if the application had to wait for VirtualGL to finish transmitting a frame before it could render a new one, then the mouse would get ahead of the 3D rendering, and the user would perceive a lag in responsiveness. Frame spoiling allows the 3D rendering to be synchronized with the mouse rather than with the image transmission pipeline. But since VirtualGL cannot usually transmit 60 frames/second to the client, unneeded frames have to be rendered, read back, and discarded so that the movement of the 3D scene appears to track the mouse movement. Unfortunately, this causes additional server overhead, but it's the only way to prevent the 3D application from feeling "draggy."

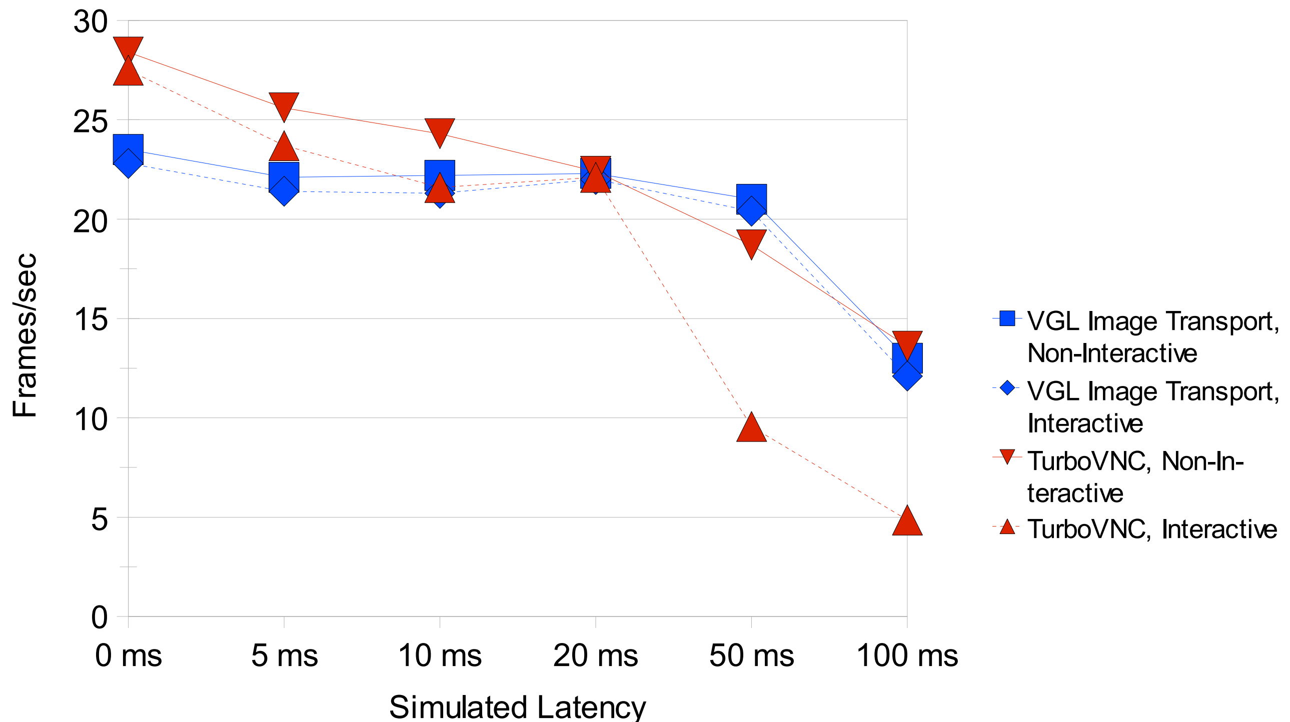
The performance of VirtualGL was compared when using both the interactive mode of GLXSpheres and the non-interactive mode (baseline), in order to quantify the additional load incurred by rendering and processing the spoiled frames on the server. When running the interactive tests, the mouse was moved continuously while measuring the frame rate, CPU load, and network load. The measuring tools were started on a 10-second delay in order to give the system enough time to reach steady state.

In general, running GLXSpheres interactively using the VGL Image Transport with frame spoiling enabled caused only a slight (no more than 5%) drop in actual frame rate relative to the baseline. The interactive cases used the server's CPUs 22-33% less efficiently than the baseline cases, and there was no significant difference in frame sizes between the interactive and baseline cases.

In TurboVNC, the interactive test cases reduced the actual frame rate by no more than 7% relative to the baseline and were only 2-5% less efficient in their use of the server CPUs than the baseline. TurboVNC already has a significantly higher encoding time than the VGL Image Transport, so the interactive tests tended to equalize this.

5.18 The Effect of Increasing Network Latency on VirtualGL's Performance

NISTnet was used to artificially add network latency to both the Linux server and the Linux client, simulating the latency of a long-haul connection. NetTest was used to verify that the correct amount of latency had been added. The bandwidth was not altered from the baseline (gigabit) case.



TurboVNC's interactive performance was much worse than that of the VGL Image Transport when the latency of the network was increased beyond 20 ms. For higher latency values, the interactive frame rate of TurboVNC began to be dominated almost entirely by the round-trip time (2x the latency) from client to server. As described in Section 5.2, RFB is a client-driven protocol. The VNC server will not send a frame until requested by the client, and the client will not send a new request until it has received a previous frame. So it is expected that each transmitted frame will require a round trip from server to client. However, it is unclear why this protocol deficiency affected only the interactive tests and not the non-interactive tests.

In contrast to TurboVNC, the VGL Image Transport protocol pushes frames from the server rather than pulling them from the client, and this eliminates the need for any round trips. The performance of the VGL Image Transport was limited only by the overhead of TCP/IP at higher latencies, and it is possible that increasing the TCP window size would improve that situation.

It is important to note that the above test compared only the steady state performance of GLXSpheres. It did not compare the amount of time it took to start up the application, display the initial window, and reach steady state. Even with the incredibly simple GUI in GLXSpheres (a single X window), the application startup time over the simulated 100 ms link was on the order of several minutes when using the VGL Image Transport and on the order of several seconds when using TurboVNC. But once the application reached steady state, the VGL Image Transport was the better performing of the two protocols.

5.19 The Effect of Decreasing Bandwidth on VirtualGL's Performance

The following table lists the observed frame sizes of the various VirtualGL and TurboVNC protocols as well as their theoretical network-limited frame rates on various interconnects. All quoted frame sizes are from the Linux → Linux baseline tests, but similar results were obtained on other platforms (except Solaris/x86, which produced frame sizes that were about 10-15% larger.)

	Frame Size (Mbits)	Network-Limited Frame Rate (Gigabit)	Network-Limited Frame Rate (100 Mbit)	Network-Limited Frame Rate (10 Mbit)	Network-Limited Frame Rate (5 Mbit)	Network-Limited Frame Rate (1 Mbit)
VGL Image Transport (JPEG 1X Q95)	2.0	500	50	5.0	2.5	0.50
VGL Image Transport (JPEG 2X Q80)	0.85	1200	120	12	6.0	1.2
VGL Image Transport (JPEG 4X Q30)	0.43	2300	230	23	12	2.3
VGL Image Transport (RGB)	25	40	4	0.40	0.20	0.040
TurboVNC (JPEG 1X Q95)	2.2	450	45	4.5	2.3	0.45
TurboVNC (JPEG 2X Q80)	1.0	1000	100	10	5.0	1.0
TurboVNC (JPEG 4X Q30)	0.57	1800	180	18	8.8	1.8
TurboVNC (RGB)	34	29	2.9	0.29	0.15	0.029

Green cells indicate that the frame rate is likely to be limited by the network (rather than the client), but the usability should still be good (10 fps or greater.) Yellow cells indicate that the network is likely to limit the frame rate, and the usability would likely be only marginal (5-10 fps.) Red cells indicate that the network is likely to limit the frame rate to an unusable level (< 5 fps.)

At the moment, neither VirtualGL nor TurboVNC offers a good solution for 1 Megabit and slower networks (DSL, T1, etc.) "Low Quality" mode (JPEG compression with 4x chrominance subsampling and quality=30) is generally about the lowest usable quality level that still allows text to be read on the screen, and even that low level of quality is incapable of producing more than about 2 frames/second on a 1 Megabit link. With a 5 Megabit link (which is about equivalent to a good cable modem connection), the "Low Quality" setting produces frame rates that are marginal to good.

Various configurations were tested over a 100 Megabit link as well as a 10 Megabit link (simulated

with NISTnet), and all of the results matched very closely to the theoretical results listed above. It should be noted, however, that this test did not factor in network latency. When running VirtualGL on a network that is both high-latency and low-bandwidth, the above results represent a best-case scenario that will almost never be achieved. It should also be noted that there is a huge amount of variability in the size of frames compressed with JPEG. For VirtualGL's baseline JPEG compression settings (quality=95 with no chrominance subsampling), we have observed compression ratios as bad as 1:1 and as good as 100:1, depending on the type of image being compressed. The baseline GLXSpheres tests generally produced images that compressed with a ratio of between 10:1 and 15:1. YMMV.

5.20 Simultaneous Usage by Multiple Clients

In order to simulate the effect of multiple clients using a VirtualGL server system at the same time, both the Linux client and SPARC client were driven simultaneously by the Linux server. The server was able to drive both clients at 23 frames/second using 100% of its CPUs and using 89 Megabits/second of network bandwidth.

This validates the provisioning rule of thumb for VirtualGL that each simultaneous user should be allocated a single CPU and 50 Megabits/second of network bandwidth.

One important note, however – when running this test, it was observed that the readback performance for the server's nVidia Quadro card would slow down dramatically after a few minutes, causing the steady state performance to drop by nearly half on both clients. This could be due to running out of Pbuffer space, although the server's graphics card had 256 MB of graphics memory and should not have run out of space with only two users. At any rate, this highlighted a potential problem with multi-user framebuffer sharing which bears further investigation.

5.21 RealVNC and NX

RealVNC, TightVNC, and NX, three popular X proxies with similar features to TurboVNC, were compared to TurboVNC in terms of actual frame rate, server CPU usage, and network usage. The results are as follows (these were all measured with the Linux server and the Linux client.)

	Actual Frame Rate	**CPU-Limited Frame Rate**	**Frame Size (Mbits)**
TurboVNC 0.4 (JPEG 1X Q95)	28.44	42.45	2.16
TurboVNC 0.4 (JPEG 2X Q80)	36.42	50.09	1.00
TurboVNC 0.4 (RGB)	25.66	64.62	33.86
RealVNC 4.1.2 (ZRLE 16M)	8.15	14.02	4.74
RealVNC 4.1.2 (Hextile 16M)	12.58	20.56	16.24
RealVNC 4.1.2 (Raw 16M)	26.08	41.40	18.76
NX 3.1.0 (JPEG 2X Q80 [qual=9], no SSL, no Zlib)	9.13	16.91	0.88
NX 3.1.0 (RGB, no SSL, no Zlib)	6.53	10.59	15.33
TightVNC 1.3.9 (JPEG 2X Q80 [qual=9])	13.73	23.27	0.94

This chart shows pretty clearly why TurboVNC exists. Its parent implementation, TightVNC, requires more than double the CPU cycles to compress each frame (at the same quality) and performs nearly three times more slowly, all for only a 6% savings in frame size. To be fair, TightVNC isn't designed for full-screen video. It provides optimizations that are mainly targeted toward running 2D applications on extremely low-bandwidth links, and it's quite likely that it does a better job of this than TurboVNC does. However, for the types of workloads generated by VirtualGL, TurboVNC is the clear winner. Not shown are the results from the Windows client, on which TightVNC produced only 2 frames/second of actual performance (for unexplained reasons.)

RealVNC's Hextile and Raw protocols proved quite usable on a gigabit connection (if one could ignore the lack of double buffering.) The Raw protocol apparently uses some sort of additional compression (probably Zlib), which would explain the additional CPU overhead as well as the reduced frame size (relative to the RGB encoding implementation in TurboVNC.) The ZRLE protocol shows promise as a lossless compression solution for 100 Megabit links, if the frame rate could be increased somehow.

Even without any added compression or encryption, NX still had a significantly higher encoding time

than any of the other proxies. As an experiment, the libjpeg codec in NX 3.1.0 was replaced with TurboJPEG, the same codec used by TurboVNC and VirtualGL. Switching to TurboJPEG improved NX's efficiency by about 50%, but even with this improvement, it was still using about twice the CPU cycles as TurboVNC to compress each frame. NX was never able to achieve more than 10-11 frames/second, even with the help of TurboJPEG. On Windows, the same experiment never produced more than 6-7 frames/second. This definitely bears further investigation, as there seems to be some unknown factor which causes exceptionally high CPU overhead on the NX server.

5.22 TurboVNC Java Viewer

Comparison of the Performance of the Native and Java TurboVNC Viewers

	Actual Frame Rate (Native) *[Baseline]*	**Actual Frame Rate (Java)**	**CPU-Limited Frame Rate (Native)** *[Baseline]*	**CPU-Limited Frame Rate (Java)**
Linux → Linux	28.44	10.78 **(-62%)**	42.45	24.72 **(-42%)**
Linux → Windows	26.08	10.26 **(-61%)**	40.62	19.43 **(-52%)**

The Java viewer was tested on both the Linux and Windows clients, remotely displaying from the Linux server. It was observed to be a bit more than 1/3 as fast as the native viewer, which is consistent with its use of the slower libjpeg codec rather than TurboJPEG. However, note that the efficiency nastiness described in Section 5.14 was also observed here. The lower frame rates produced by the Java viewer caused the server to spoil frames, which (on average) halved its efficiency.

5.23 Indirect OpenGL Rendering vs. VirtualGL

People often ask why VirtualGL is necessary when their application "runs just fine" with indirect OpenGL rendering. Some applications do produce acceptable performance in an indirect OpenGL environment, but generally that is only the case if the application uses display lists, if the 3D model is relatively small, and if textures are not used. Applications which do not use display lists must send each vertex of the 3D model to the 3D graphics card every time a frame is drawn (this is called "immediate mode" rendering.) With indirect OpenGL, the 3D graphics card is located on the client machine, so using immediate mode rendering requires sending every vertex of the 3D model from server to client every time a frame is rendered.

Display lists essentially allow an application to cache 3D vertex data on the graphics card, meaning that the vertices for a 3D model only have to be sent once. However, display lists are only suitable for static 3D models, that is models whose geometry does not change from frame to frame. For obvious reasons, this makes display lists unsuitable for most design applications, because the whole point of a design application is to modify a 3D model in real time. For volume visualization applications, the use of display lists is moot, because volume viz applications generally send most of their data in the form

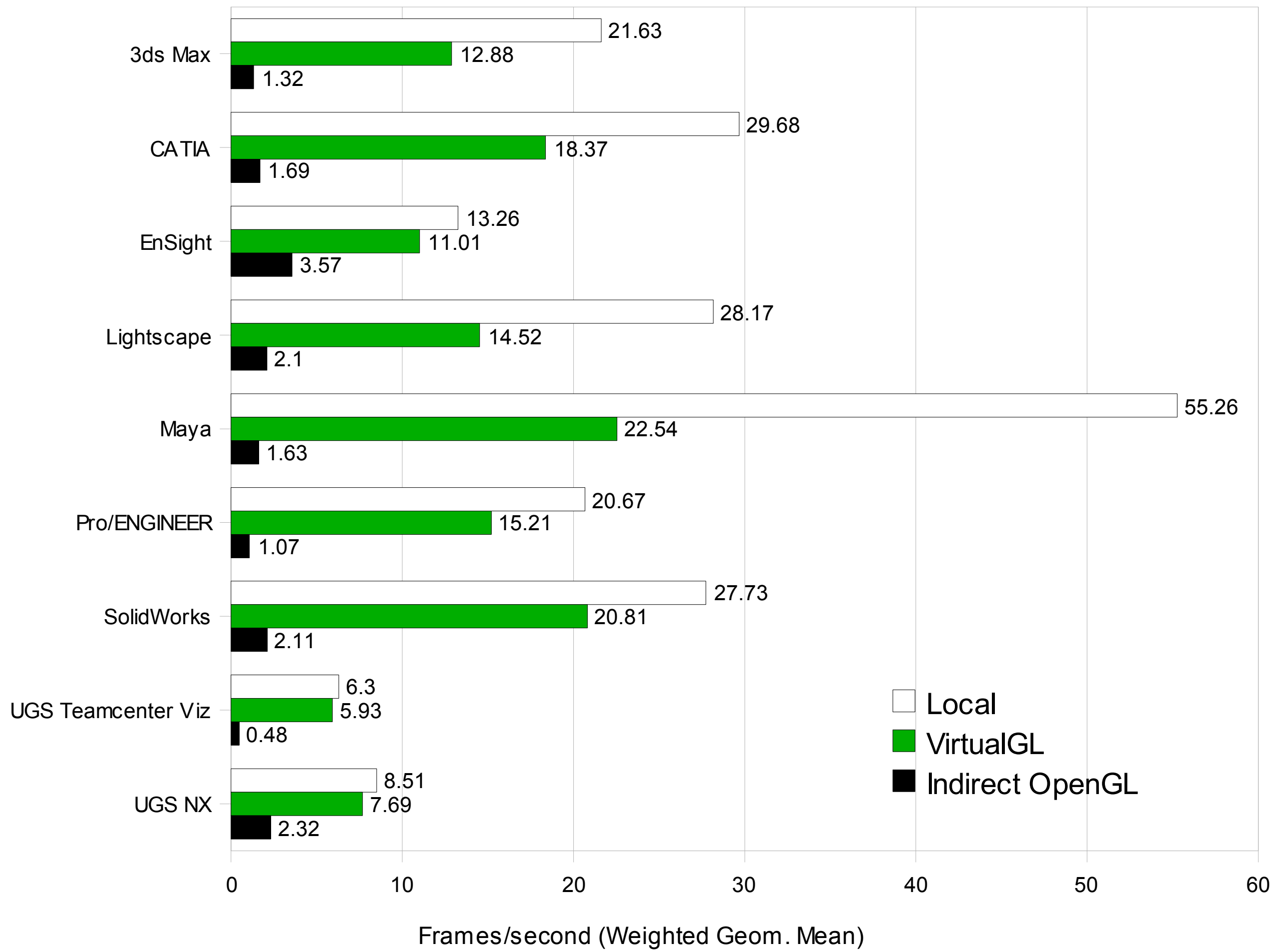
of textures. For instance, an application which is passing a planar probe through a gigavoxel-sized volumetric dataset may generate several megabytes of new textures for every frame while using only a handful of vertices.

VirtualGL originated in the oil & gas industry, an industry which regularly deals with very large geometric and volumetric datasets. These datasets are large enough that transmitting them over even a gigabit network is prohibitively slow, much too slow to even consider doing in real time. But this test attempts to show that indirect OpenGL rendering is also prohibitively slow with more “normal-sized” datasets, the types of datasets that any design engineer might create.

SPECviewperf is a suite of OpenGL benchmarks which measures the performance of 3D hardware under simulated workloads from popular 3D mechanical CAD and digital content creation applications. Application vendors work with the SPEC OpenGL Performance Committee (OPC) to define a 3D dataset which is representative of those used by their customers as well as a list of typical rendering modes and what percentage of time (on average) each of those modes is used by customers in the field. The breakdown of percentages for each rendering mode is used to develop a composite score for each application, expressed as a weighted geometric mean of frame rates for all of the rendering modes (see http://www.spec.org/gpc/opc.static/geometric.html for more details.)

As described in Section 1.1, the GLXSpheres benchmark is designed to generate an image workload rather than a geometry workload, so it uses display lists and renders relatively few polygons. Thus, it is an unrealistic benchmark to use for this particular test, since what we're really testing here is the remote display solution's ability to handle medium-to-large 3D models.

For this test, the Linux server and Linux client were used in their baseline configurations (connected by gigabit Ethernet.) A direct X11 connection was used so that SSh would not slow down the GLX protocol stream in the indirect OpenGL tests. VirtualGL was used with the VGL Image Transport enabled and frame spoiling disabled. The “local” tests were run on the VirtualGL server as a reference (this is described in more detail below.)



Since the SPECviewperf benchmarks are designed to simulate real application behavior, only the UGS/NX viewset uses display lists. Even though that viewset uses display lists, the models it contains are large enough (up to 30 million vertices) that it still performs poorly with indirect OpenGL rendering. Even on a gigabit network, none of the indirect OpenGL tests generated frame rates high enough to be considered usable in a 3D application.

The local display tests are included as a reference, since they show areas in which the performance was limited by the 3D hardware rather than by the remote display software. In particular, the UGS, EnSight, and Pro/ENGINEER tests seemed to be mostly 3D hardware-limited, so relatively little drop in frame rate was observed by running those tests remotely through VirtualGL. SolidWorks, CATIA, and Maya appeared to be mainly limited by the speed of the VirtualGL client, whereas Lightscape and 3ds Max were very likely limited by the JPEG compressor. The 3ds Max benchmark generates a great deal of stippled images, whereas the Lightscape benchmark generates a great deal of complex, multi-colored wireframe images. Both of these types of images have very high frequency components and are thus corner cases for JPEG compression. The images generated by the 3ds Max and Lightscape benchmarks tend to compress very inefficiently, thus requiring more CPU and network resources. Further research, including an analysis of CPU and network usage when running SPECviewperf, is

definitely warranted.

6 Summary

Red text indicates areas in need of further research.

- On SPARC platforms, VGL 2.1 performed significantly faster than VGL 2.0.x.

- On Solaris/x86 servers, VGL 2.1 performed noticeably faster than VGL 2.0.x when running 32-bit applications.

- VGL performed significantly faster on Solaris/x86 platforms when using mediaLib 2.5 rather than mediaLib 2.4. However, TurboJPEG/IPP still require 20-40% less CPU time to compress each frame than TurboJPEG/mediaLib. Also, mediaLib 2.5 produces JPEG frames that are 10-15% larger than those produced by IPP. It's unclear if any further optimizations in mediaLib would be possible, but there is still room for them. The disparity between the efficiency of TurboJPEG/IPP and TurboJPEG/mediaLib did not generally affect the actual frame rate with a single user.

- There was generally no significant performance difference between running 32-bit and 64-bit apps in VirtualGL, except on Solaris/x86. On that platform, the 32-bit version of VirtualGL used more CPU time to encode each frame than the 64-bit version. However, this would not likely translate into a measurable difference in frame rate except in a heavily-loaded multi-user environment.

- A second server CPU was shown to improve single-user performance in some cases, but not by much. Generally, 1 CPU per active user was validated to be a good provisioning rule.

- Enabling additional compression threads on the server (`VGL_NPROCS=2`) could not be shown to increase performance in any case, and it usually increased the encoding time on the server.

- Enabling software gamma correction (the default on Solaris/SPARC servers) increased the encoding time by a small amount, but not enough to affect the actual frame rate with a single user.

- The default VirtualGL Client drawing method (OpenGL on SPARC systems with 3D accelerators, X11 drawing on other systems) was validated to be the fastest approach. Exceed 3D was observed to perform much worse with OpenGL drawing than with X11 drawing.

- Linux and Windows clients generally performed equally, but the Linux client was significantly faster when performing OpenGL drawing (including drawing quad-buffered stereo images), when tunneling the VGL Image Transport or TurboVNC through SSh, and when decoding RGB images.

- While all platforms were able to drive the client machine to full capacity, the SPARC server

required significantly more server CPU resources to do so, both when using JPEG and RGB encoding. The SPARC server additionally exhibited poor readback performance for GL_RED, GL_GREEN, and GL_BLUE pixel formats, which caused the performance of anaglyphic stereo to suffer.

- RGB encoding was found to be a reasonable solution for point-to-point (single user) display over a gigabit link using the VirtualGL Image Transport. In this context, it provided a bit more performance then JPEG and used significantly less CPU resources on the server. And, of course, RGB is fully lossless. However, RGB encoding in TurboVNC was generally slower than JPEG, and it was only more efficient in certain cases.

- There was generally no advantage to having a gigabit connection to the client except when using RGB encoding, synchronous mode (with a remote X display connection), and color index rendering (also with a remote X display connection.)

- TurboVNC was generally found to be as fast or faster than the VGL Image Transport, except on SPARC servers and clients, but TurboVNC used significantly more server CPU resources on all platforms. Any decrease in overall frame rate caused the TurboVNC server to spoil frames, which increased its server resource usage further still.

- There was generally no advantage to using VirtualGL's built-in SSL encryption mechanism (`vglrun +s`) when compared to SSh tunneling (`vglconnect -s`), except on Windows clients. On Windows clients, when using a high-speed network, SSL encryption was found to be faster due to performance limitations in PuTTY. Similarly, tunneling TurboVNC through SSh was faster on the Linux client vs. the Windows client due to the same limitations in PuTTY.

- On a remote X11 connection, synchronous mode was found to perform well only when using gigabit connectivity and a direct X11 connection (as opposed to an SSh-forwarded X11 connection.)

- Color index rendering was found to be usable on an SSh-forwarded remote X11 connection, but it only achieved full performance with a direct X11 connection.

- Quad-buffered stereo generally performed at half the frame rate of mono and used twice the server CPU resources, except on Windows clients, where it performed at only 1/3 the frame rate of mono due to performance limitations in Exceed 3D.

- Anaglyphic stereo generally performed only a bit slower than mono, except on Solaris/SPARC servers (which were limited by a readback performance issue.) Anaglyphic stereo generally required about 50% more encoding time on the server (in other words, it was 1/3 less efficient in its use of the server's CPUs.)

- Running GLXSpheres interactively using the VGL Image Transport and frame spoiling generally caused about a 25-50% increase in encoding time on the server but did not otherwise affect performance. Running the test application interactively in TurboVNC did not cause any significant change in performance vs. the non-interactive tests.

- The VNC (RFB) protocol requires a round trip between client and server to transmit every frame, and this limited its performance severely on high-latency connections when compared to the VGL Image Transport protocol.

- "High quality" JPEG (no chrominance subsampling, quality=95) was generally observed to require 20 Megabits/second of bandwidth to achieve acceptable performance (10 fps) and 50 Megabit/second to achieve full performance.

- "Medium quality" JPEG (2X chrominance subsampling, quality=80) was generally observed to require 10 Megabits/second of bandwidth to achieve acceptable performance and 20 Megabits/second to achieve full performance.

- "Low quality" JPEG (4X chrominance subsampling, quality=30) was generally observed to require 5 Megabits/second of bandwidth to achieve acceptable performance and 10 Megabits/second to achieve full performance.

- No encoding solution currently available in TurboVNC or VirtualGL produced acceptable performance on 1 Megabit/second connections. A tighter codec than JPEG is needed to support 1 Megabit/second and smaller pipes. Problem: the codec must also be compressible at 5+ frames/second. We don't know of anything that currently fits this bill.

- The dual-processor Opteron server was able to drive two clients to full capacity, using 100% of both of its CPUs and nearly 100% of a 100 Megabit/second link. However, the nVidia card in the server began to produce substantially slower readback performance after a few minutes of this, for reasons which are unknown. The current provisioning rule of thumb (1 CPU and 50 Megabits/second for each simultaneous user) was validated.

- NX generally produced only marginal performance with VirtualGL, mostly due to its 4-7x greater encoding time (when compared with TurboVNC.)

- RealVNC's Raw encoding mode proved to be a good solution on gigabit networks, performing about equally to RGB encoding in TurboVNC but requiring much less bandwidth (due to Zlib compression, we suspect.) ZRLE and Hextile produced marginal-to-acceptable frame rates (also on a gigabit connection) but required 2-3X the encoding time of TurboVNC's default JPEG compression mode.

- TightVNC required more than double the encoding time to compress the same JPEG frames as TurboVNC, and TightVNC (in the best case) performed only 40% as fast. When displaying to the Windows client, TightVNC performed less than 1/10 as fast as TurboVNC. TightVNC produced frames that were only 6% smaller than those of TurboVNC.

- On high-speed connections, the native TurboVNC viewer was observed to be nearly 3X as fast as the Java viewer. Additionally, the Java viewer's low frame rates caused the TurboVNC server to spoil more frames, decreasing its efficiency.