From Tight to Turbo and Back Again:
Designing a Better Encoding Method for TurboVNC

Version 2b, 7/29/2015 -- The VirtualGL Project

This report attempts to explain and chronicle the performance trade-offs made when designing and developing TurboVNC, as well as to establish a reproducible method for comparing the encoding performance of TurboVNC with other VNC solutions at the low level.

1 Background

In the fall of 2004, Landmark Graphics Corporation open sourced VirtualGL, making it the first open source remote 3D framework of its kind. It quickly became apparent, however, that VirtualGL needed additional software to provide collaboration capabilities and to improve its performance on wide-area networks. VNC seemed like a reasonable candidate, since it is designed to provide precisely those two functions, but none of the off-the-shelf VNC distributions had enough performance to transmit the output of VirtualGL at "interactive" frame rates. The idea arose to take the existing TightVNC 1.2.9 code and modify it to use VirtualGL's accelerated JPEG codec (TurboJPEG.) In and of itself, however, this enhancement was insufficient to provide the necessary levels of performance. To explain why requires a brief discussion of how TightVNC works.

VNC servers send "framebuffer updates" to any VNC client that is connected to them. These updates can contain either a full image of the VNC server's remote desktop, or they can contain only image tiles ("rectangles") that differ relative to a previous full update. The fundamental assumption of the TightVNC encoding method is that the number of colors used within any given rectangle is usually much less than 16 million. The Tight encoder first tries to identify areas of significant solid color within a rectangle, and if any are found, those "subrectangles" are encoded separately. Solid subrectangles are the most efficient to encode, since they can be represented as simply a fill color and a bounding box. For the remaining non-solid subrectangles, the Tight encoder analyzes each to determine how many colors it would take to represent it. Subrectangles with 2 colors are represented as a bounding box, a 2-entry color table, and a 1-bit-per-pixel bitmap. Subrectangles with 3 to 256 colors are represented as a bounding box, a color table, and an 8-bit-per-pixel bitmap. The Tight encoder also analyzes each subrectangle to determine its relative "smoothness." Subrectangles that contain a large number of unique colors and that are relatively smooth are candidates for "gradient encoding." Gradient encoding predicts the intensity of the next pixel based on the three neighboring pixels and encodes the difference between the actual and predicted intensities. Recent versions of TightVNC also allow the user to specify that "smooth" subrectangles with large numbers of unique

colors should be encoded using JPEG instead of gradient encoding. If a high-color-depth subrectangle does not meet the smoothness criteria for either gradient or JPEG encoding, then it is encoded as a raw (RGB) image. Subrectangles that use raw, indexed color, or gradient encoding are compressed using Zlib prior to transmission.

TightVNC allows the user to specify a "Tight compression level" between 0 and 9. Each Tight compression level maps to the corresponding Zlib compression level, but Tight compression levels also specify certain other aspects of the encoder's behavior. Lower Tight compression levels, for instance, favor splitting the rectangles into smaller subrectangles. The theory behind this is that smaller subrectangles can sometimes be encoded more efficiently. For instance, if a 512-pixel subrectangle containing 2000 colors is sitting next to a 512-pixel subrectangle containing 20 colors, it is more efficient to encode one using indexed color and the other using raw rather than to encode both as a single raw subrectangle. However, JPEG changes this game somewhat, since it is designed to efficiently encode such spatial changes in color depth. Also, using smaller subrectangles increases the total number of them, which increases the protocol overhead, since a header and other metadata (a palette or a JPEG header, for instance) has to be transmitted with each subrectangle. Different Tight compression levels also use different "smoothness thresholds." The smoothness threshold is used to determine whether a subrectangle is a good candidate for gradient or JPEG encoding. Lower Tight compression levels are generally less likely to use gradient or JPEG encoding. In fact, Tight compression levels 0 through 4 have a smoothness threshold of 0 for gradient encoding, meaning that gradient encoding will never be used. Finally, the Tight compression level controls the maximum number of colors that will be used for indexed color encoding. Subrectangles that exceed this number of colors will be encoded using JPEG or gradient (if they are "smooth") or raw (if they are not "smooth".) Lower Tight compression levels favor the use of indexed color encoding, whereas higher Tight compression levels favor the use of JPEG or gradient encoding.

The initial stages of TurboVNC's development involved naively replacing the libjpeg codec in TightVNC with the much faster TurboJPEG codec from VirtualGL. However, this did not produce a significant speedup, because it turned out that the images produced by 3D applications did not generally pass the smoothness test, and thus very few of them were being encoded using JPEG subrectangles. Also, the TightVNC server was using a tremendous amount of CPU time to encode non-JPEG subrectangles, and this effectively hid the speedup from encoding the other subrectangles with a faster JPEG codec.

To make TightVNC behave more like VirtualGL, it was modified such that any subrectangle with two or more colors was encoded using TurboJPEG. This was a somewhat extreme measure, but at the time, TurboVNC was more of a proof of concept than an actual product, so performance was paramount. Thus, the TightVNC encoder was essentially converted into an almost pure JPEG encoder, similar to the one used by VirtualGL. This new encoder was always much faster than the TightVNC encoder (in some cases, by an order of magnitude or more), but until 2008, the compression efficiency of both solutions was never compared. The following sections describe and repeat the analysis that took place in 2008, which led to the re-design of the TurboVNC encoder in version 0.5.

2 Tools and Methodology

2.1 Benchmarks

Constantin Kaplinsky of the TightVNC Project conducted a study in the early days of that project which characterized the performance of various VNC encoding methods when encoding images generated by "typical" application workloads. He used Tim Waugh's RFB Proxy program to capture the output of VNC for each application workload. This output was captured using the VNC 3.3 protocol and Hextile encoding. An additional set of programs (`fbs-dump` and `compare-encodings`) were developed by Constantin to decode the output of RFB Proxy and re-encode it using various VNC encoders (including the Tight encoder used by TightVNC.) This allowed each encoder to be compared in terms of raw CPU time and compression efficiency ("tightness.")

The methodology used in this report is essentially the same as the one used by Kaplinsky, except that the set of 2D application workloads that he used was extended to include the output of "typical" 3D applications displayed using VirtualGL, since these are the types of workloads for which TurboVNC was designed. Additionally, the `compare-encodings` benchmark was expanded to include a high-precision timer, the ability to measure the performance of VNC decoders as well as encoders, and the ability to perform pixel format translation (necessary to accurately model the performance of big endian TurboVNC servers.)

When capturing the new sessions using RFB Proxy, TightVNC 1.3.9 was used as the VNC server and RealVNC 3.3.6 was used as the viewer. Hextile encoding was specified. This guaranteed that the streams captured by RFB Proxy would be compatible with the `fbs-dump` and `compare-encodings` utilities.

The modified versions of these tools, as well as the various encoders used in this study, can be found here:

https://github.com/TurboVNC/vncbenchtools

In all cases, the benchmarks were compiled using 32-bit code with GCC 4.1.2 and maximum optimizations (-O3). 32-bit code was used primarily because it represents a "worst-case" performance for TurboJPEG. The TurboJPEG-enhanced encoders presented in this report will generally perform better on 64-bit machines.

2.2 JPEG Codec

When the original study was conducted in 2008, TurboJPEG/IPP was used as the accelerated JPEG codec in the `compare-encodings` tests, since it was the JPEG codec that TurboVNC was using at the time. TurboVNC has since migrated to TurboJPEG/OSS, which is built using libjpeg-turbo, so TurboJPEG/OSS was used to obtain the data in this report. This did not change any of the conclusions relative to the original study.

2.3 Datasets

Dataset	Color Depth	Source	Description
bugzilla-16	16-bit	Tim Waugh	Using the Bugzilla web site in Netscape Navigator running on the KDE window manager.
compilation-16	16-bit	Conatantin Kaplinsky	Compiling the TightVNC server in a full-screen xterm window running on the IceWM window manager.
bars-16	16-bit	Conatantin Kaplinsky	Manipulating a photo in GIMP running on the WindowMaker window manager with desktop wallpaper enabled.
kde-hearts-16	16-bit	Conatantin Kaplinsky	Manipulating several different file manager windows running on the KDE window manager with wallpaper enabled (coloured_hearts.jpg.) Each window has a different background image.
freshmeat-8	8-bit (BGR233)	Conatantin Kaplinsky	An actual session captured over dial-up in which a person performs various Internet-related functions, including reading mail, reading online news at freshmeat.net, searching for and downloading a file, etc.
slashdot-24	24-bit	Conatantin Kaplinsky	An actual session captured over dial-up in which a person performs various Internet-related functions, including reading mail, reading online news at slashdot.org, generating a change log in CVS, etc.
photos-24	24-bit	Conatantin Kaplinsky	Viewing a photo in GIMP running on the IceWM window manager with a photo used as wallpaper.
kde-hearts-24	24-bit	Conatantin Kaplinsky	Similar to kde-hearts-16 but with a 24-bit color depth.
3dsmax-04-24	24-bit	The VirtualGL Project	The 3D Studio Max viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
catia-02-24	24-bit	The VirtualGL Project	The CATIA viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.

Dataset	Color Depth	Source	Description
ensight-03-24	24-bit	The VirtualGL Project	The EnSight viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
light-08-24	24-bit	The VirtualGL Project	The Lightscape viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
maya-02-24	24-bit	The VirtualGL Project	The Maya viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
proe-04-24	24-bit	The VirtualGL Project	The Pro/ENGINEER viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
sw-01-24	24-bit	The VirtualGL Project	The SolidWorks viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
tcvis-01-24	24-bit	The VirtualGL Project	The UGS TeamCenter Visualization viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
ugnx-01-24	24-bit	The VirtualGL Project	The UGS NX viewset from Viewperf 9.0 running in VirtualGL with no frame spoiling and a VNC server geometry of 1280 x 1024 pixels. This session was captured over a cable modem connection.
glxspheres-24	24-bit	The VirtualGL Project	The canonical GLX Spheres application from VirtualGL running in full-screen mode in VirtualGL with no frame spoiling. A VNC server geometry of 1240 x 900 pixels was used.

Dataset	Color Depth	Source	Description
googleearth-24	24-bit	The VirtualGL Project	Launch Google Earth in VirtualGL, locate and zoom in on the Coors Brewery in Golden, CO, then perform various operations on the display (including panning, zooming, and tilting upward to reveal the terrain.) A VNC server geometry of 1280 x 1024 pixels was used.
q3demo-24	24-bit	The VirtualGL Project	The first few seconds of the Quake 3 demo running in VirtualGL. A VNC server geometry of 1280 x 1024 pixels was used.

A copy of these datasets is available upon request.

2.4 Test System

For the purposes of this report, a modern workstation available at the time of this writing was used to duplicate the results observed in 2008, since the data obtained in the previous study is no longer available:

- Dell Precision T3500 with quad-core 2.8 GHz Intel Xeon W3530
- 4 GB memory
- CentOS Enterprise Linux 5.6 (64-bit)

2.5 Metrics

Two metrics are used in this report: *compression ratio* and *compression time*.

The compression ratio represents how efficiently the images in the dataset could be compressed. It is defined as:

$$\text{compression ratio} = \frac{\text{uncompressed dataset size}}{\text{compressed dataset size}}$$

The compression time is simply the total number of seconds required to compress the dataset.

The reported compression times reflect an average of two runs of the `compare-encodings` benchmark.

3 Results

3.1 Duplicating the Results of the Original TightVNC Study

First, the modified `compare-encodings` benchmark was used along with the TightVNC 1.1 encoder to duplicate the results of the earlier study conducted by Constantin Kaplinsky. In all cases, identical compression ratios were achieved, and the relative differences in compression times were similar to those observed in the earlier study. The absolute compression times were somewhat faster for our study, however, given that a 350 MHz Pentium II machine was used to conduct Kaplinsky's original study.

3.2 Establishing a Baseline

Once the results of the original study were duplicated, then the unmodified TightVNC 1.3.9 encoder was inserted into the `compare-encodings` benchmark and used to obtain a baseline level of performance. The TightVNC encoder was configured to use gradient encoding (no JPEG) with a compression level of 9. The TurboVNC 0.4 encoder, configured with perceptually lossless JPEG encoding (no chrominance subsampling, JPEG quality = 95) was used as an additional baseline. These two baselines would serve as performance targets for compression ratio and compression time, respectively.

Baseline Tests
TightVNC 1.3.9 Encoder, Gradient Encoding Enabled, Compression Level = 9
TurboVNC 0.4 Encoder, No Chrominance Subsampling, JPEG Quality = 95

Dataset	TightVNC 1.3.9 Compression Ratio	TightVNC 1.3.9 Compression Time (s)	TurboVNC 0.4 Compression Ratio	TurboVNC 0.4 Compression Time (s)
bugzilla-16	66.75	1.74		
compilation-16	274.2	1.65		
bars-16	10.14	4.57		
kde-hearts-16	11.72	1.67		
freshmeat-8	51.79	8.42		
slashdot-24	88.38	15.2	7.812	2.69
photos-24	4.255	3.57	10.42	0.355
kde-hearts-24	19.05	2.05	4.839	0.791
3dsmax-04-24	16.43	82.2	8.011	4.67
catia-02-24	38.08	79.5	9.825	5.88
ensight-03-24	19.24	103	4.426	3.48
light-08-24	11.06	21.3	7.475	1.06
maya-02-24	75.64	54.6	18.08	3.29

Dataset	TightVNC 1.3.9 Compression Ratio	TightVNC 1.3.9 Compression Time (s)	TurboVNC 0.4 Compression Ratio	TurboVNC 0.4 Compression Time (s)
proe-04-24	34.71	17.9	11.51	1.72
sw-01-24	68.83	26.5	25.41	1.67
tcvis-01-24	28.38	10.7	14.88	1.06
ugnx-01-24	36.58	48.0	25.87	6.16
glxspheres-24	12.64	6.61	18.93	0.407
googleearth-24	3.038	78.2	8.313	3.58
q3demo-24	4.188	29.7	10.36	0.897

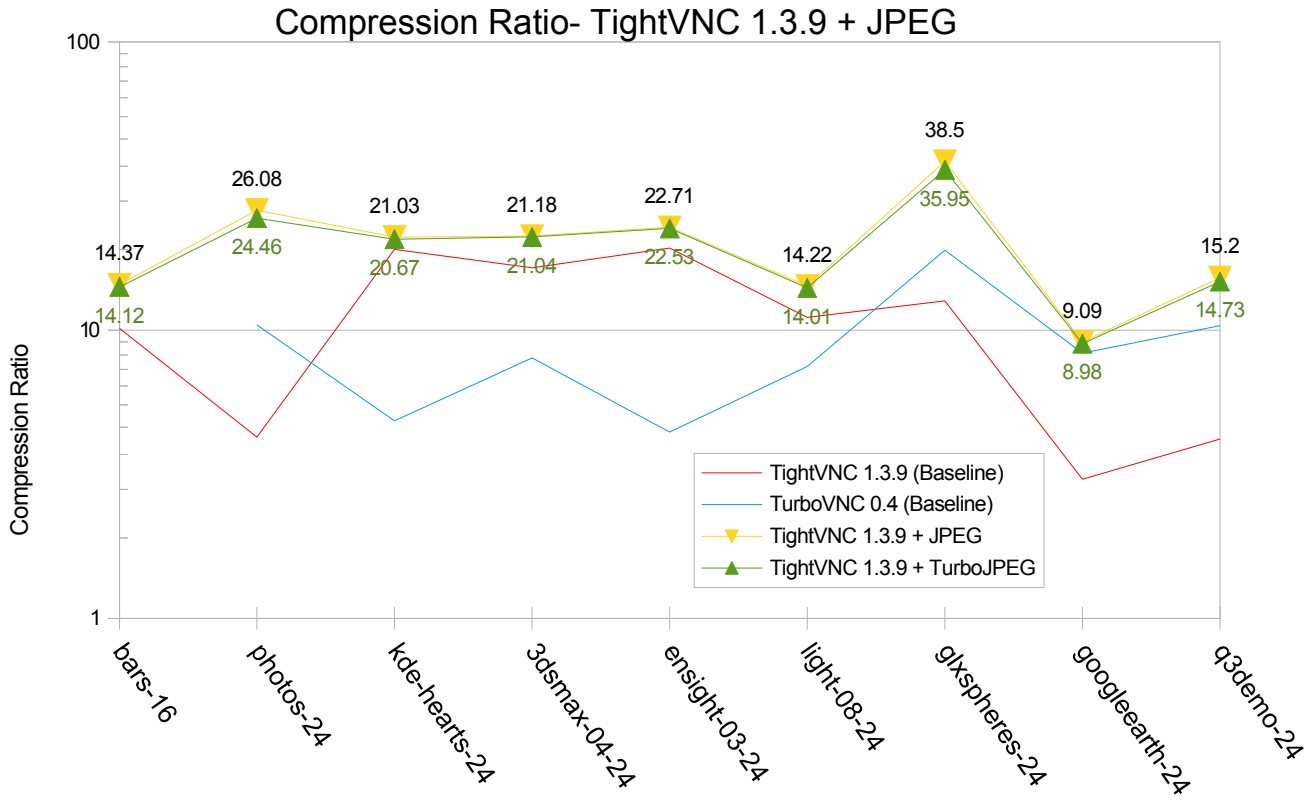
The first thing to note is that TightVNC and TurboVNC were appropriately named. TurboVNC 0.4 was faster in every case than TightVNC. In fact, on most of the datasets, it was faster by an order of magnitude or more. However, TightVNC definitely won hands down in terms of compression ratio. The only datasets that compressed better with TurboVNC 0.4 were Quake 3 Demo, Google Earth, GLXspheres, and Photos. These datasets contain large numbers of unique colors and thus could not benefit from TightVNC's indexed color encoding optimizations. Most of the other 3D workloads compressed much better (often several times better) using TightVNC. Prior to TurboVNC 0.5, it was well known that TurboVNC was faster than TightVNC, but it was assumed that this speedup did not come at the expense of very much compression efficiency. For the types of applications that TurboVNC was initially designed to support (applications such as Google Earth, for instance), this was a valid assumption. For CAD applications, however, the assumption was not valid.

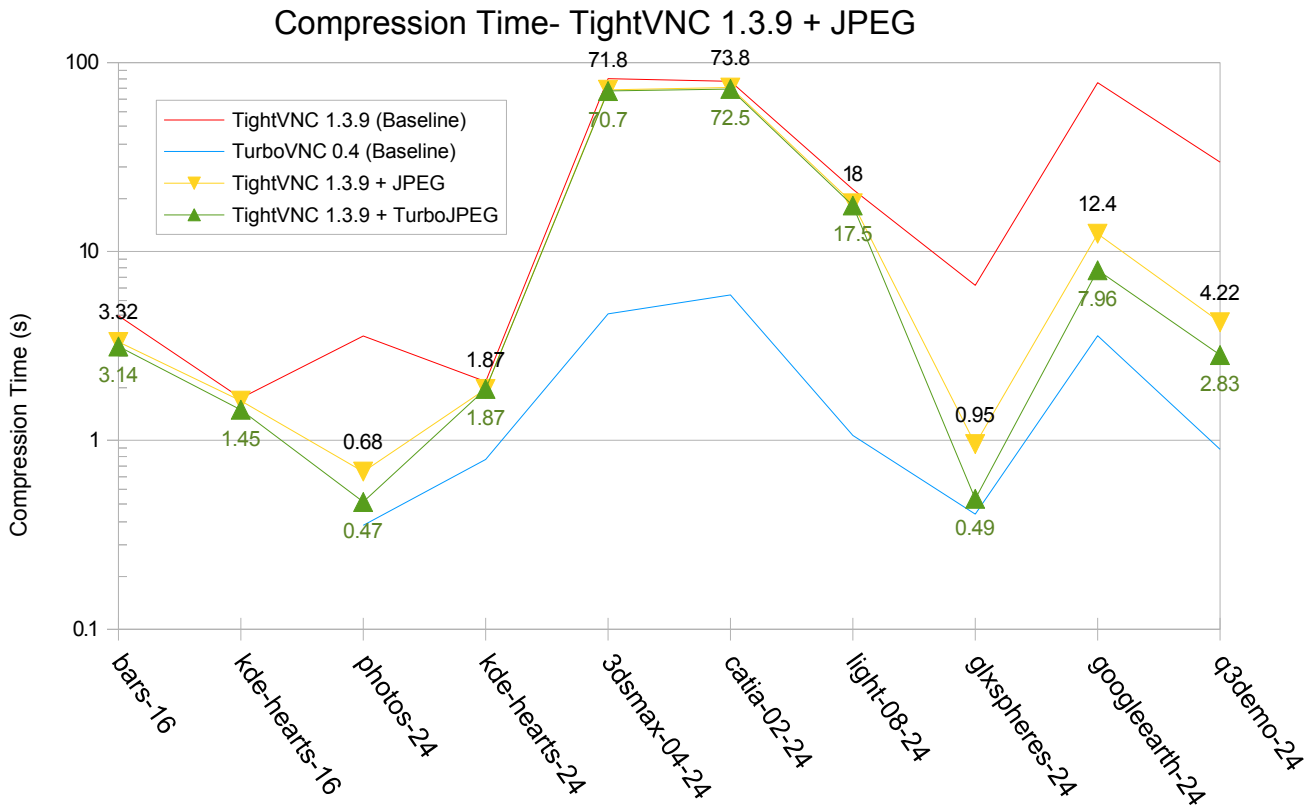
Note that, prior to version 0.5, TurboVNC did not support color depths less than 24 for its virtual screen, which is why its baseline does not contain results for the 16-bit and 8-bit datasets. Those datasets are designed to model the performance of VNC when the virtual desktop is 8-bit or 16-bit, and this situation is rarely if ever encountered in TurboVNC. Thus, those datasets are included mainly for reference.

The baseline gave us a good indication of where we currently stood. TightVNC was definitely the "tightest" form of compression, but TurboVNC was definitely the fastest. The question then became whether the two could be reconciled. The goal was to make TurboVNC "tighter" without sacrificing any of its performance.

3.3 Adding JPEG Encoding to TightVNC

The baseline TightVNC configuration was modified to use JPEG encoding with a quality level of 9 rather than gradient encoding. Then, the libjpeg codec was replaced with TurboJPEG, using the equivalent JPEG quality (4:2:2 chrominance subsampling, JPEG quality = 80.) The graphs below compare the results with the two baselines. Values are shown only for those datasets that deviated significantly (> 5%) from the TightVNC baseline.


Compression Ratio- TightVNC 1.3.9 + JPEG
Compression Ratio
100
10
1
14.37
14.12
26.08
24.46
21.03
20.67
21.18
21.04
22.71
22.53
14.22
14.01
38.5
35.95
9.09
8.98
15.2
14.73
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + JPEG
TightVNC 1.3.9 + TurboJPEG
bars-16
photos-24
kde-hearts-24
3dsmax-04-24
ensight-03-24
light-08-24
glxspheres-24
googleearth-24
q3demo-24


Compression Time- TightVNC 1.3.9 + JPEG
Compression Time (s)
100
10
1
0.1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + JPEG
TightVNC 1.3.9 + TurboJPEG
3.32
3.14
1.45
0.68
0.47
1.87
1.87
71.8
70.7
73.8
72.5
18
17.5
0.95
0.49
12.4
7.96
4.22
2.83
bars-16
kde-hearts-16
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
light-08-24
glxspheres-24
googleearth-24
q3demo-24

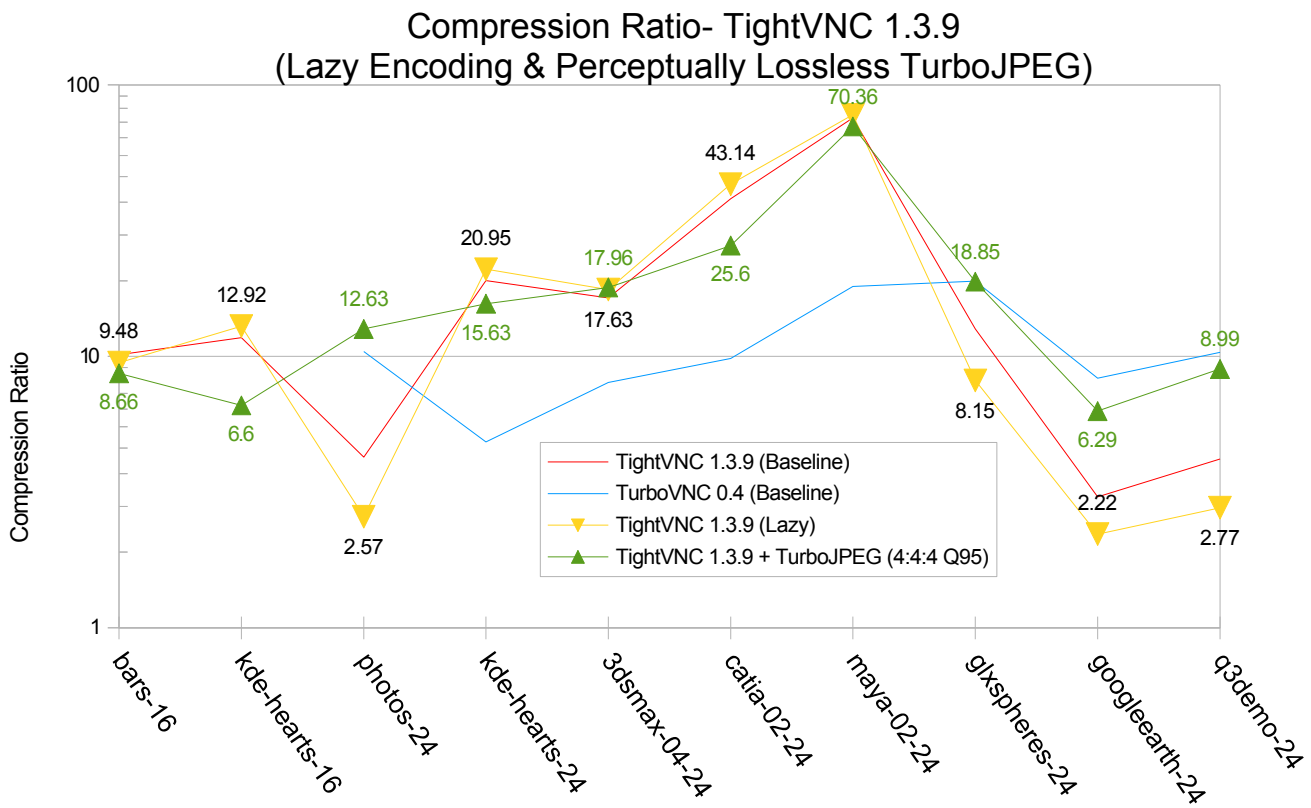
JPEG performed as fast as or faster than gradient encoding and compressed better than gradient encoding in all cases. However, the only tests that realized a significant speedup from JPEG encoding were the same four tests (Quake 3 Demo, Google Earth, GLXspheres, and Photos) that showed themselves to be good candidates for JPEG compression in the last section.

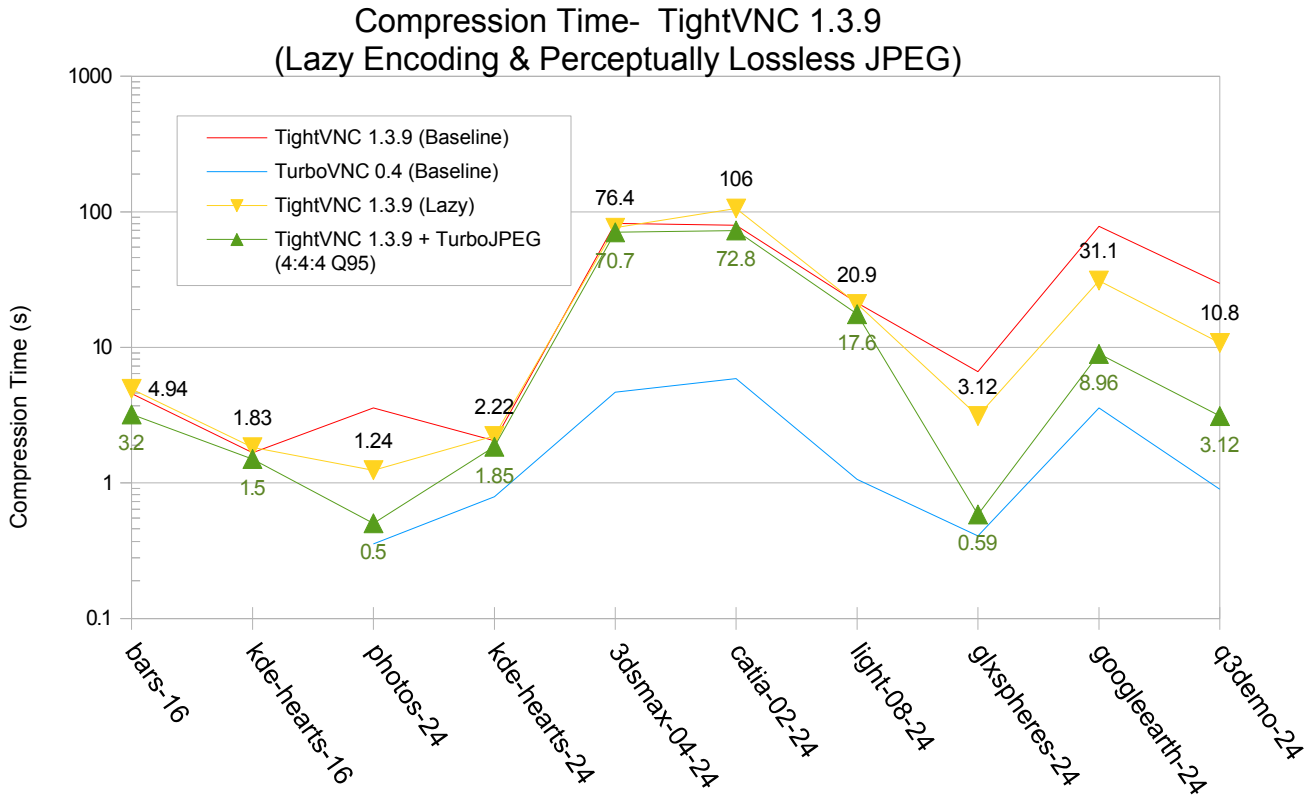
Also note that, while the Quake 3 Demo, Google Earth, GLXspheres, and Photos benchmarks were significantly faster with TurboJPEG than with libjpeg, none of the other tests realized a significant speedup from the use of TurboJPEG. Note also that, except in the case of GLXspheres, none of the 3D datasets even came close to the performance of TurboVNC 0.4.

This mirrored the results of experiments conducted in the early days of TurboVNC. JPEG was only being used regularly on a few datasets, so this limited the speedup that could be realized by replacing libjpeg with TurboJPEG. Since JPEG is not the most efficient form of compression for subrectangles with low color depths, we needed to figure out how to accelerate indexed color encoding for such subrectangles, but we also needed to figure out how to better balance the mix of indexed color and JPEG subrectangles.

3.4 How Useful is Gradient Encoding Anyway?

The baseline configuration was modified to use “lazy tight” encoding, which replaces gradient-encoded rectangles with raw-encoded rectangles. Values are shown only for those datasets that deviated significantly (> 5%) from the TightVNC baseline.


Compression Ratio- TightVNC 1.3.9
(Lazy Encoding & Perceptually Lossless TurboJPEG)
Compression Ratio
100
10
1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 (Lazy)
TightVNC 1.3.9 + TurboJPEG (4:4:4 Q95)
9.48
8.66
12.92
6.6
2.57
12.63
20.95
15.63
17.96
17.63
43.14
25.6
70.36
18.85
8.15
6.29
2.22
8.99
2.77
bars-16
kde-hearts-16
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
maya-02-24
glxspheres-24
googleearth-24
q3demo-24


Compression Time- TightVNC 1.3.9
(Lazy Encoding & Perceptually Lossless JPEG)
Compression Time (s)
1000
100
10
1
0.1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 (Lazy)
TightVNC 1.3.9 + TurboJPEG
(4:4:4 Q95)
4.94
3.2
1.83
1.5
1.24
0.5
2.22
1.85
76.4
70.7
106
72.8
20.9
17.6
3.12
0.59
31.1
8.96
10.8
3.12
bars-16
kde-hearts-16
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
light-08-24
glxspheres-24
googleearth-24
q3demo-24

The results were mixed. The Quake 3 Demo, Google Earth, GLXspheres, and Photos datasets compressed somewhat worse but also much faster without gradient encoding. Most of the other datasets actually compressed a bit better without gradient encoding. In only one case (CATIA) did turning off gradient encoding cause a significant performance hit.

The previous section showed that, in terms of performance, JPEG encoding wins hands down over gradient encoding. However, gradient encoding has the advantage of being mathematically lossless, so comparing "medium quality" JPEG with gradient encoding (as we did in the previous section) was not entirely fair. A more fair comparison would be to compare gradient encoding and TurboJPEG encoding with "perceptually lossless" quality. To that end, the TurboJPEG-enhanced Tight 1.3.9 encoder from the previous section was further modified to use perceptually lossless JPEG encoding, and those results appear in green in the above graphs.

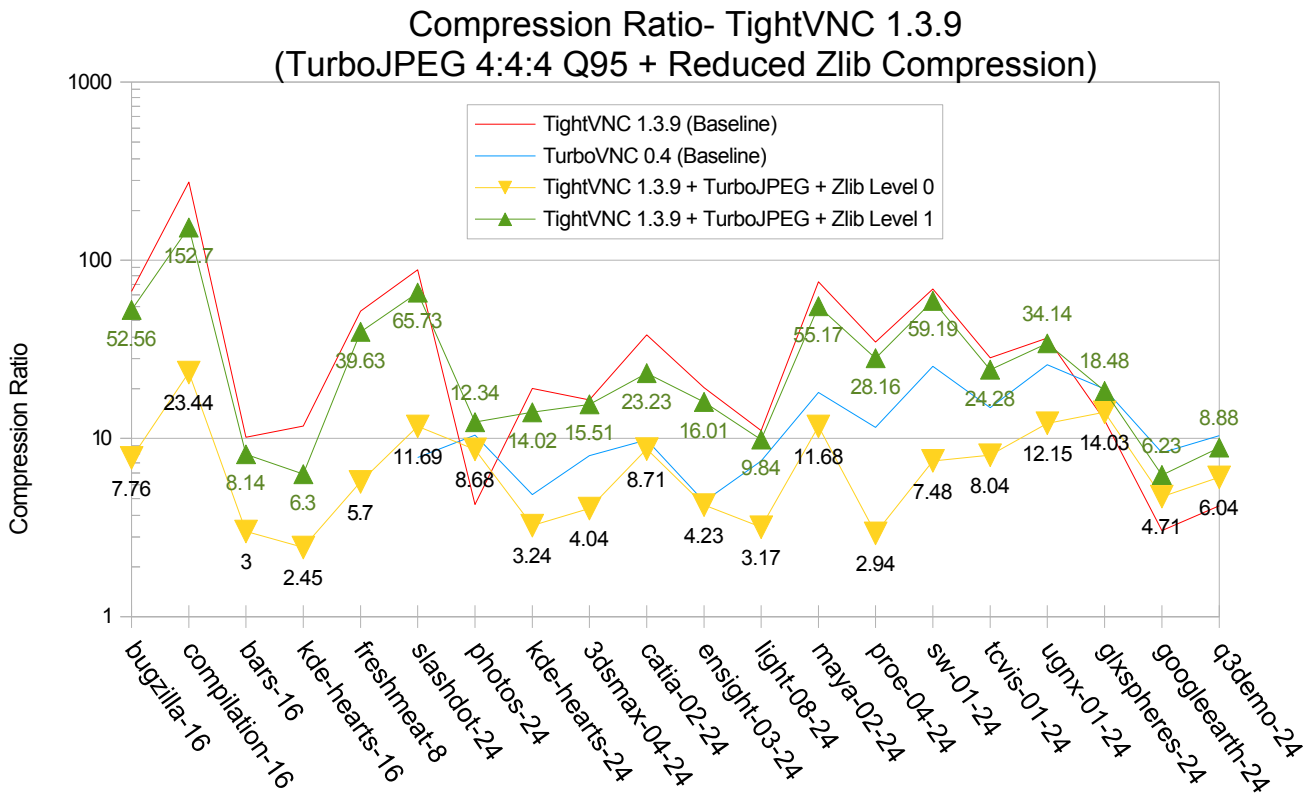
Perceptually lossless JPEG was always faster than gradient encoding, and with the exception of CATIA, all of the 3D datasets compressed as well or better with perceptually lossless JPEG than with gradient encoding.

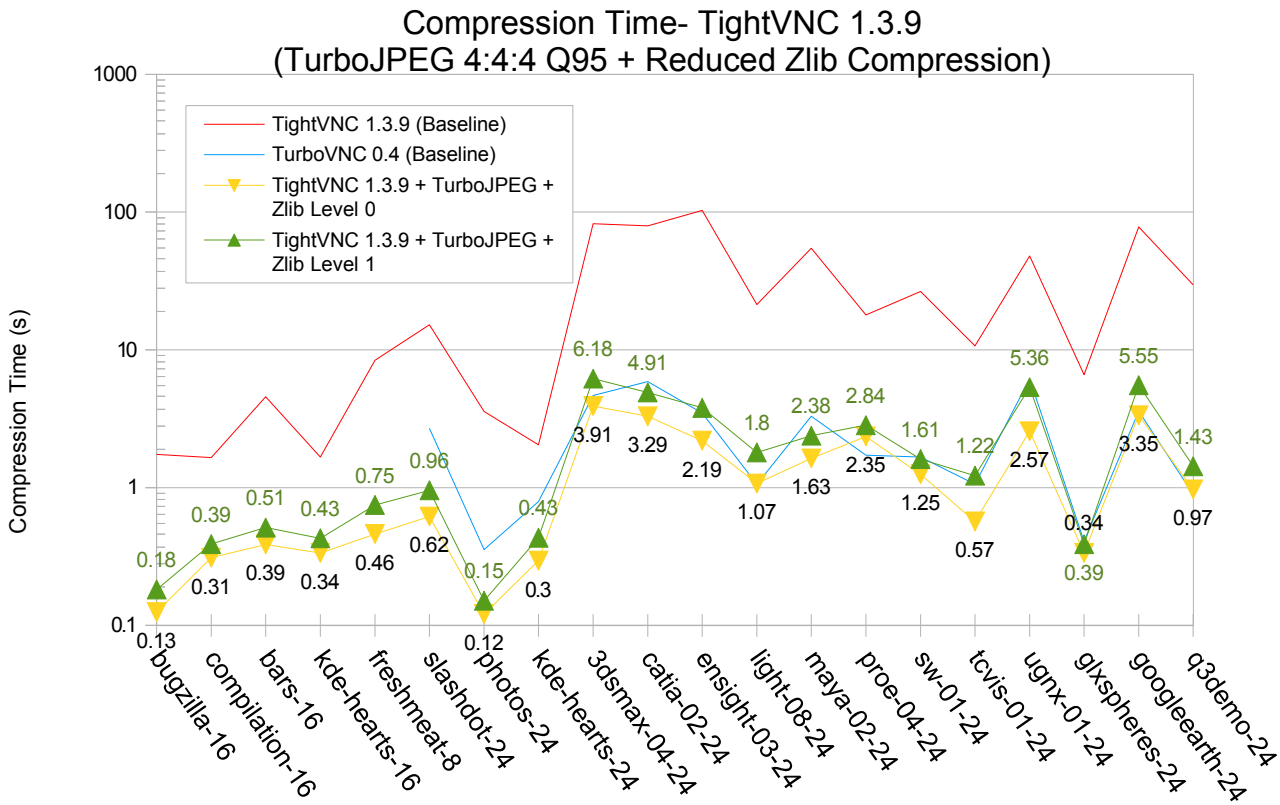
3.5 Reducing the Level of Zlib Encoding

With the exception of GLXspheres, the other datasets were still falling considerably short of the TurboVNC 0.4 performance target. One suspected reason for this was Zlib encoding. TightVNC Compression Level 9 uses Zlib with compression level 9 to compress indexed color and raw subrectangles and Zlib with compression level 6 to compress gradient-encoded subrectangles.

Reducing the compression level to 0 or 1 reduces the Zlib compression level (and, consequently, the CPU usage) accordingly. However, TightVNC compression levels 0 and 1 specify very small maximum subrectangle sizes (512 pixels and 2048 pixels, respectively.) As discussed earlier, the theory behind using smaller subrectangles is that it can allow for more efficient encoding in cases where areas of high color depth are adjacent to areas of low color depth. However, if the subrectangle is too small, this advantage can easily be outweighed by the overhead of sending hundreds or even thousands of subrectangles to update a full screen image. Experiments showed that the smaller subrectangle sizes used by Tight compression levels 0 and 1 generated a tremendous amount of network and CPU overhead. Particularly with JPEG, it is important to use larger subrectangles to hide the overhead of the JPEG header. For this reason, TurboVNC historically used the subrectangle size parameters associated with Tight Compression Level 9. With indexed color encoding, it is also important that the subrectangle size be large enough to hide the overhead of the palette.

Bearing this in mind, we modified the TurboJPEG-enhanced TightVNC 1.3.9 encoder such that Zlib compression level 0 and 1 were used for raw and indexed color encoding. However, the other Tight encoding parameters were kept at the equivalent of Tight Compression Level 9.


Compression Ratio- TightVNC 1.3.9
(TurboJPEG 4:4:4 Q95 + Reduced Zlib Compression)
Compression Ratio
1000
100
10
1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + TurboJPEG + Zlib Level 0
TightVNC 1.3.9 + TurboJPEG + Zlib Level 1
52.56
152.7
7.76
23.44
8.14
3
6.3
2.45
39.63
5.7
65.73
11.69
12.34
8.68
14.02
3.24
15.51
4.04
23.23
8.71
16.01
4.23
9.84
3.17
55.17
11.68
28.16
2.94
59.19
7.48
24.28
8.04
34.14
12.15
18.48
14.03
6.23
4.71
8.88
6.04
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugnx-01-24
glxspheres-24
googleearth-24
q3demo-24

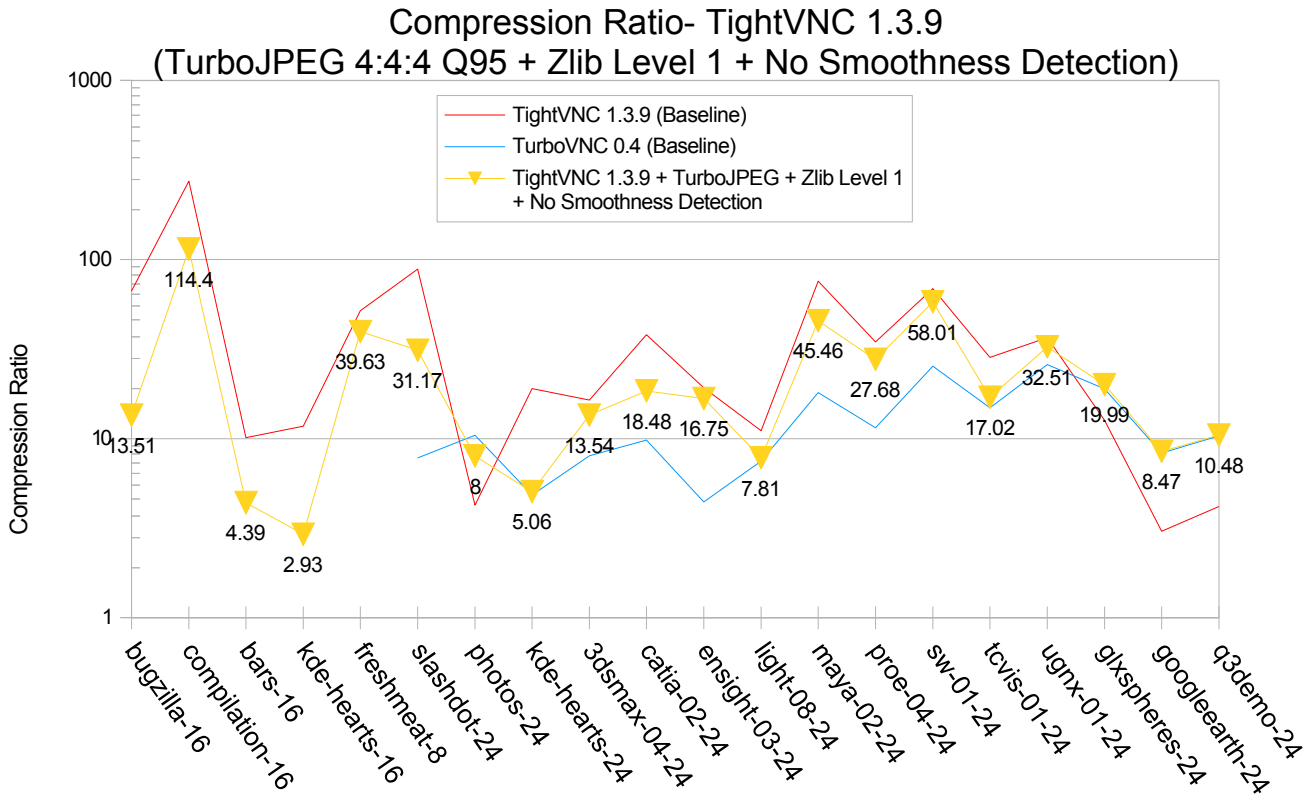

Compression Time- TightVNC 1.3.9
(TurboJPEG 4:4:4 Q95 + Reduced Zlib Compression)
Compression Time (s)
1000
100
10
1
0.1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + TurboJPEG + Zlib Level 0
TightVNC 1.3.9 + TurboJPEG + Zlib Level 1
0.18
0.39
0.51
0.43
0.75
0.96
0.43
6.18
4.91
1.8
2.38
2.84
1.61
1.22
5.36
0.34
5.55
1.43
0.13
0.31
0.39
0.34
0.46
0.62
0.15
0.12
0.3
3.91
3.29
2.19
1.07
1.63
2.35
1.25
0.57
2.57
0.39
3.35
0.97
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugnx-01-24
glxspheres-24
googleearth-24
q3demo-24

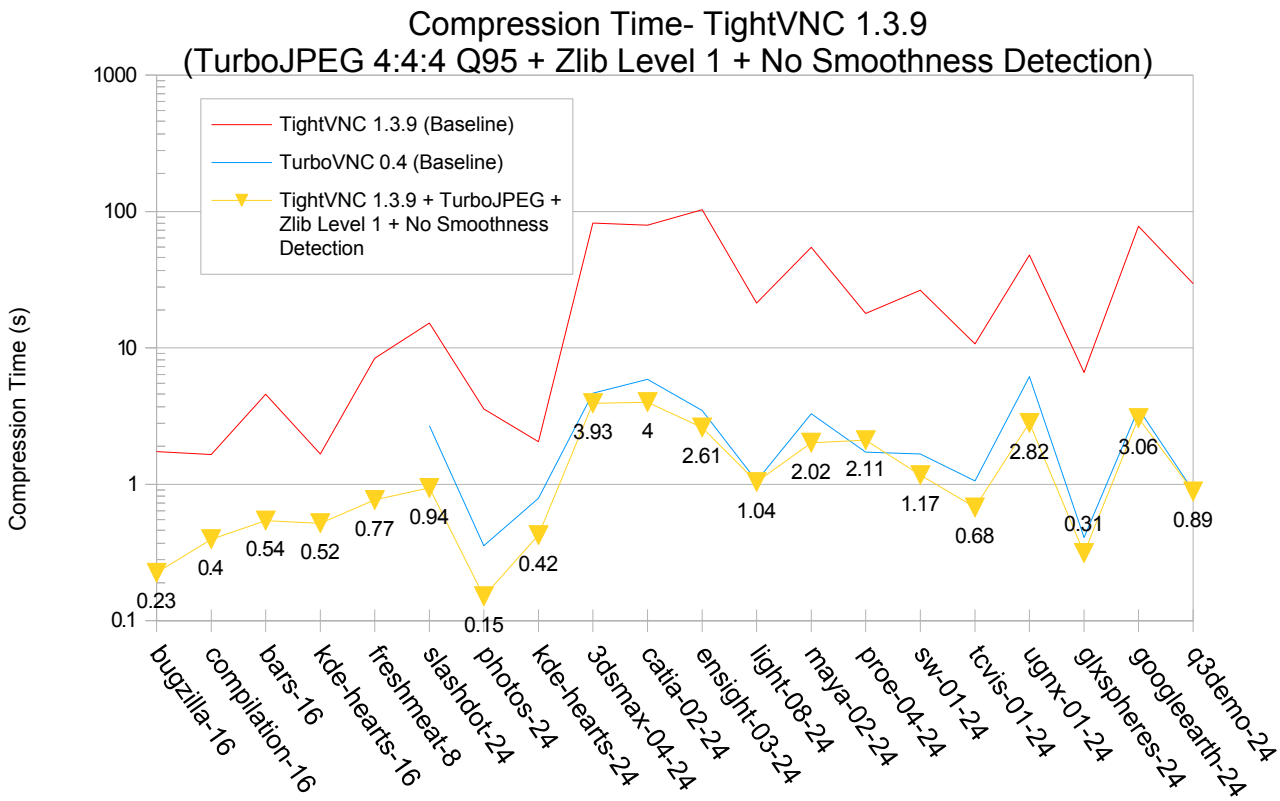
With no Zlib compression, most of the datasets exceeded the TurboVNC 0.4 performance target. However, the Quake 3 Demo and Pro/ENGINEER datasets were still slower (by 8% and 27%, respectively.) Also, most of the datasets compressed much worse than with TurboVNC 0.4.

Adding in a nominal amount of Zlib compression improved the compression considerably, to the point at which, with the exception of CATIA, none of the 3D datasets compressed more than 30% worse than the TightVNC baseline. However, Google Earth and Quake 3 Demo still did not compress as well as they did under TurboVNC 0.4, and now only about half of the datasets met the TurboVNC 0.4 performance target. We were on the right track, but further tweaking needed to be done.

3.6 Eliminating Smoothness Detection

Building upon the previous results, the TurboJPEG-enhanced TightVNC 1.3.9 encoder was further modified so it did not perform smoothness detection. Thus, all subrectangles that had enough unique colors to exceed the palette threshold were encoded using TurboJPEG.


Compression Ratio- TightVNC 1.3.9
(TurboJPEG 4:4:4 Q95 + Zlib Level 1 + No Smoothness Detection)
Compression Ratio
1000
100
10
1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + TurboJPEG + Zlib Level 1 + No Smoothness Detection
13.51
114.4
4.39
2.93
39.63
31.17
8
5.06
13.54
18.48
16.75
7.81
45.46
27.68
58.01
17.02
32.51
19.99
8.47
10.48
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugnx-01-24
glxspheres-24
googleearth-24
q3demo-24


Compression Time- TightVNC 1.3.9
(TurboJPEG 4:4:4 Q95 + Zlib Level 1 + No Smoothness Detection)
Compression Time (s)
1000
100
10
1
0.1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + TurboJPEG + Zlib Level 1 + No Smoothness Detection
0.23
0.4
0.54
0.52
0.77
0.94
0.15
0.42
3.93
4
2.61
1.04
2.02
2.11
1.17
0.68
2.82
0.31
3.06
0.89
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugnx-01-24
glxspheres-24
googleearth-24
q3demo-24

This was more like it. Almost all of the datasets compressed significantly faster and significantly better than with TurboVNC 0.4. However, the Photos dataset compressed 23% worse than with TurboVNC 0.4, and the Pro/ENGINEER dataset was slower by about 19%. Since the Photos dataset represents an important 2D workload, a regression on that dataset was unacceptable.

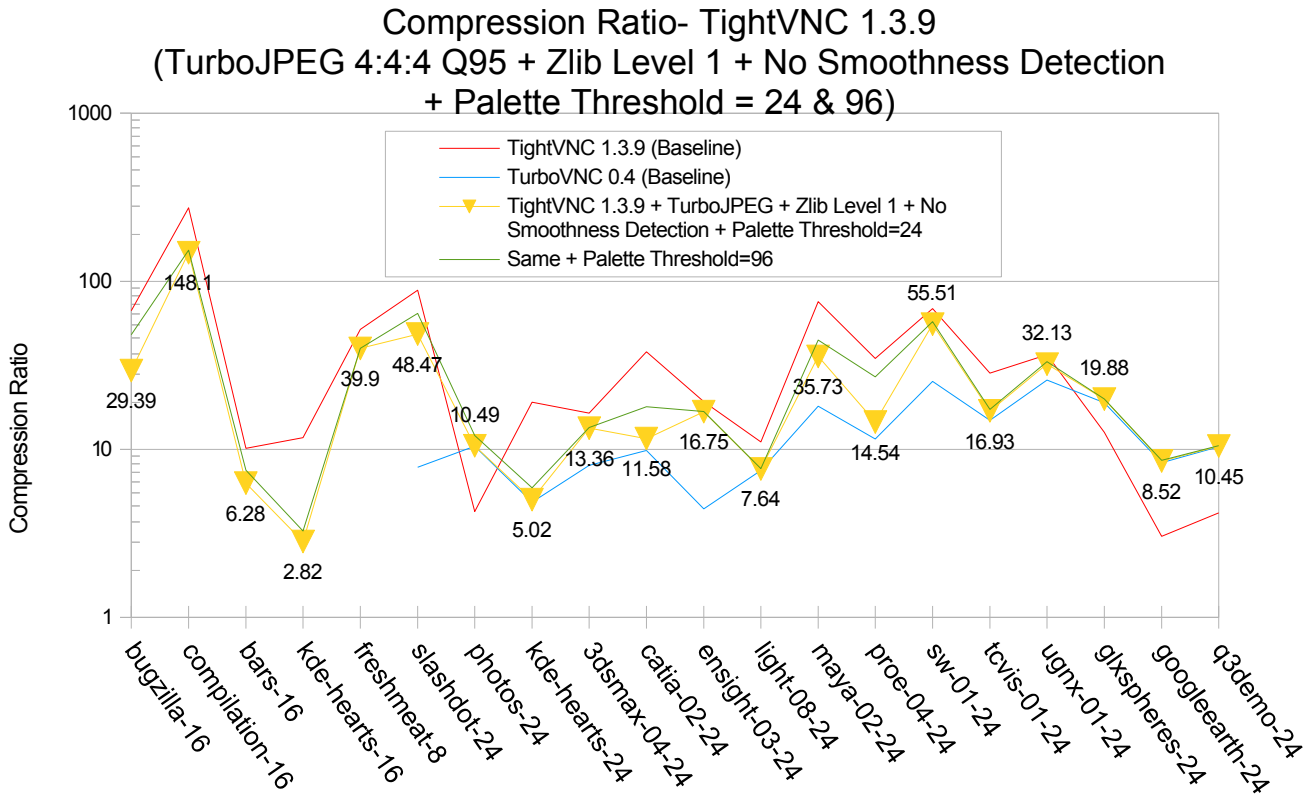
3.7 Adjusting the Palette Threshold

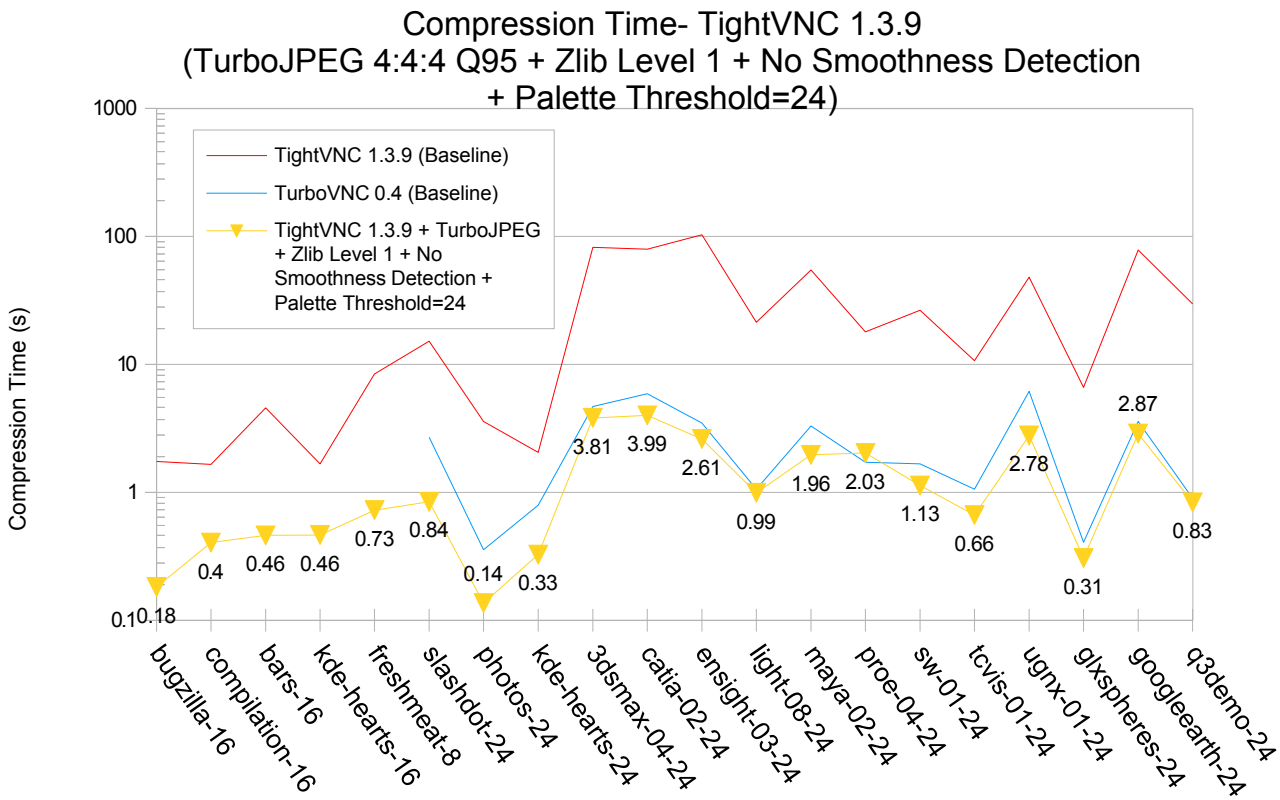
Each compression level in TightVNC has a corresponding "maximum colors divisor." The number of pixels in a given subrectangle is divided by this number to get the "palette threshold", which determines the maximum number of colors to encode using indexed color encoding. Any subrectangle with more colors than this will be encoded using JPEG encoding (since we disabled the smoothness detection routines in the previous section.)

The default max. colors divisor for TightVNC Compression Level 9 is 96. That means that any subrectangle whose number of unique colors is less than its pixel count divided by 96 will be encoded using indexed color encoding. Effectively, the palette threshold will be 256 for any subrectangle of 24,576 pixels or larger, and since Compression Level 9 also specifies a maximum subrectangle size of 65,536 pixels, subrectangles of 24k would not be uncommon. Subrectangles of this size with any significant color depth would be more quickly and efficiently compressed using TurboJPEG, as the previous sections showed. What we really want to do is use indexed color encoding only on subrectangles that JPEG compresses inefficiently. In general, that means we want to use indexed color encoding on subrectangles with very sharp spatial changes in luminance or chrominance (for instance, wireframe images.) As it turns out, such subrectangles also tend to have very low color depths, particularly since we've already eliminated any significant areas of solid color that may have surrounded them.

Thus, lowering the palette threshold does two things: (1) it acts as a quick and dirty smoothness threshold, since, at least as it applies to 3D applications, smooth images are likely to have more colors than sharp images, and (2) it effectively limits the size of indexed color subrectangles, since larger subrectangles will tend to have more colors and thus will be less likely to fall below the palette threshold.

The palette threshold can be thought of as a performance dial. Turning it to the left increases performance at the expense of compression ratio, and turning it to the right increases compression ratio at the expense of performance. Without going into the gory details, various palette thresholds were tried, and values in the range of 24 to 96 proved to have the best balance. Values lower than 24 caused the compression ratio to, in some cases, fall below the TurboVNC 0.4 baseline. Values higher than 96 caused the performance for multiple datasets to fall short of the TurboVNC 0.4 baseline.


Compression Ratio- TightVNC 1.3.9
(TurboJPEG 4:4:4 Q95 + Zlib Level 1 + No Smoothness Detection
+ Palette Threshold = 24 & 96)
Compression Ratio
1000
100
10
1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + TurboJPEG + Zlib Level 1 + No Smoothness Detection + Palette Threshold=24
Same + Palette Threshold=96
29.39
148.1
6.28
2.82
39.9
48.47
10.49
5.02
13.36
11.58
16.75
7.64
35.73
14.54
55.51
16.93
32.13
19.88
8.52
10.45
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugnx-01-24
glxspheres-24
googleearth-24
q3demo-24


Compression Time- TightVNC 1.3.9
(TurboJPEG 4:4:4 Q95 + Zlib Level 1 + No Smoothness Detection
+ Palette Threshold=24)
Compression Time (s)
1000
100
10
1
0.1
TightVNC 1.3.9 (Baseline)
TurboVNC 0.4 (Baseline)
TightVNC 1.3.9 + TurboJPEG + Zlib Level 1 + No Smoothness Detection + Palette Threshold=24
0.18
0.4
0.46
0.46
0.73
0.84
0.14
0.33
3.81
3.99
2.61
0.99
1.96
2.03
1.13
0.66
2.78
0.31
2.87
0.83
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugnx-01-24
glxspheres-24
googleearth-24
q3demo-24

There ain't no such thing as a free lunch, but by balancing the threshold between indexed color and JPEG encoding, we significantly improved both the performance and compression ratio of most of the 2D datasets relative to the previous test, at the expense of some compression ratio loss on a few of the 3D datasets (Pro/E, Maya, and CATIA.) In all cases, our compression ratio was now better than the TurboVNC 0.4 baseline, and, with the exception of the Pro/E dataset, the performance was also better in all cases than the TurboVNC 0.4 baseline. A 16% drop in performance relative to TurboVNC 0.4 was the best we could do with Pro/E. This performance gap is, however, easily mitigated by using the 64-bit version of TurboVNC (available with TurboVNC 0.6 or later) or by simply lowering the JPEG quality.

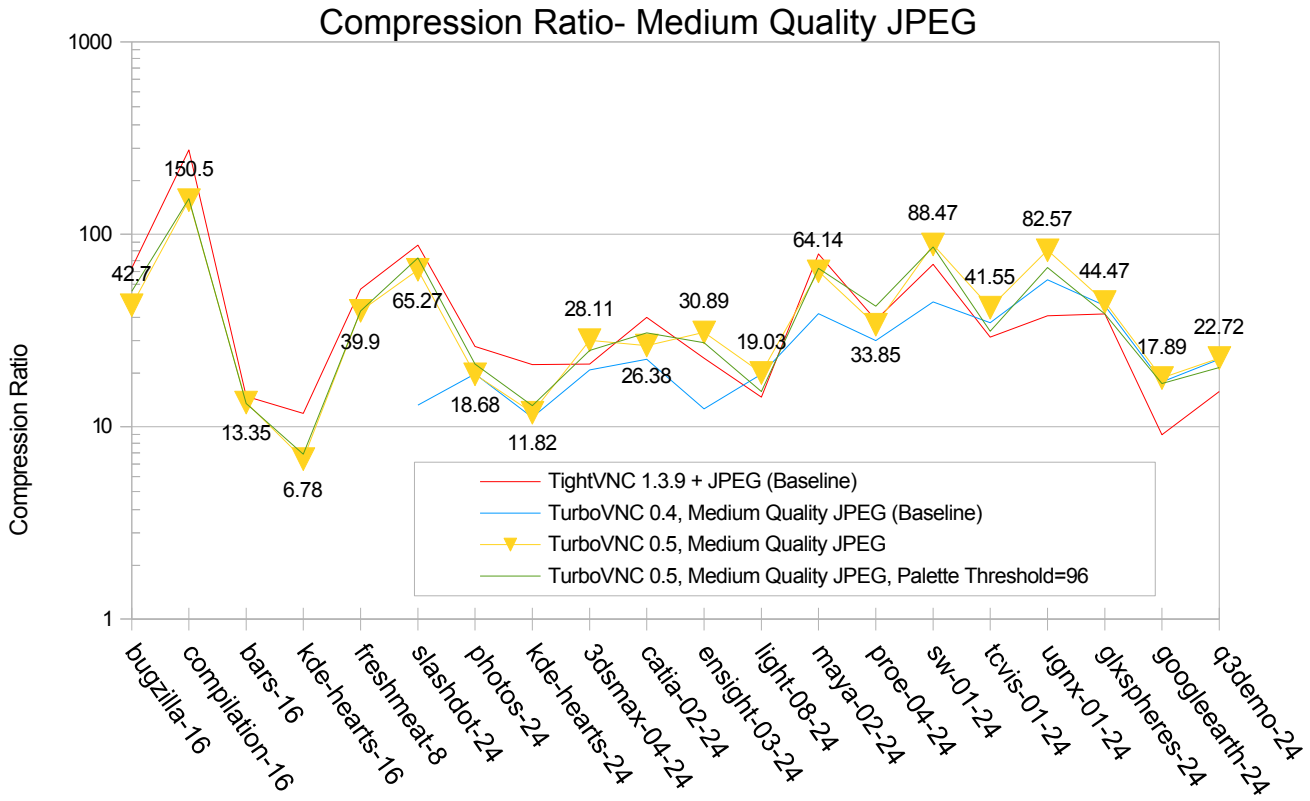
The TurboJPEG-enhanced TightVNC 1.3.9 encoder with no smoothness detection and a palette threshold of 24, plus a few additional optimizations to the palette fill routines, is what ultimately became the TurboVNC 0.5 encoder. The encoding method represented in yellow in the graphs above became the "Tight + Perceptually Lossless JPEG" encoding method.

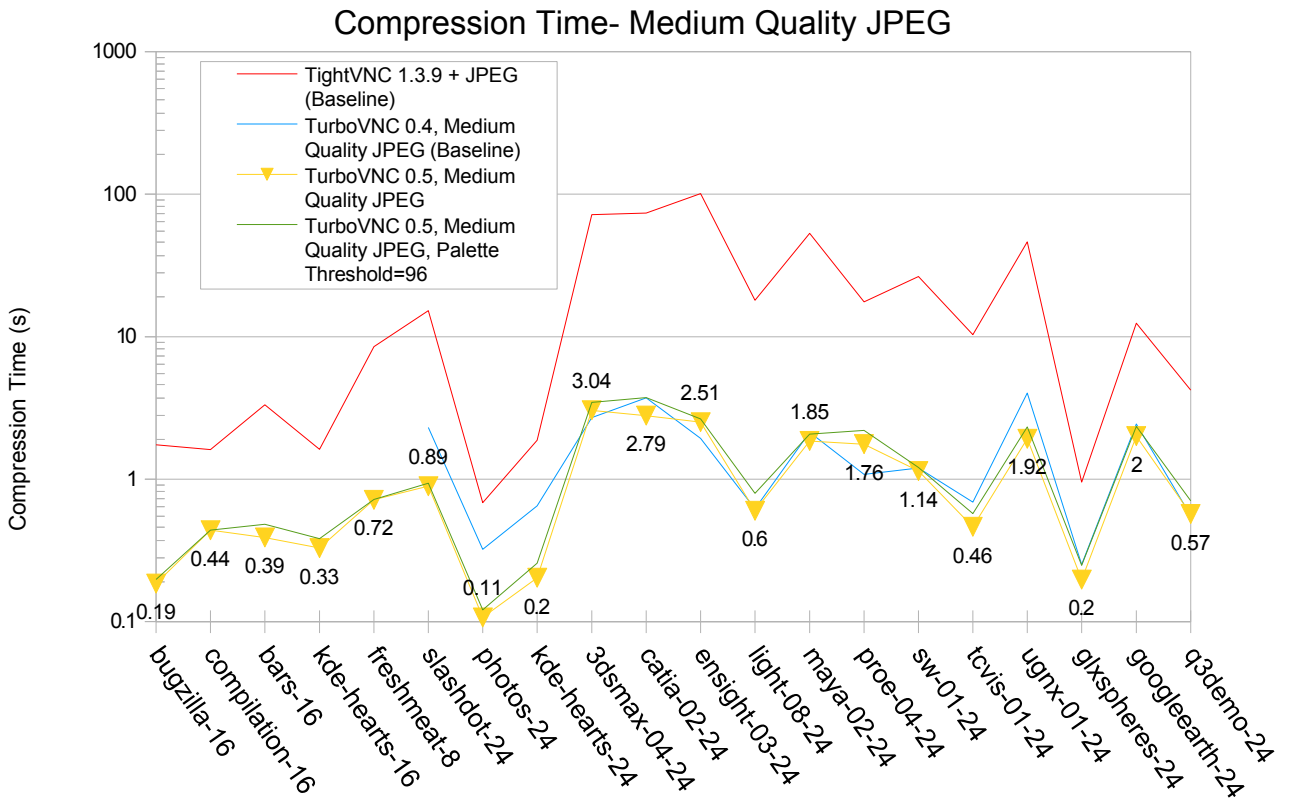
One might conclude from the above data that a palette threshold of 96 is a better choice. On the x86 workstation used for this study, setting the palette threshold to 96 improved the compression ratio significantly for several of the datasets (Pro/E, Maya, and CATIA, and almost all of the 2D datasets), and in no case did a palette threshold of 96 perform more than 6% slower than a palette threshold of 24. However, there were additional factors that went into the choice of a palette threshold of 24. The original study also encompassed SPARC platforms, on which it was typically necessary to convert the palettes from big endian to little endian prior to transmission. This additional computational overhead made it desirable to reduce the use of indexed color encoding as much as possible.

The next section describes other factors that led to the choice of 24 as a palette threshold.

3.8 Medium Quality JPEG Comparisons

Using perceptually lossless JPEG does not tell the whole story. As the JPEG quality is dialed down, both the compression ratio and compression time for JPEG subrectangles improve significantly, whereas the performance of indexed color subrectangles remains the same. Thus, it was desirable to repeat the analysis from the previous section using "medium quality JPEG" (4:2:2 subsampling, JPEG quality = 80) in order to verify that the "golden" mix of indexed color and JPEG subrectangles still produced optimal performance. For this test, the TightVNC 1.3.9 encoder with JPEG enabled (see Section 3.3) was used as a compression ratio baseline, and the TurboVNC 0.4 encoder with medium-quality JPEG was used as a performance baseline.


Compression Ratio- Medium Quality JPEG
Compression Ratio
1000
100
10
1
42.7
150.5
13.35
6.78
39.9
65.27
18.68
11.82
28.11
26.38
30.89
19.03
64.14
33.85
88.47
41.55
82.57
44.47
17.89
22.72
TightVNC 1.3.9 + JPEG (Baseline)
TurboVNC 0.4, Medium Quality JPEG (Baseline)
TurboVNC 0.5, Medium Quality JPEG
TurboVNC 0.5, Medium Quality JPEG, Palette Threshold=96
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugmx-01-24
glxspheres-24
googleearth-24
q3demo-24


Compression Time- Medium Quality JPEG
Compression Time (s)
1000
100
10
1
0.1
TightVNC 1.3.9 + JPEG (Baseline)
TurboVNC 0.4, Medium Quality JPEG (Baseline)
TurboVNC 0.5, Medium Quality JPEG
TurboVNC 0.5, Medium Quality JPEG, Palette Threshold=96
0.19
0.44
0.39
0.33
0.72
0.89
0.11
0.2
3.04
2.79
2.51
0.6
1.85
1.76
1.14
0.46
1.92
0.2
2
0.57
bugzilla-16
compilation-16
bars-16
kde-hearts-16
freshmeat-8
slashdot-24
photos-24
kde-hearts-24
3dsmax-04-24
catia-02-24
ensight-03-24
light-08-24
maya-02-24
proe-04-24
sw-01-24
tcvis-01-24
ugmx-01-24
glxspheres-24
googleearth-24
q3demo-24

The compression ratio of TurboVNC 0.5 with medium quality JPEG compared quite favorably to that of TightVNC 1.3.9. Most of the 3D datasets compressed better under TurboVNC (UGS NX, in particular, was 119% better.) Only two were significantly worse: CATIA (-29%) and Maya (-19%). Most of the 2D datasets compressed worse under TurboVNC, generally in the range of -20% to -40% relative to TightVNC. In all cases, the compression ratios met or exceeded the TurboVNC 0.4 baseline, and most exceeded it by a significant amount.

The compression ratio graph also reveals one reason why a palette threshold of 24 was chosen. Setting the palette threshold to 96 caused the compression ratio to fall below the TurboVNC 0.4 baseline by 9-18% for a few datasets (Quake 3 Demo, GLX Spheres, UGS TeamCenter Visualization, and Lightscape.)

It should also be noted that, even with a palette threshold of 24, some of the datasets fell short of the TurboVNC 0.4 performance baseline on this test. Pro/E was 39% slower than TurboVNC 0.4, EnSight was 23% slower, and 3D Studio Max was 11% slower. This was deemed to be an acceptable sacrifice, since medium quality JPEG is already quite a bit faster than perceptually lossless JPEG, and it is generally only used in cases in which the network is the primary bottleneck. However, setting the palette threshold to 96 made the situation much worse, and many of the other datasets now fell below the TurboVNC 0.4 performance target as well. That was not acceptable.

In short, a palette threshold of 24 was confirmed to be the best overall solution.