Evaluation and extension of SGI Vizserver

Tong Chu, James E. Fowler, Robert J. Moorhead II

Department of Electrical & Computer Engineering
Engineering Research Center
Mississippi State University, Mississippi State, MS

ABSTRACT

SGI Vizserver enables remote visualization in a manner transparent to the user application by producing rendered output at geographically remote locations while utilizing the powerful pipeline and expansive memory of an Onyx2 Infinite Reality machine located at some centralized place. Since the communication of visualization imagery typically requires enormous bandwidth, Vizserver offers two built-in options for compression which provide high-quality images at interactive frame rates for local-area networks with bandwidths of 100 Mbps. However, these built-in compressors are not well suited to truly remote users who are separated from the server by great distances and connected through low- and very-low-bandwidth links. In this paper, we propose two external compression algorithms that connect to Vizserver via an API to achieve 1) greater flexibility in terms of the user's control over distortion and bandwidth performance, and 2) better overall performance for truly remote users. Of all the techniques considered, we find that a simple frame-differencing scheme is best suited to very-low-bandwidth operation in that it achieves visually lossless performance at a frame rate higher than that of the built-in options, which saturate the network and incur substantial amounts of frame dropping.

Keywords: remote visualization, low-complexity video compression, DPCM, frame difference

1. INTRODUCTION

SGI Vizserver¹ is a product developed by Silicon Graphics, Inc., to enable remote-visualization applications. Specifically, SGI Vizserver is designed to provide users remote access to graphics pipelines of Onyx2 Infinite Reality machines so that they may view rendered output from visualization applications at geographically remote locations while utilizing the powerful pipeline and expansive memory of an Onyx2 machine located at a some centralized place.

The fundamental principle underlying SGI Vizserver is that all rendering takes place on the remote and more powerful Onyx2 instead of on the user's presumably less powerful local machine. Under this paradigm, the user runs a visualization application remotely, while Vizserver captures the rendered imagery output from the application and transmits an image sequence to the local machine for display. An important aspect of the product is that the operation of Vizserver is completely transparent to the application—the application renders visualization imagery as if it were running locally on the Onyx2, with all sophisticated hardware advantages (e.g., memory, graphics pipeline) of the Onyx2 utilized. Vizserver itself does no rendering; it merely controls where rendering takes place and provides a communication path to the local machine for display. Since the image sequences arising in visualization applications can typically have large resolution and frame rate, the bandwidth required of the communication link to the local machine can be particularly large. As a consequence, Vizserver attempts to mitigate the bandwidth burden on the communication link by providing optional compression of the transmitted images.

Although the field of video compression has advanced significantly over the last decade, most modern video-coding techniques are of little or no use in a remote-visualization system such as Vizserver which must simultaneously preserve high image quality, operate at high frame rates, and incur little delay, or *latency*, for interactive visualization sessions. Popular compression techniques such as the discrete cosine transform, wavelet transforms, and motion compensation are simply too computationally expensive to maintain interactive latency or fast frame rate. As a result, only a handful of algorithms are potential candidates for providing the extremely low-complexity compression needed by Vizserver. These techniques include simple differencing such as differential pulse code modulation (DPCM),² simple uniform scalar quantization,³ and low-complexity vector quantization³ such as block truncation coding (BTC).⁴

As currently implemented, Vizserver offers two built-in compression schemes, both of which are based on BTC. Both compression options have fixed encoding and decoding latency and, thus, running times which are independent of image content and depend only on image size. Relatively quick encoding and decoding is perhaps the most salient advantage of these algorithms; however, this advantage is coupled with high bandwidth demands. Additionally, these two compression schemes have the

Figure 1. Remote visualization using the remote-display capabilities inherent to the X Window System. Rendering takes place in the OpenGL software library, and resulting images are transmitted, uncompressed, to the X server using the X Protocol.

drawback of inflexibility in terms of the user's control over the bandwidth and image-quality performance, making it difficult for the user to fine-tune Vizserver to the particular application demands and network resources at hand.

In addition to the built-in image compression options, Vizserver offers an application program interface (API) which allows the addition of external compression algorithms. In this paper, we investigate two low-complexity video-coding techniques interfacing to Vizserver via its API with goal of adding greater flexibility in terms of bandwidth and image-quality performance. In particular, we strive to increase the range of quality-bandwidth choice available for low-bandwidth networks. Specifically, we look at DPCM and frame differencing which are simple differencing techniques that attempt to remove redundancy among pixels either within a frame (DPCM) or between successive frames (frame differencing). In experimental results, we compare the performance of these two compression extensions to that of Vizserver's built-in algorithms and illustrate that our frame-difference algorithm provides a substantially increased range in available performance. Additionally, if constrained to a very low bandwidths, the proposed techniques deliver faster frame rates with acceptable image quality, despite the greater encoding and decoding speed of the built-in compression options. We illustrate that, in this very-low-bandwidth case, the higher bandwidth demands of the built-in algorithms lead to increased data loss due to network congestion and thus a decreased frame rate relative to the performance of our proposed techniques.

2. VIZSERVER ARCHITECTURE

To best understand Vizserver and how it works, it is essential to understand how remote visualization can be accomplished in the absence of Vizserver. The most straightforward approach to remote visualization is to use the remote-display capabilities that have long been fundamental to the operation of the X Window System. This X-based remote-visualization paradigm is shown in Figure 1. Here, a visualization application runs on the remote machine while the OpenGL software library provides all graphics rendering. The OpenGL library produces, using software rendering routines, a sequence of images which is transmitted to the local machine for display.* For this transmission, the OpenGL library acts as a standard X client and communicates with the X server running on the local machine. The standard X Protocol is used for the communication; specifically, the images are transmitted individually, frame-by-frame, using a sequence of XPutImage commands, which simply transmit the image pixels in an uncompressed, raster-scan fashion. Although completely transparent to the application, this X-based approach typically does not represent a reasonable paradigm for remote-visualization applications. Although the application can employ the presumably large memory resources of the remote machine, all rendering takes place in software in the OpenGL library rather than employing the hardware graphics pipeline. This reliance on software rendering will usually result in a significantly decreased frame rate as compared to hardware rendering. An additional drawback to this X-based approach is that, because communications take place via the X Protocol, no compression is applied to the transmitted images. As a result, the bandwidth required of the communications link is usually prohibitive, unless the frame rate is even further reduced.

SGI Vizserver provides an alternative to X-based remote visualization while retaining the advantage of transparency to the visualization application.¹ A block diagram of the Vizserver system is shown in Figure 2. The application, completely unaware

*An alternative paradigm for X-based remote visualization entails the sending of OpenGL commands, rather than rendered images, from the remote to local machine, with rendering taking place on the local machine; for this approach to work, both machines must be equipped with special software to transmit/receive the GLX protocol for OpenGL-command transmission, otherwise operation defaults to as described above. We do not explore this possibility in this report as we assume we are interested in only approaches in which rendering takes place on the remote machine.

Figure 2. Remote visualization using SGI Vizserver. Rendering takes place in the hardware pipeline of the remote machine, and Vizserver compresses resulting images and transmits them to the local machine.

of the presence of Vizserver, runs on the remote machine as if the remote machine were local. That is, the OpenGL commands employed by the application are intercepted by the Vizserver server and directed to the remote-machine graphics pipeline. The Vizserver server then captures the pipeline output from the frame buffer and transmits the resulting sequence of images to the Vizserver client running on the local machine. Finally, the Vizserver client directs the images to the frame buffer of the local machine for display. The Vizserver system can optionally compress the images transmitted from the Vizserver server to the client to reduce the bandwidth required of the network. We overview the compression options provided by Vizserver in the following sections.

2.1. Built-in compression in Vizserver

The images extracted from the remote-machine frame buffer can be compressed by the Vizserver server before transmission to the local machine. There are three compression options built into Vizserver: 1) no compression, 2) color-cell compression (CCC), and 3) interpolated-cell compression (ICC). Both the CCC and ICC compression schemes are designed to provide very low latency and require very low computation overhead in order to not detrimentally affect the frame rate of the Vizserver system, nor impede its interactivity. Since both of these techniques are *lossy* compression methods, some distortion is introduced; i.e., the images displayed on the local machine after transmission are not exactly the same as those extracted from the frame buffer on the remote machine. As we illustrate below, the quantitative amount of this distortion, and its qualitative visual affect, are relatively small. On the other hand, the amount of compression achieved by these algorithms is correspondingly small as well, resulting in a relatively large bandwidth requirement that renders the built-in compression options particularly susceptible to data loss due to network congestion.

2.1.1. CCC

Color-Cell Compression (CCC)⁵ is a simple compression technique based upon a color-quantization variant of BTC.⁴ It has the advantages of requiring very low amounts of computation with a small latency that is fixed (i.e., not image dependent). As employed in Vizserver, CCC achieves a fixed compression ratio of 8:1. The operation of CCC is described below.

The CCC algorithm processes blocks of color pixels of size 4×4 . A 4×4 block is extracted from the image, the CCC algorithm compresses the pixels of that block, and then the procedure is repeated for the next 4×4 block. The operations applied to each 4×4 block are as follows. Each pixel in the image block is originally represented as an 8-bit red value, an 8-bit green value, and an 8-bit blue value, for a total of 24 bits/pixel. Using these 24 bits/pixel of color information, the luminance of each pixel in the block is calculated. The range of luminance values in the block is found and a threshold within this range is calculated. Each luminance value in the block is then compared to the threshold to create a 4×4 binary mask indicating whether the corresponding luminance value is above or below the threshold. This mask-creation process is illustrated in Figure 3a. The binary mask is then used to partition the original color image block into two fields—those pixels corresponding to mask value 0 and those corresponding to mask value 1. Using the colors of the pixels corresponding to mask value 0, an “average” color

is calculated, shown in Figure 3b as *Color 0*. A similar process for pixels corresponding to mask value 1 yields *Color 1*. Both *Color 0* and *Color 1* are calculated as 24-bit color values—the final step in the CCC algorithm is to quantize these two colors to 5 bits of red, 6 bits of green, and 5 bits of blue each (this quantization is implemented by preserving only the x most significant bits of the 8-bit color-component value, where $x = 5$ for red and blue, and $x = 6$ for green). Thus, both *Color 0* and *Color 1* are represented using 16 bits each.

On the decoding side, the Vizserver client receives the mask and the 16-bit representations of *Color 0* and *Color 1*. An approximation to the original 4×4 image block is obtained by using the mask to place either *Color 0* or *Color 1* into the block as appropriate. This block is then placed into the image being reconstructed by the Vizserver client, and the process repeats for the next block.

The CCC algorithm is a compression scheme with fixed latency. That is, the amount of time needed to code each 4×4 block is independent of the contents of the block, and is the same for all blocks. As a consequence, the block-coding time depends on only the speed of the remote machine. Likewise, on the client side, the decoding of each block depends only on the speed of the local machine. Since the CCC decoding process is less complicated than the CCC encoding process (decoding does not need to calculate luminance or find *Color 0/Color 1*), the latency of the Vizserver client is usually much less than that of the Vizserver server.

The bit rate achieved by the CCC algorithm is as follows. For each 4×4 block of 24-bit color pixels, the CCC algorithm outputs a 16-bit binary mask, 16 bits for *Color 0*, and 16 bits for *Color 1*. Thus, the bit rate for CCC is

$$\frac{16 + 16 + 16}{4 \times 4} = 3 \text{ bits/pixel.} \quad (1)$$

Since the original image is represented with 24 bits/pixel, the compression ratio is

$$R_{\text{CCC}} = \frac{24}{3} = 8 : 1. \quad (2)$$

2.1.2. ICC

The Interpolated Cell Compression (ICC) algorithm was developed by SGI specifically for the Vizserver application. ICC compression is very similar to the CCC algorithm described above. The sole difference is that the mask specifies four colors rather than two. That is, instead of being a binary mask as in CCC, the ICC mask contains 2 bits for each pixel of the block. This allows the mask to specify a total of 4 possible colors: *Color 0*, *Color 1*, *Color 2*, and *Color 3*. An interpolation strategy based upon luminance values is used to determine which of these four colors is used for each pixel in the block. Like CCC, the ICC algorithm is a fixed-latency compression scheme; since ICC is more complicated than CCC, the ICC latency is more than that of CCC.

The bit rate achieved by the ICC algorithm is as follows. For each 4×4 block of 24-bit color pixels, the ICC algorithm outputs a $4 \times 4 \times 2 = 32$ bit mask, 16 bits for *Color 0*, 16 bits for *Color 1*, 16 bits for *Color 2*, and 16 bits for *Color 3*. Thus, the bit rate for ICC is

$$\frac{32 + 16 + 16 + 16 + 16}{4 \times 4} = 6 \text{ bits/pixel.} \quad (3)$$

Since the original image is represented with 24 bits/pixel, the compression ratio for ICC is

$$R_{\text{ICC}} = \frac{24}{6} = 4 : 1. \quad (4)$$

2.2. Vizserver API

The Vizserver system includes an API which allows for custom, external compression algorithms to be added to Vizserver. The external compression “modules” connect to the Vizserver server via the API, receive a rendered image, and then apply the custom compression algorithm. The resulting compressed bitstream is then sent to the Vizserver client through the API. On the decoding side, the bitstream is received and properly decoded by the corresponding decoding module.¹ The Vizserver API gives users the ability to implement compression algorithms tailored to their particular application as well as to the network and computational resources available. A broad range of performance options thus becomes possible, albeit under certain limitations that are imposed by the API and which we examine below.

Figure 3. The operation of the CCC compression algorithm. (a) Extraction of the binary mask based on thresholding the luminance values of the pixels in the 4×4 block. (b) Creation and quantization to 16-bits of the two colors used to represent all 16 pixels of the block.

In order to permit fast execution speed and thus high frame-encoding rate, Vizserver permits the use of multithread programming in the encoder; however, only limited synchronization logic is provided. In particular, the order in which data is presented to the external encoder module through the API is not guaranteed to match the true temporal order of the data. That is, blocks of data from within a given frame may be sent through the API in any order, not necessarily the “raster-scan” order that an external encoder module might expect. This lack of guaranteed temporal order might be problematic for spatial-decorrelation schemes which attempt to remove redundancy among pixels within a frame. However, this potential problem can be overcome by requiring Vizserver to pass all pixels for a given frame to the external compression module at once, rather than allowing the data to be sent in blocks smaller than a frame. To do so, the external compression module designates that the data block size for API communication is an entire frame.

A further complication arises in that, under conditions of network congestion, compressed frames may be discarded, or “dropped,” by Vizserver after compressed data is received from the external encoder module but before transmission of the compressed data to the client. As a result, the external decoder module is not guaranteed to receive all the frames that the encoder processed; therefore, temporal-decorrelation schemes, which attempt to remove redundancy between successive frames, are not feasible.

3. VIZSERVER EXTENSION VIA ADDITION OF EXTERNAL COMPRESSION MODULES

Since the compression ratio for Vizserver’s built-in compression options is fixed, users can control the bandwidth required by these compressors only through changing the size of the rendered frame. The primary goal of the present work is to offer greater flexibility by providing direct user control of the tradeoff between bandwidth and image quality. Thus, users will be better able to tailor Vizserver to their application needs and to their computational and network resources. Additionally, when we consider experimental results below, we will see below that Vizserver’s built-in compression options produce relatively low-distortion

Figure 4. DPCM encoder module— Q is uniform scalar quantization, Q^{-1} is inverse quantization, P is the DPCM predictor

images at the local client, but this high image fidelity comes at a cost of significant bandwidth burden. Indeed, Vizserver is intended to be deployed over networks of capacious bandwidth (on the order of 100 Mbps), which would essentially require that the user be on the same local-area network (LAN) as the server. This bandwidth requirement is too high for truly remote users—those users geographically separated from the remote server by many miles and connected through only low-bandwidth links (1.5-10 Mbps, i.e., T1 link to 10-Mbps Ethernet). Thus, a secondary goal of the work we report here is to provide better performance for truly remote users by reducing the bandwidth burden with minimal sacrificing of frame rate. Below, we describe two compression algorithms that we investigate as external compression modules added to the Vizserver system in order to achieve these goals.

3.1. DPCM

DPCM² is a common predictive coding scheme with a long history of use in compression applications. The DPCM encoder calculates a prediction of the current pixel from neighboring pixels, and this prediction is subtracted from the actual pixel value to form a difference value. The collection of difference values, the “difference image,” is then quantized and encoded. The DPCM encoder tends to remove “spatial redundancy”—the intraframe correlation between pixels—to achieve compression.² The DPCM decoder merely inverts the quantization and adds the quantized difference value to a prediction to form the reconstructed pixel. In order to prevent decoder drift—the undesirable situation in which reconstructed pixels progressively diverge from their true values—the predictions formed in the encoder and in the decoder must be the same. Because the decoder can access only reconstructed pixel values in forming a prediction, the encoder must make its prediction from the same reconstructed pixels as well. Thus, the encoder includes a “copy” of the decoder so that the encoder can “track” decoder operation and force synchronization. As noted previously, the multithreaded nature of encoding in Vizserver serves to undermine such synchronization between encoder and decoder. However, in the case of DPCM, synchronization dependency can be confined to a single frame by forcing Vizserver to supply image data in blocks consisting of an entire frame and “restarting” both the encoder and decoder to a known state after each frame. Under these conditions, a DPCM encoder and decoder pair within the Vizserver system can maintain synchronization at all times, since Vizserver drops only entire frames when network congestion is encountered.

Figure 4 shows the architecture of the DPCM encoder module. Operation is as follows. First, pixels are converted from red-green-blue (RGB) color space to luminance-chrominance (YUV) color space. Since the UV chrominance components have

Table 1. The runlength/Rice entropy coder

<i>Codeword</i>	<i>Sequence of quantizer indices</i>
0	Run of 2^k zeros
1c0v	Run of $c < 2^k$ zeros, followed by positive index i (v is the binary representation of i)
1c1v	Run of $c < 2^k$ zeros, followed by negative index i (v is the binary representation of $-i$)

typically less perceptual relevance than does luminance Y ,² the encoder module downsamples each chrominance component by a factor of two. Then, DPCM encoding is applied to each component separately, with simple prediction and quantization used to minimize latency. Specifically, the DPCM predictor for pixel $x(m, n)$ at location (m, n) within the current frame is

$$\tilde{x}(m, n) = \frac{x(m-1, n) + x(m, n-1)}{2}, \quad (5)$$

while the difference value is calculated as

$$d(m, n) = x(m, n) - \tilde{x}(m, n). \quad (6)$$

Next, uniform scalar quantization is applied to the difference value to form an “image” of quantizer indices $i(m, n)$,

$$i(m, n) = \text{round} \left(\frac{d(m, n)}{q} \right), \quad (7)$$

where q is the quantizer stepsize, and $\text{round}(\cdot)$ is rounding to the nearest integer. Finally, an entropy coding produces an output bitstream from the quantization indices.

The entropy coder employed in our DPCM compression module is a runlength/Rice coder.⁶ In this technique, the symbol “0” represents a sequence of 2^k quantization indices which are all zero, while a “1” followed by a k -bit number c represents a sequence of fewer than 2^k zeros followed by a nonzero quantization index. Each nonzero quantization index is encoded in sign-magnitude form. Table 1 illustrates the details. The parameter k is adapted in backward fashion, increasing every time a “0” codeword is emitted, and decreasing every time a codeword starting with a “1” is emitted. This Rice coder is very close to being an optimal variable-to-variable length coder while its computational complexity is lower than that of other forms of entropy coding,⁶ e.g., arithmetic coding.⁷

3.2. Frame difference

While DPCM aims at reducing the spatial redundancy due to correlation between neighboring pixels within a single frame, frame differencing aims at reducing temporal redundancy due to correlation between successive frames. Since successive frames in a visualization session are usually very similar, the previous frame often provides a fairly accurate prediction of the current frame. We therefore expect frame differencing to provide quite efficient compression within a remote visualization system.

Algorithmically, frame differencing is identical to DPCM—only the predictor is different. For frame differencing, the prediction of pixel $x(m, n, t)$ at location (m, n) in the frame at time t is given by

$$\tilde{x}(m, n, t) = x(m, n, t-1). \quad (8)$$

To incorporate frame differencing into Vizserver, we would employ an external compression module with architecture identical to that depicted in Figure 4. However, as discussed in Section 2.2, Vizserver may arbitrarily drop frames after they are encoded, causing the encoder and decoder of a frame-difference module to lose synchronization. Therefore, frame differencing cannot be feasibly implemented within Vizserver as an external compression module. Since frame differencing cannot be deployed in the Vizserver system in reality, we use software simulation to estimate bandwidth, frame-rate, and image-quality figures for our experimental results for a hypothetical frame-differencing compressor. We find that Vizserver’s limitation on temporal decorrelation is particularly unfortunate since our experimental results show that, for very-low-bandwidth networks, frame differencing outperforms not only DPCM but also the built-in Vizserver compression options. We turn our attention to these experimental results in the next section.

4. EXPERIMENTAL RESULTS

In order to evaluate the performance of the Vizserver system, as well as that of the external compression modules we proposed above, we conducted a number of experiments employing a variety of visualization applications. In our experiments, we observed the performance of Vizserver by measuring several key criteria, namely

- the *frame rate* of images rendered at the local machine,
- the *network bandwidth* of the communication occurring between the remote and local machines, and
- the *distortion* (image quality) between the images displayed on the local machine and the original images rendered on the remote machine.

Clearly, these criteria are interrelated, with performance for one affecting performance of the others. For example, if distortion is decreased (i.e., image quality improved), one expects to observe a rise in bandwidth. Below, we examine the factors affecting performance for each of these criteria.

Frame rate—One of the key components of the overall performance of Vizserver is the frame rate of the images as displayed on the local machine. Clearly, the local-machine display frame rate is dependent on many factors, including the current conditions of the remote machine, the local machine, and the network in between. Specifically, the visualization application, unaware of the presence of Vizserver, renders images at its normal, application-dependent frame rate using the remote-machine pipeline. If the Vizserver server encodes frames at a rate that is less than the rate at which frames are rendered through the graphics pipeline, frames are in effect rendered but discarded (i.e., not encoded and thus not transmitted to the client). Since larger frames take longer to compress and encode, increasing the size of the rendered frame will reduce the rate that frames are transmitted to the client. If, on the other hand, Vizserver finishes the encoding of a frame but finds the network *saturated* to its full transmission capacity, Vizserver discards the encoded frame rather than waiting for the network congestion to clear. This frame dropping results in a slower rate of frames being displayed on the local machine. In our experimental results, frame-rate figures are an estimated time-average frame rate as observed by using the SGI IRIX `osview` command on the local O2 machine to measure the number of `swapbuf` completes over a one-second interval, which gives a rough estimate the number of frames per second (fps) displayed.

Bandwidth—In addition to depending on the encoder and decoder speeds, the frame rate observed at the local machine also depends on the bandwidth that can be supported by the network—frames may be lost due to network congestion, as mentioned above. Since all frames displayed at the local machine must have been transmitted across the network, in our results, we estimate the network bandwidth directly from the observed local-machine frame rate. Specifically, the bandwidth in bits/second (bps) is

$$B = \frac{f \times M \times N \times 24}{R}, \quad (9)$$

where f is the observed local-machine frame rate, M and N give the width and height, respectively, of the frame, and R is the compression ratio employed ($R_{\text{CCC}} = 8$, $R_{\text{ICC}} = 4$, $R_{\text{none}} = 1$, $R_{\text{DPCM}} = \text{variable}$, $R_{\text{frame diff.}} = \text{variable}$). In reality, the true bandwidth may differ slightly from this estimate due to inaccuracy in the measuring of frame rate as a time average; however, we have generally found a good agreement between this estimate and direct observation of bandwidth using the IRIX `netstat` command.

Distortion—As part of our experimental evaluation of Vizserver, we measured the image distortion quantitatively, as well as investigated the visual affects of this distortion. Quantitatively, we used a signal-to-noise ratio (SNR) derived from the 1964 CIE color-space distance measure² (as calculated in the CIE Modified UCS space) as a distortion metric.

Table 2 gives the details of the architectures of the remote and local machines we used in the following experiments. We use Vizserver Version 1.1, released April 2000, for all measurements. Although we have evaluated Vizserver for several visualization applications, we present specific results here for an ocean-current visualization that runs locally on the Onyx2 machine at 36 fps frame rate with 640×480 frame size.

Table 2. Remote and local machines used in experimental observations

	<i>Remote Machine</i>	<i>Local Machine</i>
Machine Type	Onyx2	O2
Processor Type	R10000	R5000
Processor Speed	250 MHz	200 MHz
Number of Processors	8	1
Primary Cache Size	32 kB	32 kB
Secondary Cache Size	4 MB	1 MB
Main Memory Size	4 GB	256 MB

Table 3. Performance of built-in compressors over a 100-Mbps switched Ethernet network

<i>Compression Method</i>	<i>Frame Rate (fps)</i>	<i>Network</i>	
		<i>Bandwidth (Mbps)</i>	<i>Distortion (dB)</i>
CCC	16	14.7	11.1
ICC	14	25.8	12.6
None	8	59.0	lossless

4.1. High-bandwidth performance

Table 3 shows our measurements for Vizserver's built-in compression options when deployed over a 100-Mbps switched Ethernet, a relatively high-bandwidth network. The natural frame rate for the application (i.e., the frame rate observed when the application runs locally on the Onyx2 machine) was 36 fps. We see from these results that, when CCC compression is employed, Vizserver manages to display roughly 16 fps on the local machine, or almost $\frac{1}{2}$ of the frames rendered on the remote machine. However, when ICC is used, the frame rate drops slightly as expected since, being a slightly more computationally complex compression algorithm, ICC will require slightly more time to encode each frame, resulting in a greater number of frames that are rendered but not encoded. On the other hand, when no compression is used, the encoder speed is actually at its fastest (no processing is done); however, a *slower* frame rate is observed for the no-compression case. Since an unloaded switched Ethernet can typically achieve throughput of only 60% of its rated bandwidth due to Ethernet packet overhead, the network becomes saturated at 60 Mbps. In our experiments, the 100-Mbps Ethernet saturates when no compression is used, and frames are dropped due to network congestion despite the fact that the increased encoder speed causes fewer frames to be lost due to encoder latency. The net result is a frame rate slower than when CCC or ICC is used. In terms of image quality, we see in Table 3 that ICC produces slightly higher SNR than CCC, but examining reconstructed images reveals that the distortion incurred by both methods is imperceptible, i.e., both methods are visually lossless.

4.2. Low-bandwidth performance

100 Mbps is a bandwidth typical for a LAN; however, truly remote users may face much more modest network capacities. For instance, a 10-Mbps Ethernet (with true bandwidth capacity of 6 Mbps) or a 1.5-Mbps T1 line may be more realistic bandwidths encountered by many remote users. Figure 5 illustrates bandwidth and distortion performance in this low-bandwidth range for the custom compression modules proposed in Section 3. Varying the quantizer stepsize in the DPCM algorithm produces bandwidths that scale smoothly from about 0.5 Mbps to 8 Mbps. We measure DPCM frame-rate performance at about 2.7 fps regardless of bandwidth, since network saturation does not occur at the high compression ratios obtained by our DPCM module.

Recall that, since frame differencing is not supported by the Vizserver API, we use a software simulation to estimate performance figures for a hypothetical frame-differencing compressor. To estimate frame-rate performance, we time the encoder speed of the simulation on the Onyx2 server at roughly $0.48\mu s$ per pixel, which gives about 6.7 fps for a 640×480 frame size. On the other hand, the decoder on the slower O2 machine was timed at about $1.2\mu s$ per pixel, corresponding to about 2.7 fps. Since the slower of these rates would be the rate for the system as a whole, we estimate that the frame-differencing compressor would achieve a frame-rate of about 2.7 fps, the same as for DPCM. The continuum of distortion and bandwidth figures as measured with the frame-difference simulation is plotted in Figure 5.

Figure 5. Low-bandwidth performance of DPCM, frame differencing, and the built-in Vizserver compressors. Frame rates (in fps) as measured on the local machine are provided for each technique.

Due to limitations in our experimental setup (i.e., only a 100-Mbps Ethernet was available in our lab), performance measurements for the built-in CCC and ICC methods were available only for the high-bandwidth case. However, frame dropping does not affect distortion performance, so we can assume that, if deployed over a low-bandwidth network, the built-in compressors would operate at network saturation with the same distortions measured in Section 4.1. In Figure 5, we show the four performance points we would expect for CCC and ICC operating with saturated T1 (1.5 Mbps) and 10-Mbps Ethernet (6 Mbps) links.

For the visualization sequence we consider here, distortion is visibly tolerable at CIE SNR values of 5 dB or greater, while distortions above about 10 dB are imperceptible. We see that frame differencing achieves the minimum tolerable distortion at about 1.25 Mbps, less than the bandwidth of the T1 link. At 1.5 Mbps, the built-in compression options achieve somewhat higher SNR than that of frame differencing, but this advantage is of minimal significance since all three techniques operate at visibly lossless distortion levels. Only the DPCM compressor has substantial distortion at the 1.5 Mbps bandwidth mark. The network congestion that would be encountered by the built-in options when operating at 1.5 Mbps would entail a significant amount of frame dropping; so much so, in fact, that CCC and ICC would operate at only 1.6 and 0.8 fps, respectively, while frame differencing and DPCM achieve rates between 1.5 and 3 times faster. Clearly, if it were possible to add frame differencing to Vizserver, we could obtain good quality images at very low bandwidths without slowing the frame rate as much as is necessary with the built-in options.

Turning our attention to the 10-Mbps performance mark, we see that the built-in options offer performance very similar to that of our DPCM compressor, while our hypothetical frame-differencing technique achieves a slight superiority in distortion performance (again, this is negligible since all the techniques are visually lossless at this bandwidth). In this case, though the built-in options still saturate the network, the amount of frame dropping is less, and the built-in options achieve frame rates substantially higher than either that of DPCM or frame differencing. Thus, it appears that the superiority of frame differencing

appears primarily at very low bandwidths rather than at moderately low bandwidths.

As a final note, we observe that, in the low-bandwidth range of performance considered here, the built-in compressors always operate at network saturation; that is, they will consume all available bandwidth. Under real circumstances, not all network bandwidth will be available to Vizserver, and so the performance of the built-in options as presented in Figure 5 may be overly optimistic. However, both the frame-differencing and DPCM approaches that we have proposed offer a smooth continuum of performance points in the low-bandwidth range. For example, the bandwidth of the frame-differencing module could be scaled back from 6 Mbps to 4 or even 3 Mbps with only a small impact on distortion performance, and no penalty in frame rate, in order to free up bandwidth for other applications sharing the network link.

5. CONCLUSIONS

The goal of the work we presented here is the development of low-complexity video-coding schemes that “plug into” SGI Vizserver in order to achieve 1) greater flexibility in terms of the user’s control over distortion and bandwidth performance, and 2) better overall performance for truly remote users. Vizserver’s two built-in compression options achieve interactive frame rates, but only when supplied with the relatively high bandwidths that one typically encounters only on a LAN. Additionally, these options are inflexible in terms of distortion-bandwidth tradeoff possibilities. We consider and implement two other algorithms based on simple differencing that give users more flexibility in choosing performance. Additionally, our proposed techniques achieve fairly high compression as well, and thus do not suffer from the network congestion, or corresponding severe frame dropping, as do the built-in options when deployed on very-low-bandwidth links.

Our experiments show that one of the techniques we considered here, frame differencing, exhibits particularly strong potential for very-low-bandwidth performance, achieving tolerable distortion performance without saturating a very-low-bandwidth network and maintaining a frame rate higher than that of the built-in options. However, the Vizserver API places several limitations on the architecture of external compression modules—schemes employing temporal decorrelation, such as our frame-differencing proposal, are not implementable. Thus, our investigation of this approach has been strictly hypothetical and conducted via software simulation rather than actual implementation within the Vizserver system. DPCM, another simple differencing scheme that we implement and investigate, relies on merely intraframe, rather than interframe, differencing, and is thus implementable as an external Vizserver compression module. Unfortunately, we find that performance for this technique as is somewhat inferior to that of the hypothetical frame-differencing compressor.

As a final note, we observe that, in the many years that the problem of video coding has been considered, only a few low-complexity video-coding schemes have emerged. We have discussed four here. In general terms, the compression options built-in to Vizserver tend to provide excellent image quality and fast frame rates but at a cost of heavy bandwidth burden. On the other hand, our proposed simple differencing approaches provide the substantial compression needed to operate at very low bandwidths without network congestion, but the frame rates achieved are significantly less than the application’s inherent rendering rate. Unfortunately, we know of no coding scheme that can achieve both great speed and high compression simultaneously.

ACKNOWLEDGMENTS

This work was done as part of the High Performance Visualization Center Initiative (HPVCI) funded by the Department of Defense High Performance Computing Modernization Program (HPCMP).

REFERENCES

1. C. Ohazama, “OpenGL Vizserver White Paper.” White Paper, 1999. Silicon Graphics, Inc.
2. A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.
3. A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.
4. M. Rabbani and P. W. Jones, *Digital Image Compression Techniques*, SPIE, Bellingham, WA, 1991.
5. G. Cambell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg, and D. J. Sandin, “Two Bit/Pixel Full Color Encoding,” in *Computer Graphics (Proceedings of SIGGRAPH 86)*, D. C. Evans and R. J. Athay, eds., vol. 20, pp. 215–223, (Dallas, TX), August 1986.
6. H. S. Malvar, “Fast Progressive Wavelet Coding,” in *Proceedings of the IEEE Data Compression Conference*, J. A. Storer and M. Cohn, eds., pp. 336–343, (Snowbird, UT), March 1999.
7. I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmetic Coding for Data Compression,” *Communications of the ACM* **30**, pp. 520–540, June 1987.