Software-only Compression, Rendering, and Playback of Digital Video

Paramvir Bahl
Paul S. Gauthier
Robert A. Ulichney

Software-only digital video involves the compression, decompression, rendering, and display of digital video on general-purpose computers without specialized hardware. Today's faster processors are making software-only video an attractive, low-cost alternative to hardware solutions that rely on specialized compression boards and graphics accelerators. This paper describes the building blocks behind popular ISO, ITU-T, and industry-standard compression schemes, along with some novel algorithms for fast video rendering and presentation. A platform-independent software architecture that organizes the functionality of compressors and renderers into a unifying software interface is presented. This architecture has been successfully implemented on the Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems. To maximize the performance of codecs and renderers, issues pertaining to flow control, optimal use of available resources, and optimizations at the algorithmic, operating-system, and processor levels are considered. The performance of these codecs on Alpha systems is evaluated, and the ensuing results validate the potential of software-only solutions. Finally, this paper provides a brief description of some sample applications built on top of the software architecture, including an innovative video screen saver and a software VCR capable of playing multiple, compressed bit streams.

Full-motion video is fast becoming commonplace to users of desktop computers. The rising expectations for low-cost, television-quality video with synchronized sound have been pushing manufacturers to create new, inexpensive, high-quality offerings. The bottlenecks that have been preventing the delivery of video without specialized hardware are being cast aside rapidly as faster processors, higher-bandwidth computer buses and networks, and larger and faster disk drives are being developed. As a consequence, considerable attention is currently being focused on efficient implementations of flexible and extensible software solutions to the problems of video management and delivery. This paper surveys the methods and architectures used in software-only digital video systems.

Due to the enormous amounts of data involved, compression is almost always used in the storage and transmission of video. The high level of information redundancy in video lends itself well to compression, and many methods have been developed to take advantage of this fact. While the literature is replete with compression methods, we focus on those that are recognized as standards, a requirement for open and interoperable systems. This paper describes the building blocks behind popular compression schemes of the International Organization for Standardization (ISO), the International Telecommunication Union-Telecommunication Standardization Sector (ITU-T), and within the industry.

Rendering is another enabling technology for video on the desktop. It is the process of scaling, color adjusting, quantization, and color space conversion of the video for final presentation on the display. As an example, Figure 1 shows a simple sequence of video decoding. In the section Video Presentation, we discuss rendering methods, along with some novel algorithms for fast video rendering and presentation, and describe an implementation that parallels the techniques used in Digital's hardware video offerings.

We follow that discussion with the section The Software Video Library, in which we present a common architecture for video compression, decompression, and playback that allows integration into Digital's multimedia products. We then describe two sample applications, the Video Odyssey screen saver

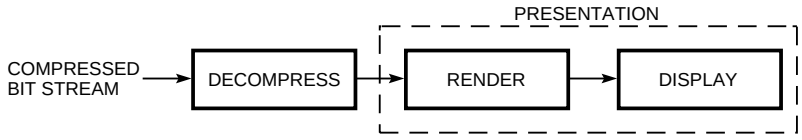


Figure 1
Components in a Video Decoder Pipeline

and a software-only video player. We conclude our paper by surveying related work in this rapidly evolving area of software digital video.

Video Compression Methods

A system that compresses and decompresses video, whether implemented in hardware or software, is called a video codec (for compressor/decompressor). Most video codecs consist of a sequence of components usually connected in pipeline fashion. The codec designer chooses specific components based on the design goals. By choosing the appropriate set of building blocks, a codec can be optimized for speed of decompression, reliability of transmission, better color reproduction, better edge retention, or to perform at a specific target bit rate. For example, a codec could be designed to trade off color quality for transmission bit rate by removing most of the color information in the data (color subsampling). Similarly a codec may include a simple decompression model (less processing per pixel) and a complex compression process to boost the playback rate at the expense of longer compression times. (Compression algorithms that take longer to compress than to decompress are said to be asymmetric.) Once the components and trade-offs have been chosen, the designer then fine tunes the codec to perform well in a specific application space such as teleconferencing or video browsing.

Video Codec Building Blocks

In this section, we present the various building blocks behind some popular and industry-standard video codecs. Knowledge of the following video codec components is essential for understanding the compression process and to appreciate the complexity of the algorithms.

Chrominance Subsampling Video is usually described as being composed of a sequence of images. Each image is a matrix of pixels, and each pixel is represented by three 8-bit values: a single luminance value (Y) that signifies brightness, and two chrominance values (U and V, or sometimes Cb and Cr) which, taken together, specify a unique color. By reducing the amount of color information in relation to luminance (subsampling the chrominance), we can reduce the size of an image with little or no perceptual effect. The most common chrominance subsampling technique decimates the color signal by 2:1 in the horizontal direction. This is done either by simply throwing out the color information of alternate pixels or by averaging the colors of two adjacent pixels and using the average for the color of the pixel pair. This technique is commonly referred to as 4:2:2 subsampling. When compared to a raw 24-bit image, this results in a compression of two-thirds. Decimating the color signal by 2:1 in both the horizontal and the vertical direction (by ignoring color information for alternate lines in the image) starts to result in some perceptible loss of color, but the compression increases to one-half. This is referred to as 4:2:0 subsampling: for every 4 luminance samples, there is a single color specified by a pair of chrominance values. The ultimate chrominance subsampling is to throw away all color information and keep only the luminance data (monochrome video). This not only reduces the size of the input data but also greatly simplifies processing for both the compressor and the decompressor, resulting in faster codec performance. Some teleconferencing systems allow the user to switch to monochrome mode to increase frame rate.

Transform Coding Converting a signal, video or otherwise, from one representation to another is the task of a transform coder. Transforms can be useful for video compression if they can convert the pixel data into a form in which redundant and insignificant information in the video's image can be isolated and removed. Many transforms convert the spatial (pixel) data into frequency coefficients that can then be selectively eliminated or quantized. Transform coders address three central issues in image coding: (1) decorrelation (converting statistically dependent image elements into independent spectral coefficients), (2) energy compaction (redistribution and localization of energy into a small number of coefficients), and (3) computational complexity. It is well documented that human vision is biased toward low frequencies. By transforming an image to the frequency domain, a codec can capitalize on this knowledge and remove or reduce the high-frequency components in the quantization step, effectively compressing the image. In addition, isolating and eliminating high-frequency components in an image results in noise reduction since most noise in video, introduced during

the digitization step or from transmission interference, appears as high-frequency coefficients. Thus transforming helps compression by decorrelating (or whitening) signal samples and then discarding nonessential information from the image.

Unitary (or orthonormal) transforms fall into either of two classes: fixed or adaptive. Fixed transforms are independent of the input signal; adaptive transforms adapt to the input signal.[1] Examples of fixed transforms include the discrete Fourier transform (DFT), the discrete cosine transform (DCT), the discrete sine transform (DST), the Harr transform, and the Walsh-Hadamard transform (WHT). An example of an adaptive transform is the Karhunen-Loeve transform (KLT). Thus far, no transform has been found for pictorial information that completely removes statistical dependence between the transform coordinates. The KLT is optimum in the mean square error sense, and it achieves the best energy compaction; however, it is computationally very expensive. The WHT is the best in terms of computation cost since it requires only additions and subtractions; however, it performs poorly in decorrelation and energy compaction. A good compromise is the DCT, which is by far the most widely used transform in image coding. The DCT is closest to the KLT in the energy-packing sense, and, like the DFT, it has fast computation algorithms available for its implementation.[2] The DCT is usually applied in a sliding window on the image with a common window size of 8 pixels by 8 lines (or simply, 8 by 8). The window size (or block size) is important: if it is too small, the correlation between neighboring pixels is not exploited; if it is too large, block boundaries tend to become very visible. Transform coding is usually the most time-consuming step in the compression/decompression process.

Scalar Quantization A companion to transform coding in most video compression schemes is a scalar quantizer that maps a large number of input levels into a smaller number of output levels. Video is compressed by reducing the number of symbols that need to be encoded at the expense of reconstruction error. A quantizer acts as a control knob that trades off image quality for bit rate. A carefully designed quantizer provides high compression for a given quality. The simplest form of a scalar quantizer is a uniform quantizer in which the quantizer decision levels are of equal length or step size. Other important quantizers include Lloyd-Max's minimum mean square error (MMSE) quantizer and an entropy constraint quantizer.[3,4] Pulse code modulation (PCM) and adaptive differential pulse code modulation (ADPCM) are examples of two compression schemes that rely on pure quantization without regard to spatial and temporal redundancies and without exploiting the nonlinearity in the human visual system.

Predictive Coding Unless the image is changing rapidly, a video sequence will normally contain sequences of frames that are very similar. Predictive coding uses this fact to reduce the data volume by comparing pixels in the current frame with pixels in the same location in the previous frame and encoding the difference. A simple form of predictive coding uses the value of a pixel in one frame to predict the value of the pixel in the same location in the next frame. The prediction error, which is the difference between the predicted value and the actual value of the pixel, is usually small. Smaller numbers can be encoded using fewer quantization levels and fewer coding bits. Often the difference is zero, which can be encoded very compactly. Predictive coding can also be used within an image frame where the predicted value of a pixel may be the value of its neighbor or a weighted average of the pixels in the region. Predictive coding works best if the correlation between adjacent pixels that are spatially as well as temporally close to each other is strong. Differential PCM and delta modulation (DM) are examples of two compression schemes in which the predicted error is quantized and coded. The decompressor recovers the signal by applying this error to its predicted value for the sample. Lossless image compression is possible if the prediction error is coded without being quantized.

Vector Quantization An alternative to transform-based coding, vector quantization attempts to represent clusters of pixel data (vectors) in the spatial domain by predetermined codes.[5] At the encoder, each data vector is matched or approximated with a code word in the codebook, and the address or index of that code word is transmitted instead of the data vector itself. At the decoder, the index is mapped back to the code word, which is then used to represent the original data vector. Identical codebooks are needed at the compressor (transmitter) and the decompressor (receiver). The main complexity lies in the design of good representative codebooks and algorithms for finding best matches efficiently when exact matches are not available. Typically, vector quantization is applied to data that has already undergone predictive coding. The prediction error is mapped to a subset of values that are expected to occur most frequently. The process is called vector quantization because the values to be matched in the tables are usually vectors of two or more values. More elaborate vector quantization schemes are possible in which the difference data is searched for larger groups of commonly occurring values, and these groups are also mapped to single index values.

The amount of compression that results from vector quantization depends on how the values in the codebooks are calculated. Compression may be adjusted smoothly by designing a set of codebooks

and picking the appropriate one for a given desired compression ratio.

Motion Estimation and Compensation Most codecs that use interframe compression use a more elaborate form of predictive coding than described above. Most videos contain scenes in which one or more objects move across the image against a fixed background or in which an object is stationary against a moving background. In both cases, many regions in a frame appear in the next frame but at different positions. Motion estimation tries to find similar regions in two frames and encodes the region in the second frame with a displacement vector (motion vector) that shows how the region has moved. The technique relies on the hypothesis that a change in pixel intensity from one frame to another is due only to translation.

For each region (or block) in the current frame, a displacement vector is evaluated by matching the information content of the measurement window with a corresponding measurement window W within a search area S, placed in the previous frame, and by searching for the spatial location that minimizes the matching criterion $\vec{d}$. Let $L_i(x,y)$ represent the pixel intensity at location (x,y) in frame i; and if (d_x,d_y) represents the region displacement vector for the interval $n(=(i+n)-i)$, then the matching criterion is defined as

$$\vec{d} = \min_{(d_x,\,d_y)\in S} \left\{ \sum_{(x,y)\in W} \left\| L_i(x,y) - L_{i-n}(x-d_x, y-d_y) \right\| \right\} \quad n \geq 1 \tag{1}$$

The most widely used distance measures are the absolute value $||x||=|x|$ and the quadratic norm $||x||=x^2$. Since finding the absolute minimum is guaranteed only by performing an exhaustive search of a series of discrete candidate displacements within a maximum displacement range, this process is computationally very expensive. A single displacement vector is assigned to all pixels within the region.

Motion compensation is the inverse process of using a motion vector to determine a region of the image to be used as a predictor.

Although the amount of compression resulting from motion estimation is large, the coding process is time-consuming. Fortunately, this time is needed only in the compression step. Decompression using motion estimation is relatively fast since no searching has to be done. For data replenishment, the decompressor simply uses the transmitted vector and accesses a region in the previous frame pointed to by the vector for data replenishment. Region size can vary among the codecs using motion estimation but is typically 16 by 16.

Frame/Block Skipping One technique for reducing data is to eliminate it entirely. In a teleconferencing situation, for example, if the scene does not change (above some threshold criteria), it may be acceptable to not send the new frame (drop or skip the frame). Alternatively, if bandwidth is limited and image quality is important, it may be necessary to drop frames to stay within a bit-rate budget. Most codecs used in teleconferencing applications have the ability of temporal subsampling and are able to gracefully degrade under limited bandwidth situations by dropping frames.

A second form of data elimination is spatial subsampling. The idea is similar to chrominance subsampling discussed previously. In most transform-based codecs, a block (8 by 8 or 16 by 16) is usually skipped if the difference between it and the previous block is below a predetermined threshold. The decompressor may reconstruct the missing pixels by using the previous block to predict the current block.

Entropy Encoding Entropy encoding is a form of statistical coding that provides lossless compression by coding input samples according to their frequency of occurrence. The two methods used most frequently include Huffman coding and run-length encoding.[6] Huffman coding assigns fewer bits to most frequently occurring symbols and more bits to the symbols that appear less often. Optimal Huffman tables can be generated if the source statistics are known. Calculating these statistics, however, slows down the compression process. Consequently, predeveloped tables that have been tested over a wide range of source images are used. A second and simpler method of entropy encoding is run-length encoding in which sequences of identical digits are replaced with the digit and the number in the sequence. Like motion estimation, entropy encoding puts a heavier burden on the compressor than the decompressor.

Before ending this section, we would like to mention that a number of other techniques, including object-based coding, model-based coding, segmentation-based coding, contour-texture oriented coding, fractal coding, and wavelet coding are also available to the codec designer. Thus far, our coverage has concentrated on explaining only those techniques that have been used in the video compression schemes currently supported by Digital. In the next section, we describe some hybrid schemes that employ a number of the techniques described above; these schemes are the basis of several international video coding standards.

Overview of Popular Video Compression Schemes

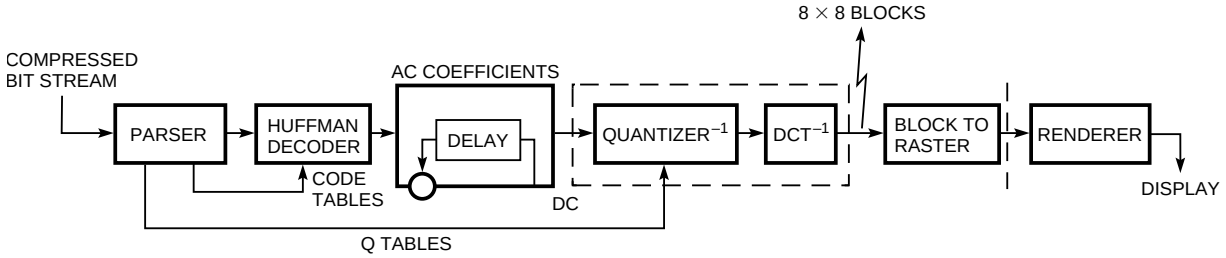
The compression schemes presented in this section can be collectively classified as first-generation video coding schemes.[7] The common assumption in all these methods is that there is statistical correlation between

pixels. Each of these methods attempts to exploit this correlation by employing redundancy reduction techniques to achieve compression.

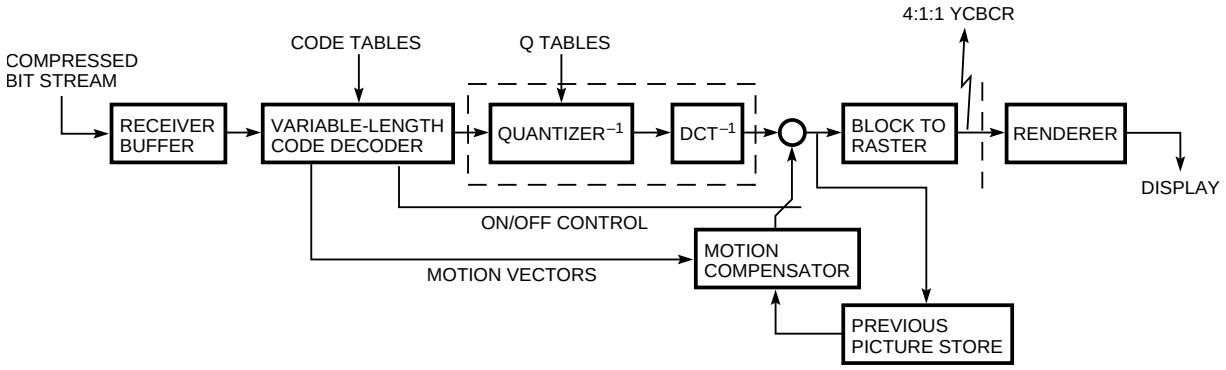
Motion-JPEG Algorithm Motion-JPEG (or M-JPEG) compresses each frame of a video sequence using the ISO's Joint Photographic Experts Group (JPEG) continuous-tone, still-image compression standard.[8] As such, it is an intraframe compression scheme. It is not wed to any particular subsampling format, image color space, or image dimensions, but most typically 4:2:2 subsampled YCbCr, source input format (SIF, 352 by 240) data is used. The JPEG standard specifies both lossy and lossless compression schemes. For video, only the lossy baseline DCT coding scheme has gained acceptance. The scheme relies on selective quantization of the frequency coefficients followed by Huffman and run-length encoding for its compression. The standard defines a bit-stream format that contains both the compressed data stream and coding parameters such as the number of components, quantization tables, Huffman tables, and sampling factors. Popular M-JPEG file formats usually build on top of the JPEG-specified formats with little or no modification. For example, Microsoft's audio-video interleaved (AVI) format encapsulates each JPEG frame with its associated audio and adds an index to the start of each frame at the end of the file. Video editing on a frame-by-frame basis is possible with this format. Another advantage is frame-limited error propagation in networked, distributed applications. Many video digitizer boards incorporate JPEG compression in hardware to compress and decompress video in real time. Digital's Sound & Motion J300 and FullVideo Supreme JPEG are two such boards.[9,10] The baseline JPEG codec is a symmetric algorithm as may be seen in Figure 2a and Figure 3.

ITU-T's Recommendation H.261 The ITU-T's Recommendation H.261 is a motion-compensated, DCT-based video coding standard.[11] Designed for the teleconferencing market and developed primarily for low-bit-rate Integrated Services Digital Network (ISDN) services, H.261 shares similarities with ISO's JPEG still-image compression standard. The target bit rate is $p \times 64$ kilobits per second with p ranging between 1 and 30 (H.261 is also known as $p \times 64$). Only two frame resolutions, common intermediate format (CIF, 352 by 288) and quarter-CIF (QCIF, 176 by 144), are allowed. All standard-compliant codecs must be able to operate with QCIF; CIF is optional. The input color space is fixed by the International Radio Consultative Committee (CCIR) 601 YCbCr standard's with 4:2:0 subsampling (subsampling of chrominance components by 2:1 in both the horizontal and the vertical direction). Two types of frames are defined: key frames that are coded independently and non–key frames that are coded with respect to a previous frame. Key frames are coded in a manner similar to JPEG. For non–key frames, block-based motion compensation is performed to compute interframe differences, which are then DCT coded and quantized. The block size is 16 by 16, and each block can have a different quantization table. Finally, a variable word-length encoder (usually employing Huffman and run-length methods) is used for coding the quantized coefficients. Rate control is done by dropping frames, skipping blocks, and increasing quantization. Error correction codes are embedded in the bit stream to help detect and possibly correct transmission errors. Figure 2b shows a block diagram of an H.261 decompressor.

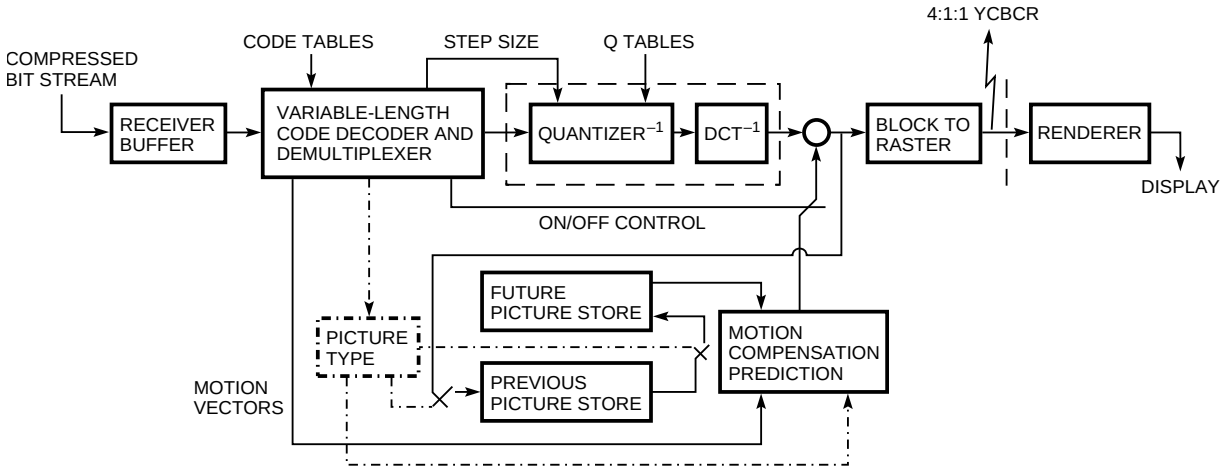
ISO's MPEG-1 Video Standard The MPEG-1 video standard was developed by ISO's Motion Picture Experts Group (MPEG). Like the H.261 algorithm, MPEG-1 is also an interframe video codec that removes spatial redundancy by compressing key frames using techniques similar to JPEG and removes temporal redundancy through motion estimation and compensation.[11,12] The standard defines three different types of frames or pictures: intra or I-frames that are compressed independently; predictive or P-frames that use motion compensation from the previous I- or P-frame; and bidirectional or B-frames that contain blocks predicted from either a preceding or following P- or I-frame (or interpolated from both). Compression is greatest for B-frames and least for I-frames. (A fourth type of frame, called the D-frame or the DC-intracoded frame, is also defined for improving fast-forward-type access, but it is hardly ever used.) There is no restriction on the input frame dimensions, though the target bit rate of 1.5 megabits per second is for video containing SIF frames. Subsampling is fixed at 4:2:0. MPEG-1 employs adaptive quantization of DCT coefficients for compressing I-frames and for compressing the difference between actual and predicted blocks in P- and B-frames. A 16-by-16 sliding window, called a macroblock, is used in motion estimation; and a variable word-length encoder is used in the final step to further lower the output bit rate. The full MPEG-1 standard specifies a system stream that includes a video and an audio substream, along with timing information needed for synchronization between the two. The video substream contains the compressed video data and coding parameters such as picture rate, bit rate, and image size. MPEG-1 has become increasingly popular primarily because it offers better compression than JPEG without compromising on quality. Several vendors and chip manufacturers offer specialized hardware for MPEG compression and decompression. Figure 2c shows a block diagram of an MPEG-1 video decompressor.



(a) Baseline JPEG Decompressor (ISO Standard, 1992)



(b) Recommendation H.261 Decompressor (ITU-T Standard, 1990)



(c) MPEG-1 Video Decompressor (ISO Standard, 1994)

Figure 2
Playback Configurations for Compressed Video Streams

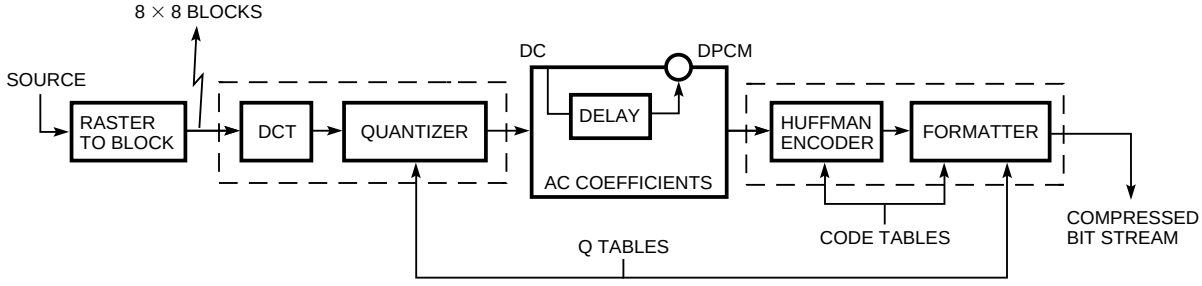


Figure 3
ISO's Baseline JPEG Compressor

Intel's INDEO Video Compression Algorithm Intel's proprietary INDEO video compression algorithm is used primarily for video presentations on personal computer (PC) desktops. It employs color subsampling, pixel differencing, run-length encoding, vector quantization, and variable word-length encoding. The chrominance components are heavily subsampled. For every block of 4-by-4 luminance samples, there is a single sample of Cb and Cr. Furthermore, samples are shifted one bit to convert them to 7-bit values. The resulting precompression format is called YVU9, because on average there are 9 bits per pixel. This subsampling alone yields a reduction of 9/24. Run-length encoding is employed to encode any run of zero pixel differences.

PCWG's INDEO-C Video Compression Algorithm INDEO-C is the video compression component of a teleconferencing system derived from the Personal Conferencing Specification developed by the Personal Conferencing Work Group (PCWG), an industry group led by Intel Corporation. Like the MPEG standard, the PCWG specification defines the compressed bit stream and the decoder but not the encoder. INDEO-C is optimized for low-bit-rate, ISDN-based connections and, unlike its desktop compression cousin, is transform-based. It is an interframe algorithm that uses motion estimation and a 4:1 chrominance subsampling in both directions. Spatial and temporal loop filters are used to remove high-frequency artifacts. The transform used for converting spatial data to frequency coefficients is the slant transform, which has the advantage of requiring only shifts and adds with no multiplies. Like the DCT, the fast slant transform (FST) is applied on image subblocks for coding both intraframes and difference frames. As was the case in other codecs, run-length coding and Huffman coding are employed in the final step. Compression and decompression of video in software is faster than other interframe schemes like MPEG-1 and H.261.

Compression Schemes under Development In addition to the five compression schemes described in this section, four other video compression standards, which are currently in various stages of development within ISO and ITU-T, are worth mentioning: ISO's MPEG-2, ITU-T's Recommendation H.262, ITU-T's Recommendation H.263, and ISO's MPEG-4.[13,14] Although the techniques employed in MPEG-2, H.262, and H.263 compression schemes are similar to the ones discussed above, the target applications are different. H.263 focuses on providing low-bit-rate video (below 64 kilobits per second) that can be transmitted over narrowband channels and used for real-time conversational services. The codec would be employed over the plain old telephone system (POTS) with modems that have the V.32 and the V.34 modem technologies. MPEG-2, on the other hand, is aimed at bit rates above 2 megabits per second, which support a wide variety of formats for multimedia applications that require better quality than MPEG-1 can achieve. One of the more popular target applications for MPEG-2 is coding for high-definition television (HDTV). It is expected that ITU-T will adapt MPEG-2 so that Recommendation H.262 will be very similar, if not identical, to it. Finally, like Recommendation H.263, ISO's MPEG-4's charter is to develop a generic video coding algorithm for low-bit-rate multimedia applications over a public switched telephone network (PSTN). A wide variety of applications, including those operating over error-prone radio channels, are being targeted. The standard is expected to embrace coding methods that are very different from its precursors and will include the so-called second-generation coding techniques.[7] MPEG-4 is expected to reach draft stage by November 1997.

This ends our discussion on video compression techniques and standards. In the next section, we turn our attention to the other component of the video playback solution, namely video rendering. We describe the general process of video rendering and present a novel algorithm for efficient mapping of out-of-range colors to feasible red, green, and blue (RGB) values that can be represented on the target display device. Out-of-range colors can occur when the display quality is adjusted during video playback.

Video Presentation

Video presentation or rendering is the second important component in the video playback pipeline (see Figure 1). The job of this subsystem is to accept decompressed video data and present it in a window of specified size on the display device using a specified number of colors. The basic components are sketched in Figure 4 and described in more detail in a previous issue of this *Journal*.[15] Today, most desktop systems do not include hardware options to perform these steps, but some interesting cases are available as described in this issue.[9,16] When such accelerators are not available, software-only implementation is necessary. Software

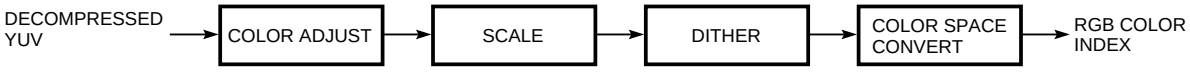


Figure 4
Components of Video Rendering

rendering algorithms, although very efficient, can still consume as many computation cycles as are used to decompress the data.

All major video standards represent image data in a luminance-chrominance color space. In this scheme, each pixel is composed of a single luminance component, denoted as Y, and two chrominance components that are sometimes referred to as color difference signals Cb and Cr, or signals U and V. The relationship between the familiar RGB color space and YUV can be described by a 3-by-3 linear transformation:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \mathbf{M} \begin{bmatrix} y \\ u \\ v \end{bmatrix}, \tag{2}$$

where the transformation matrix,

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & a \\ 1 & b & c \\ 1 & d & 0 \end{bmatrix}. \tag{3}$$

The matrix is somewhat simple with only four values that are not 0 or 1. These constants are $a = 1.402$, $b = -.344$, $c = -.714$, and $d = 1.722$.

The RGB color space cube becomes a parallelepiped in YUV space. This is pictured in Figure 5, where the black corner is at the bottom, and the white corner is at the top; the red, green, and blue corners are as labeled. The chrominance signals U and V are usually subsampled, so the rendering subsystem must first restore these components and then transform the YUV triplets to RGB values.

Typical frame buffers are configured with 8 bits of color depth. This hardware colormap must, in general, be shared by multiple applications, which puts a premium on each of the 256 color slots in the map. Each application, therefore, must be able to request rendering to a limited number of colors. This can be accomplished most effectively with a multilevel dithering scheme, as represented by the dither block in Figure 4.

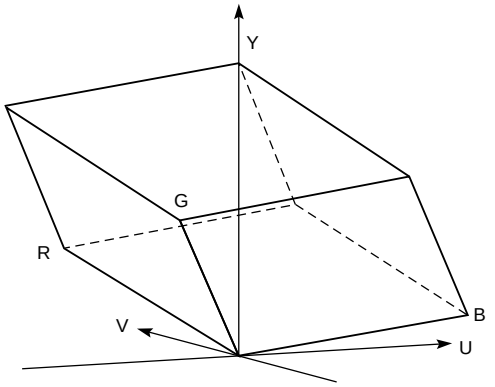


Figure 5
The RGB "Cube" in YUV Space

The color adjustment block controls brightness, contrast and saturation by means of simple look-up tables.

Along with up-sampling the chrominance, the scale block in Figure 4 can also change the size of the image. Although arbitrary scaling is best performed in combination with filtering, it is found to be too expensive to do in a software-only implementation. For the case of enlargement, a trade-off can be made between image quality and speed; contrary to what is shown in Figure 4, image enlargement can occur after dithering and color space converting. Of course, this would result in scaled dithered pixels, which are certainly less desirable, but it would also result in faster processing.

To optimize computational efficiency, color space conversion from YUV to RGB takes place after YUV dithering. Dithering greatly reduces the number of YUV triplets, thus allowing a single look-up table to perform the color space conversion to RGB as well as map to the final 8-bit color index required by the graphics display system. Digital pioneered this idea and has used it in a number of hardware and software-only products.[17]

Mapping Out-of-Range Colors

Besides the obvious advantages of speed and simplicity, using a look-up table to convert dithered YUV values to RGB values has the added feature of allowing careful mapping of out-of-range YUV values. Referring again to Figure 5, the RGB solid describes those *r, g,* and *b* values that are feasible, that is, have the normalized range $0 \le r, g, b \le 1$. The range of possible values in YUV space are those for $0 \le y \le 1$ and $-.5 \le u, v \le .5$. It turns out that the RGB solid occupies only 23.3 percent of this possible YUV space; thus there is ample possibility for so-called infeasible or out-of-range colors to occur. Truncating the *r, g,* and *b* values of these colors has the effect of mapping back to the RGB parallelepiped along lines perpendicular to its nearest surface; this is undesirable since it will result in changing both the hue angle or polar orientation in the chrominance plane and the luminance value. By storing the mapping in a look-up table, decisions can be made a priori as to exactly what values the out-of-range values should map to.

There is a mapping where both the luminance or *y* value and the hue angle are held constant at the expense of a change in saturation. This section details how a closed-form solution can be found for such a mapping. Figure 6 is a cross section of the volume in Figure 5 through a plane at $y = y_0$. The object is to find the point on the surface of the RGB parallelepiped that maps the out-of-range point (y_0, u_0, v_0) in the plane of constant y_0 (constant luminance) and along a straight line to the *u-v* origin (constant hue angle). The solution is the intersection of the closest RGB surface and the line between (y_0, u_0, v_0) and $(y_0, 0, 0)$. This line can

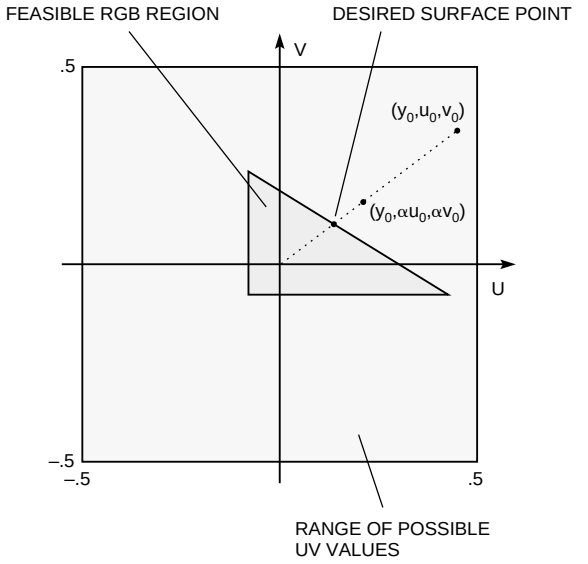


Figure 6
Mapping Out-of-Range YUV Points to the Surface of the RGB Parallelepiped in a Plane of Constant y_0

be parametrically represented as the locus $(y_0, \alpha u_0, \alpha v_0)$ for a single parameter α. The RGB values for these points are

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \mathbf{M} \begin{bmatrix} y_0 \\ \alpha u_0 \\ \alpha v_0 \end{bmatrix} = \begin{bmatrix} \alpha(av_0)+y_0 \\ \alpha(bu_0+cv_0)+y_0 \\ \alpha(du_0)+y_0 \end{bmatrix}, \quad (4)$$

where the matrix **M** is as given in equation (2). To find where this parametric line will intersect the RGB parallelepiped, we can first solve for the α at the intercept values at each of the six bounding surface planes as follows:

Surface Plane	Intercept Value
$r=1$	$\alpha_1=(1-y_0)/av_0$
$g=1$	$\alpha_2=(1-y_0)/(bu_0+cv_0)$
$b=1$	$\alpha_3=(1-y_0)/du_0$
$r=0$	$\alpha_4=(\alpha_1-1)$
$g=0$	$\alpha_5=(\alpha_2-1)$
$b=0$	$\alpha_6=(\alpha_3-1)$

Exactly three α_i will be negative, with each describing the intercept with extended RGB surface planes opposite the *u-v* origin. Of the remaining three α_i, the two largest values will describe intercepts with extended RGB surface planes in infeasible RGB space. This is because the RGB volume, a parallelepiped, is a convex polyhedron. Thus the solution must simply be the smallest positive α_i. Plugging this value of α into equation (4) produces the desired RGB value.

The Software Video Library

When we started this project, we had two objectives in mind: to showcase the processing power of Digital's newly developed Alpha processor and to use this power to make digital video easily available to developers and end users by providing extremely low-cost solutions. We knew that because of the compute-intensive nature of video processing, Digital's Alpha processor would outperform any competitive processor in a head-to-head match. By providing the ability to manipulate good-quality desktop video without the need for additional hardware, we wanted to make Alpha-based systems the computers of choice for end users who wanted to incorporate multimedia into their applications.

Our objectives translated to the creation of a software video library that became a reality because of three key observations. The first one is embedded in our motivation: processors had become powerful enough to perform complex signal-processing operations at real-time rates. With the potential of even greater speeds in the near future, low-cost multimedia solutions would be possible since audio and video decompression could be done on the native processor without any additional hardware.

A second observation was that multiple emerging audio/video compression standards, both formal and industry de facto, were gaining popularity with application vendors and hence needed to be supported on Digital's platforms. On careful examination of the compression algorithms, we observed that most of the prominent schemes used common building blocks (see Figure 2). For example, all five international standards—JPEG, MPEG-1, MPEG-2, H.261, and H.263—have DCT-based transform coders followed by a quantizer. Similarly, all five use Huffman coding in their final step. This meant that work done on one codec could be reused for others.

A third observation was that the most common component of video-based applications was video playback (for example, videoconferencing, video-on-demand, video player, and desktop television). The output decompressed streams from the various decoders have to be software-rendered for display on systems that do not have support for color space conversion and dithering in their graphics adapters. An efficient software rendering scheme could thus be shared by all video players.

With these observations in mind, we developed a software video library containing quality implementations of ISO, ITU-T, and industry de facto video coding standards. In the sections to follow, we present the architecture, implementation, optimization, and performance of the software video library. We complete our presentation by describing examples of video-based applications written on top of this library,

including a novel video screen saver we call Video Odyssey and a software-only video player.

Architecture

Keeping in mind the observations outlined above, we designed a software video library (SLIB) that would

- Provide a common architecture under which multiple audio and video codecs and renderers could be accessed
- Be the lowest, functionally complete layer in the software video codec hierarchy
- Be fast, extensible, and thread-safe, providing reentrant code with minimal overhead
- Provide an intuitive, simple, flexible, and extensible application programming interface (API) that supports a client-server model of multimedia computing
- Provide an API that would accommodate multiple upper layers, allowing for easy and seamless integration into Digital's multimedia products

Our intention was not to create a library that would be exposed to end-user applications but to create one that would provide a common architecture for video codecs for easy integration into Digital's multimedia products. SLIB's API was purposely designed to be a superset of Digital's Multimedia Services' API for greater flexibility in terms of algorithmic tuning and control. The library would fit well under the actual programming interface provided to end users by Digital's Multimedia Services. Digital's Multimedia API is the same as Microsoft's Video For Windows API, which facilitates the porting of multimedia applications from Windows and Windows NT to Digital UNIX and OpenVMS platforms. Figure 7 shows SLIB in relation to Digital's multimedia software hierarchy. The shaded regions indicate the topics discussed in this paper.

As mentioned, the library contains routines for audio and video codecs and Digital's propriety video-rendering algorithms. The routines are optimized both algorithmically and for the particular platform on which they are offered. The software has been successfully implemented on multiple platforms, including the Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems.

Three classes of routines are provided for the three subsystems: (1) video compression and decompression, (2) video rendering, and (3) audio processing. For each subsystem, routines can be further classified as (a) setup routines, (b) action routines, (c) query routines, and (d) teardown routines. Setup routines create and initialize all relevant internal data structures. They also compute values for the various look-up tables such as the ones used by the rendering subsystem. Action routines perform the actual coding, decoding, and rendering operations. Query routines may be used before setup or between action routines. These provide the programmer with information about the capability

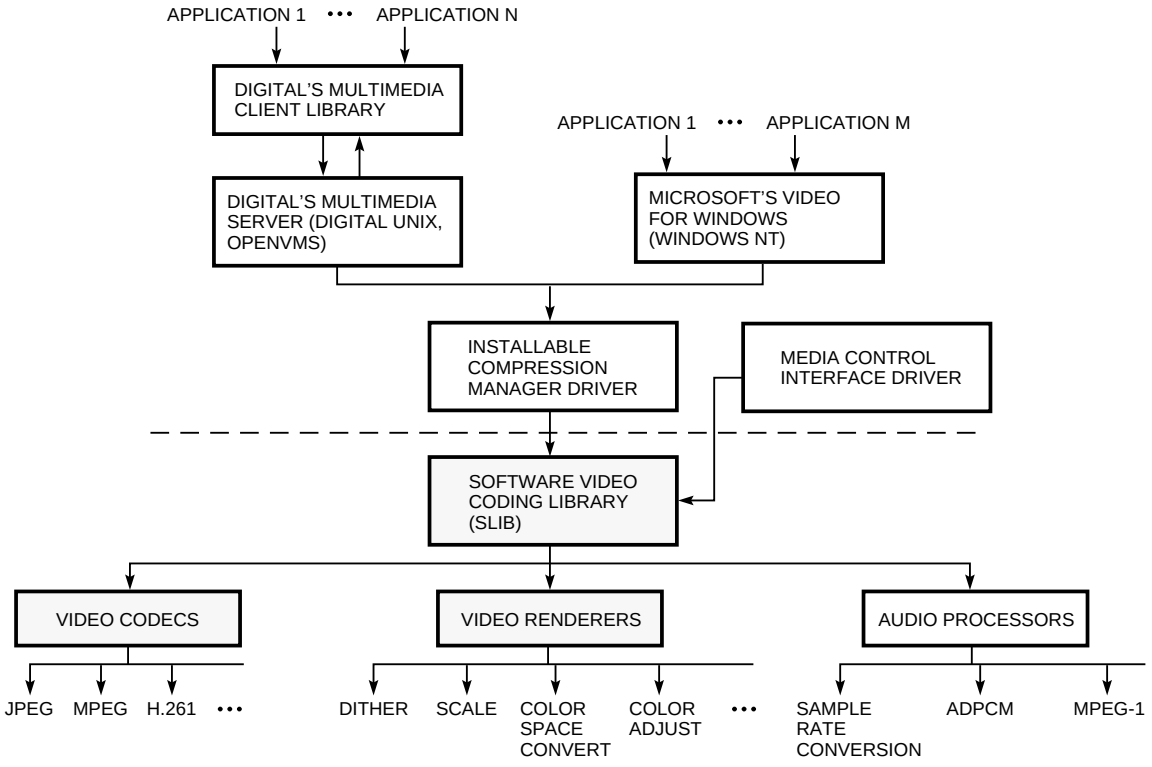


Figure 7
Software Video Library Hierarchy

of the codec such as whether or not it can handle a particular input format and provide information about the bit stream being processed. These routines can also be used for gathering statistics. Teardown routines, as the name suggests, are used for closing the codec and destroying all internal memory (state information) associated with it. For all video codecs, SLIB provides convenience functions to construct a table of contents containing the offsets to the start of frames in the input bit stream. These convenience functions are useful for short clips: once a table of contents is built, random access and other VCR functions can be implemented easily. (These routines are discussed further in the section on sample applications.)

Implementation of Video Codecs

In this section, we present the program flow for multimedia applications that incorporate the various video codecs. These applications are built on top of SLIB. We also discuss specific calls from the library's API to explain concepts.

Motion JPEG Motion JPEG is the de facto name of the compression scheme that uses the JPEG compression algorithm developed for still images to code video sequences. The motion JPEG (or M-JPEG) player was the first decompressor we developed. We had recently completed the Sound & Motion J300 adapter that could perform JPEG compression, decompression, and dithering in hardware.[9,10] We now wanted to develop a software decoder that would be able to decode video sequences produced by the J300 and its successor, the FullVideo Supreme JPEG adapter, which uses the peripheral component interconnect (PCI).[10] Only baseline JPEG compression and decompression have been implemented in SLIB. This is sufficient for greater than 90 percent of today's existing applications. Figure 2a and Figure 3 show the block diagrams for the baseline JPEG codec, and Figure 8 shows the flow control for compressing raw video using the video library routines. Due to the symmetric structure of the algorithm, the flow diagram for the JPEG decompressor looks very similar to the one for the JPEG compressor.

The amount of compression is controlled by the amount of quantization in the individual image frames constituting the video sequence. The coefficients for every 8-by-8 block within the image $F(x,y)$ are quantized and dequantized as

$$\begin{aligned} \mathrm{F}_q(x,y) &= \left\lfloor \frac{F(x,y)}{\mathrm{QTable}(x,y)} \right\rfloor \mathrm{F}(x,y) \\ &= \mathrm{F}_q(x,y) \times \mathrm{QTable}(x,y). \end{aligned} \tag{5}$$

In equation (5), QTable represents the quantization matrices, also called visibility matrices, associated with the frame $F(x,y)$. (Each component constituting the frame can have its own QTable.) SLIB provides routines to download QTables to the encoder explicitly; tables provided in the ISO specification can be used as defaults. The library provides a quality factor that can scale the base quantization tables, thus providing a control knob mechanism for varying the amount of compression from frame to frame. The quality factor may be dynamically varied between 0 and 10,000, with a value of 10,000 causing no quantization (all quantization table elements are equal to 1), and a value of 0 resulting in maximum quantization (all quantization table elements are equal to 255). For intermediate values:

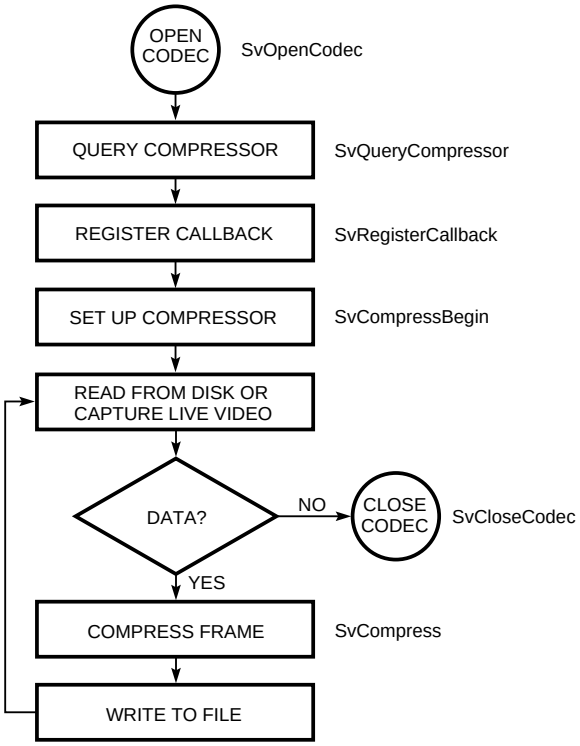


Figure 8
Flow Control for M-JPEG Compression

$$\mathrm{QTable}(x,y) = \tag{6}$$
$$Clip\left(\left\lfloor \frac{VisibilityTable(x,y)\times(10^4 - QualFactor)\times 255}{10^4 \times \min\left(VisibilityTable(x,y)\right)} \right\rfloor\right).$$

The Clip() function forces the out-of-bounds values to be either 255 or 1. At the low end of the quality setting (small values of the quality factor), the above formula produces quantization tables that cause noticeable artifacts.

Although Huffman tables do not affect the quality of the video, they do influence the achievable bit rate for a given video quality. As with quantization tables, SLIB provides routines for loading and using custom Huffman tables for compression. Huffman coding works best when the source statistics are known; in

practice, statistically optimized Huffman tables are rarely used due to the computational overhead involved in their generation. In the case where these tables are not explicitly provided, the library uses as default the baseline tables suggested in the ISO specification. In the case of decompression, the tables may be present in the compressed bit stream and can be examined by invoking appropriate query calls. In the AVI format, Huffman tables are not present in the compressed bit stream, and the default ISO tables are always used.

Query routines for determining the supported input and output formats for a particular compressor are also provided. For M-JPEG compression, some of the supported input formats include interleaved 4:2:2 YUV, noninterleaved 4:2:2 YUV, interleaved and noninterleaved RGB, 32-bit RGB, and single component (monochrome). The supported output formats include JPEG-compressed YUV and JPEG-compressed single component.

ISO's MPEG-1 Video Once we had implemented the M-JPEG codec, we turned our attention to the MPEG-1 decoder. MPEG-1 is a highly asymmetric algorithm. The committee developing this standard purposely kept the decompressor simple: it was expected that there would be many cases of compress once and decompress multiple times. In general, the task of compression is much more complex than that of decompression. As of this writing, achieving real-time performance for MPEG-1 compression in software is not possible. Thus we concentrated our energies on implementing and optimizing an MPEG-1 decompressor while leaving MPEG-1 compression for batch mode. Someday we hope to achieve real-time compression all in software with the Alpha processor. Figure 9 illustrates the high-level scheme of how SLIB fits into an MPEG player. The MPEG-1 system stream is split into its audio and video substreams, and each is handled separately by the different components of the video library. Synchronization between audio and video is achieved at the application layer by using the presentation time-stamp information embedded in the system stream. A timing controller module within the application can adjust the rate at which video packets are presented to the SLIB video decoder and renderer. It can indicate to the decoder whether to skip the decoding of B- and P-frames.

Figure 10 illustrates the flow control for an MPEG-1 video player written on top of SLIB. The scheme relies on a callback function that is registered with the codec during initial setup, and a SvAddBuffers function, written by the client, which provides the codec with the bitstream data to be processed. The codec is primed by adding multiple buffers, each typically containing a single video packet from the demultiplexed system stream. These buffers are added to the codec's internal buffer queue. After enough data has been provided, the decoder is told to parse the bit stream in its buffer queue until it finds the next (first) picture. The client application can specify which type of picture to locate (I, P, or B) by setting a mask bit. After the picture is found and its information returned to the client, the client may choose to either decompress this picture or to skip it by invoking the routine to find the next picture. This provides an effective mechanism for rate control and for VCR controls such as step forward, fast forward, step back, and fast reverse. If the client requests that a non–key picture (P or B) be decompressed and the codec does not have the required reference (I or P) pictures needed to perform this operation, an error is returned. The client can then choose to abort or proceed until the codec finds a picture it can decompress.

During steady state, the codec may periodically invoke the callback function to exchange messages with the client application as it compresses or decompresses the bit stream. Most messages sent by the codec expect some action from the client. For example, one of the messages sent by the codec to the application is

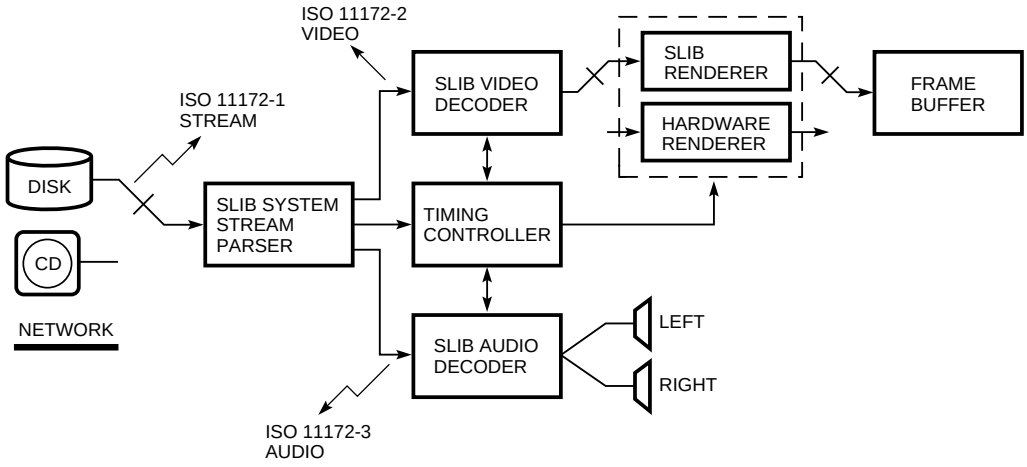


Figure 9
SLIB as Part of a Full MPEG Player

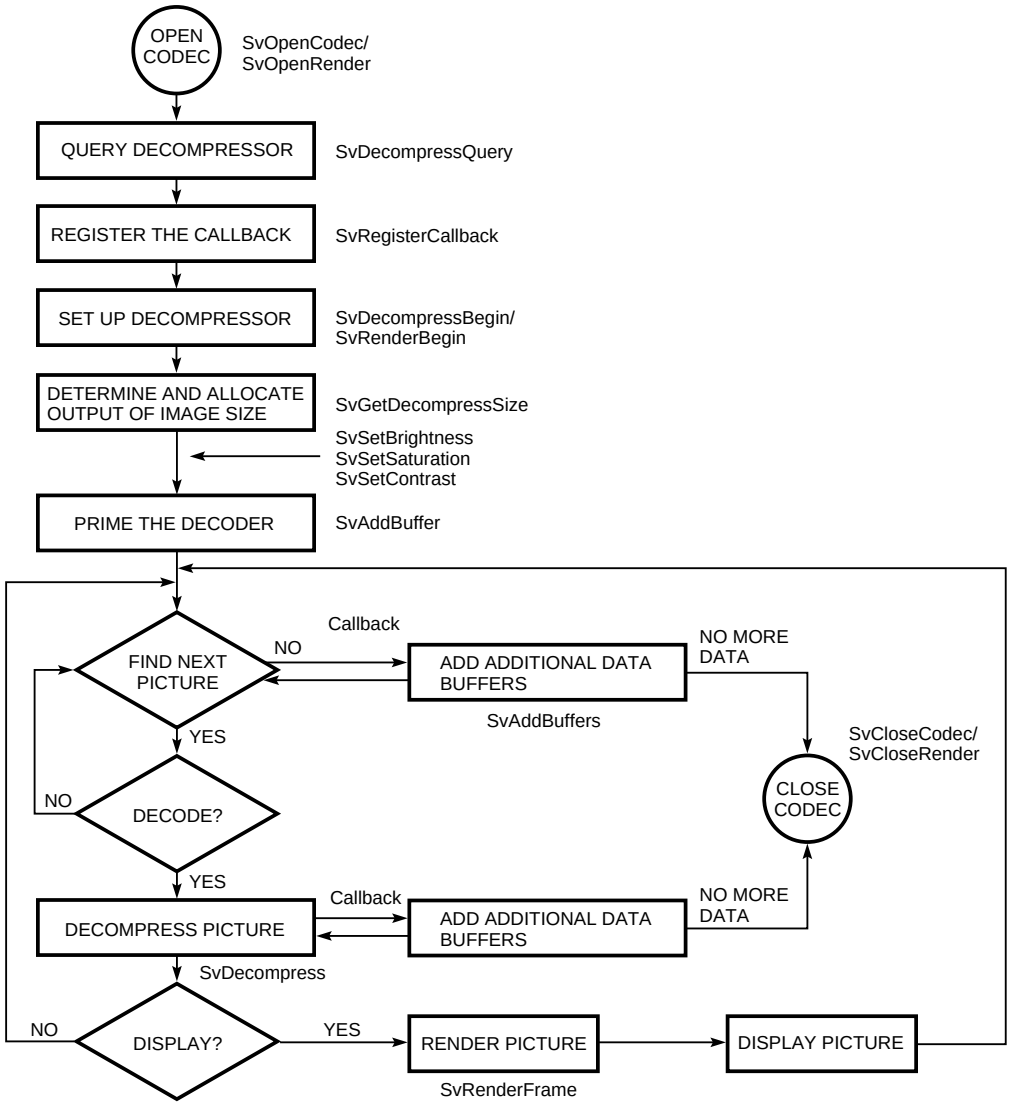


Figure 10
Flow Control for MPEG-1 Video Playback

a CB_END_BUFFERS message, which indicates the codec has run out of data and the client needs to either add more data buffers or abort the operation. Another message, CB_RELEASE_BUFFERS, indicates the codec is done processing the bit-stream data in a data buffer, and the buffer is available for client reuse. One possible action for the client is to fill this newly available buffer with more data and pass it back to the codec. In the other direction, the client may send messages to the codec through a ClientAction field. Table 1 gives some of the messages that can be sent to the codec by the application.

Another use for the callback mechanism is to accommodate client operations that need to be intermixed between video encoding/decoding operations. For example, the application may want to process audio samples while it is decompressing video. The codec can then be configured such that the callback function is invoked at a (near) periodic rate. A CB_PROCESSING message is sent to the application by the codec at regular intervals to give it an opportunity for rate control of video and/or to perform other operations.

Table 1
List of Client Messages

Message	Interpretation
CLIENT_ABORT	Abort processing of the frame
CLIENT_CONTINUE	Continue processing the frame
CLIENT_DROP	Do not decompress
CLIENT_PROCESS	Start processing

Typically the order in which coded pictures are presented to the decoder does not correspond to the order in which they are to be displayed. Consider the following example:

Display Order	I1	B2	B3	P4	B5	B6	P7	B8
Decoder Input	I1	P4	B2	B3	P7	B5	B6	I10

The order mismatch is an artifact of the compression algorithm—a B-picture cannot be decoded until both its past and future reference frames have been decoded. Similarly a P-picture cannot be decoded until its past reference frame has been decoded. To get around this problem, SLIB defines an output multibuffer. The size of this multibuffer is approximately equal to three times the size of a single uncompressed frame. For example, for a 4:2:0 subsampled CIF image, the size of the multibuffer would be 352 by 288 by 1.5 by 3 bytes (the exact size is returned by the library during initial codec setup). After steady state has been reached, each invocation to the decompress call yields the correct next frame to be displayed as shown in Figure 11. To avoid expensive copy operations, the multibuffer is allocated and owned by the software above SLIB.

ITU-T's Recommendation H.261 (a.k.a. $p \times 64$) At the library level, decompressing an H.261 stream is very similar to MPEG-1 decoding with one exception: instead of three types of pictures, the H.261 recommendation defines only two, key frames and non–key frames (no bidirectional prediction). The implication for implementation is that the size of the multibuffer is approximately twice the size of a single decompressed frame. Furthermore, the order in which compressed frames are presented to the decompressor is the same as the order in which they are to be displayed.

To satisfy the H.261 recommendation, SLIB implements a streaming interface for compression and decompression. In this model, the application feeds input buffers to the codec, which processes the data in the buffers and returns the processed data to the application through a callback routine. During decompression, the application layer passes input buffers containing sections of an H.261 bit stream. The bit stream can be divided arbitrarily, or, in the case of live teleconferencing, each buffer can contain data from a transmission packet. Empty output buffers are also passed to the codec to fill with reconstructed images. Picture frames do not have to be aligned on buffer boundaries. The codec parses the bit stream and, when enough data is available, reconstructs an image. Input buffers are freed by calling the callback routine. When an image is reconstructed, it is placed in an output buffer and the buffer is returned to the application through the callback routine. The compression process is similar, but input buffers contain images and output buffers contain bit-stream data. One advantage to this streaming interface is that the application layer does not need to know the syntax of the H.261 bit stream. The codec is responsible for all bit-stream parsing. Another advantage is that the callback mechanism for returning completed images or bit-stream buffers allows the application to do other tasks without implementing multithreading.

SLIB's architecture and API can easily accommodate ISO's MPEG-2 and ITU-T's H.263 video compression algorithms because of their similarity to the MPEG-1 and H.261 algorithms.

Implementation of Video Rendering

Our software implementation of video rendering essentially parallels the hardware realization detailed elsewhere in this issue.[9] As with the hardware implementation, the software renderer is fast and simple because the complicated computations are performed off line in building the various look-up tables. In both hardware and software cases, a shortcut is achieved by dithering in YUV space and then converting to some small number of RGB index values in a look-up table.[16]

Although in most cases the mapping values in the look-up tables remain fixed for the duration of the run, the video library provides routines to dynamically adjust image brightness, contrast, saturation, and the number of colors. Image scaling is possible but affects performance. When quality is important, the software performs scaling before dithering and when speed is the primary concern, it is done after dithering.

Optimizations

We approached the problem of optimization from two directions: Platform-independent optimizations, or algorithmic enhancements, were done by exploiting knowledge of the compression algorithm and the

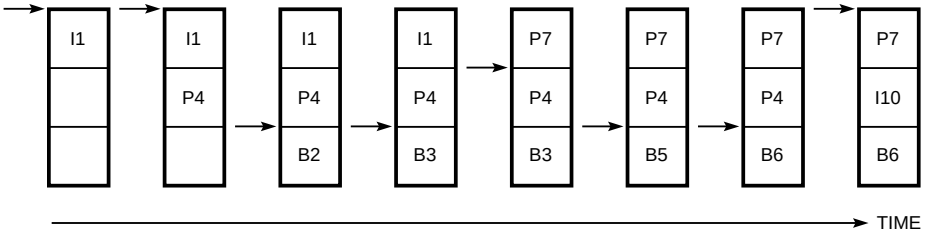


Figure 11
Multibuffering in SLIB

input data stream. Platform-dependent optimizations were done by examining the services available from the underlying operating system and by evaluating the attributes of the system's processor.

As can be seen from Table 2, the DCT is one of the most computationally intensive components in the compression pipeline. It is also common to all five international standards. Therefore, a special effort was made in choosing and optimizing the DCT. Since all five standards call for the inverse DCT (IDCT) to be postprocessed with inverse quantization, significant algorithmic savings were obtained by computing a scalar multiple of the DCT and merging the appropriate scaling into the quantizer. The DCT implemented in the library is a modified version of the one-dimensional scaled DCT proposed by Arari et al.[18] The two-dimensional DCT is obtained by performing a one-dimensional DCT on the columns followed by a one-dimensional DCT on the rows. A total of 80 multiplies and 464 adds are needed for a fully populated 8-by-8 block. In highly compressed video, the coefficient matrix to be transformed is generally sparse because a large number of elements are "zeroed" out due to heavy quantization. We exploit this fact to speed up the DCT computations. In the decoding process, the Huffman decoder computes and passes to the IDCT a list of rows and columns that are all zeros. The IDCT then simply skips these columns.[19] Another optimization uses a different IDCT, depending on the number of nonzero coefficients. The overall speedup due to these techniques is dependent on the amount of compression. For lightly compressed video, we observed that the overhead due to these techniques slowed down the decompressor. We overcame this difficulty by building into SLIB the adaptive selection of the appropriate optimization based on continuous statistics gathering. Run-time statistics of the number of blocks per frame that are all zeros are maintained, and the number of frames over which these statistics are evaluated is provided as a parameter for the client applications. Statistic gathering is minimal: a counter update and an occasional compare.

The second component of the video decoders we looked at was the Huffman decoder. Analysis of the compressed data indicated that short-code-length symbols were a large part of the compressed bit stream. The decoder was written to handle frequently occurring very short codes (< 4 bits) as special cases, thus avoiding loads from memory. For short codes (< 8 bits), look-up tables were used to avoid bit-by-bit decoding. Together, these two classes of codes account for well over 90 percent of the total collection of the variable-length codes.

A third compute-intensive operation is raster-to-block conversion in preparation for compression. This operation had the potential of slowing down the compressor on Alpha-based systems on which byte and short accesses are done indirectly. We implemented an assembly language routine that would read the uncompressed input color image and convert it to three one-dimensional arrays containing 8-by-8 blocks in sequence. Special care was taken to keep memory references aligned. Relevant bytes were obtained through shifting and masking operations. Level shifting was also incorporated within the routine to avoid touching the same data again.

Other enhancements included replacing multiplies and divides with shifts and adds, avoiding integer to floating-point conversions, and using floating-point operations wherever possible. This optimization is particularly suited to the Alpha architecture, where floating-point operations are significantly faster than integer operations. We also worked to reduce memory bandwidth. Ill-placed memory accesses can stall the processor and slow down the computations. Instructions generated by the compiler were analyzed and sometimes rescheduled to void data hazards, to keep the on-chip pipeline full, and to avoid unnecessary loads and stores. Critical and small loops were unrolled to make better use of floating-point pipelines. Reordering the computations to reuse data already in registers and caches helped minimize thrashing in the cache and the translation lookaside buffer. Memory was accessed through offsets rather than pointer

Table 2
Typical Contributions of the Major Components in the Playback of Compressed Video (SIF)

Coding Scheme	Bit-stream Parser	Huffman and Run-length Decoder	Inverse Quantizer	IDCT	Motion Compression, Block to Raster	Vector Quantization (INDEO only)	Tone Adjust, Dither, Quantize and Color Space Convert	Display
M-JPEG decode	0.8%	12.4%	10.5%	35.2%	——	——	33.7%	7.4%
MPEG-1 decode	0.9%	13.0%	9.7%	19.7%	20.2%	——	31.4%	5.1%
INDEO decode	1.0%	——	——	——	——	57.5%	36.0%	5.5%

increments. More local variables than global variables were used. Wherever possible, fixed values were hard coded instead of using variables that would need to be computed. References were made to be 32-bit or 64-bit aligned accesses instead of byte or short.

Consistent with one of the design goals, SLIB was made thread-safe and fully reentrant. The Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems all offer support for multithreaded applications. Applications such as video playback can improve their performance by having separate threads for reading, decompressing, rendering, and displaying. Also, a multithreaded application scales up well on a multiprocessor system. Global multithreading is possible if the library code is reentrant or thread-safe. When we were trying to multithread the library internals, we found that the overhead caused by the birth and death of threads, the increase in memory accesses, and the fragmentation of the codec pipeline caused operations to slow down. For these reasons, routines within SLIB were kept single-threaded. Other operating-system optimizations such as memory locking, priority scheduling, nonpreemption, and faster timers that are generally good for real-time applications were experimented with but not included in our present implementation.

Performance on Digital's Alpha Machines

Measuring the performance of video codecs is generally a difficult problem. In addition to the usual dependencies such as system load, efficiency of the underlying operating system, and application overhead, the speed of the video codecs is dependent on the content of the video sequence being processed. Rapid movement and action scenes can delay both compression and decompression, while slow motion and high-frequency content in a video sequence can generally result in faster decompression. When comparing the performance of one codec against another, the analyst must make certain that all codecs process the same set of video sequences under similar operating conditions. Since no sequences have been accepted as standard, the analyst must decide which sequences are most typical. Choosing a sequence that favors the decompression process and presenting those results is not uncommon, but it can lead to false expectations. Sequences with similar peak signal-to-noise ratio (PSNR) may not be good enough, because more often than not PSNR (or equivalently the mean square error) does not accurately measure signal quality. With these thoughts in mind, we chose some sequences that we thought were typical and used these to measure the performance of our software codecs. We do not present comparative results to codecs implemented elsewhere since we did not have access to these codecs and hence could not test these with the same sequences.

Table 3 presents the characteristics of the three video sequences used in our experiments. Let $L_i(x,y)$ and $\hat{L}_i(x,y)$ represent the luminance component of the original and the reconstructed frame i; let n and m represent the horizontal and vertical dimensions of a frame; and let N be the number of frames in the video sequence. Then the Compression Ratio, the average output BitsPerPixel, and the average PSNR are calculated as

$$\text{Compression Ratio} = \frac{\sum_{i=1}^{N} \text{bits in frame}[i] \text{ of original video}}{\sum_{i=1}^{N} \text{bits in frame}[i] \text{ of compressed video}} \quad (7)$$

$$\text{Avg. BitsPerPixel} = \frac{1}{N \times n \times m} \sum_{i=1}^{N} \text{bits in frame}[i] \text{ of compressed video} \quad (8)$$

$$\text{Avg. PSNR} = 20\log_{10} \frac{255}{\frac{1}{N}\sum_{i=1}^{N} \left(\sqrt{\frac{1}{nm}\sum_{x=1}^{n}\sum_{y=1}^{m} \left[L_i(x,y) - \hat{L}_i(x,y) \right]^2} \right)}. \quad (9)$$

Figure 12 shows the PSNR for individual frames in the video sequences along with the distribution of frame size for each of three test sequences. Frame dimensions within a sequence always remain constant.

Table 4 provides specifications of the workstations and PCs used in our experiments for generating the various performance numbers. The 21064 chip is Digital's first commercially available Alpha processor. It has a load-store architecture, is based on a 0.75-micrometer complementary metal-oxide semiconductor (CMOS) technology, contains 1.68 million transistors, has a 7- and 10-stage integer and floating-point pipeline, has separate 8-kilobyte instruction and data caches, and is designed for dual issue. The 21064A microprocessor has the same architecture as the 21064 but is based on a 0.5-micrometer CMOS technology and supports faster clock rates.

We provide performance numbers for the video sequences characterized in Table 3. Figure 13 provides measured data on CPU usage when compressed video (from Table 3) is played back at 30 frames per second on the various test platforms shown in Table 4. We chose "percentage of CPU used" as a measure of performance because we wanted to know whether the CPU could handle any other tasks when it was doing video processing. Fortunately, it turned out the

Table 3
Characteristics of the Video Sequences Used to Generate the Performance Numbers Shown in Figure 12

Name	Compression Algorithm	Spatial Resolution (width X height)	Temporal Resolution (No. of Frames)	Avg. BitsPerPixel	Compression Ratio	Avg. PSNR (dB)
Sequence 1	M-JPEG	352 × 240	200	0.32	50:1	31.56
Sequence 2	MPEG-1 Video	352 × 288	200	0.17	69:1	32.28
	M-JPEG	352 × 240	200	0.56	28:1	31.56
Sequence 3	INDEO	352 × 240	200	0.16	47:1	28.73

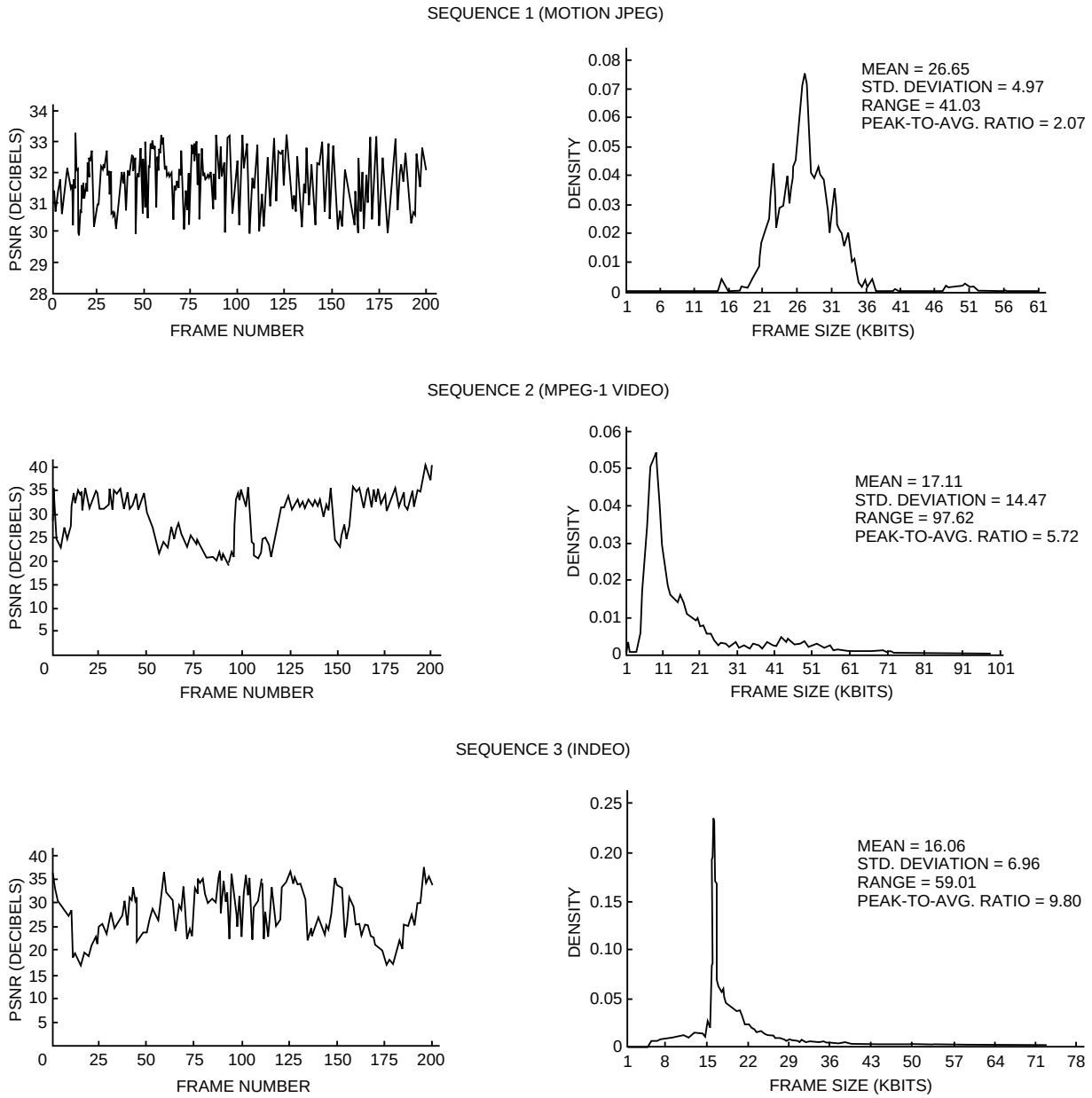


Figure 12
Characteristics of the Three Test Sequences

Table 4
Specifications of Systems Used in Experimentation

System Name	CPU	Bus	Clock Rate	Cache	Memory	Operating System	Disk
AlphaStation 600 5/266 workstation	Alpha 21164A	PCI	266 MHz (3.7 ns)	2 MB	64 MB	Digital UNIX V3.2	RZ28B
AlphaStation 200 4/266 workstation	Alpha 21064A	PCI	266 MHz (3.7 ns)	2 MB	32 MB	Digital UNIX V3.0	RZ58
DEC 3000/M900 workstation	Alpha 21064A	TURBOchannel	275 MHz (3.6 ns)	2 MB	64 MB	Digital UNIX V3.2	RZ58
DEC 3000/M500 workstation	Alpha 21064	TURBOchannel	133 MHz (7.5 ns)	512 KB	32 MB	Digital UNIX V3.0	RZ57

answer was a resounding "Yes" in the case of Alpha processors. The video playback rate was measured with software video rendering enabled. When hardware rendering is available, estimated values for video playback are provided.

From Figure 13, it is clear that today's workstations are capable of playing SIF video at full frame rates with no hardware acceleration. High-quality M-JPEG and MPEG-1 compressed video clips can be played at full speed with 20 percent to 60 percent of the CPU available for other tasks. INDEO decompression is faster than M-JPEG and MPEG due to the absence of DCT processing. (INDEO uses a vector quantization method based on pixel differencing.) On three out of

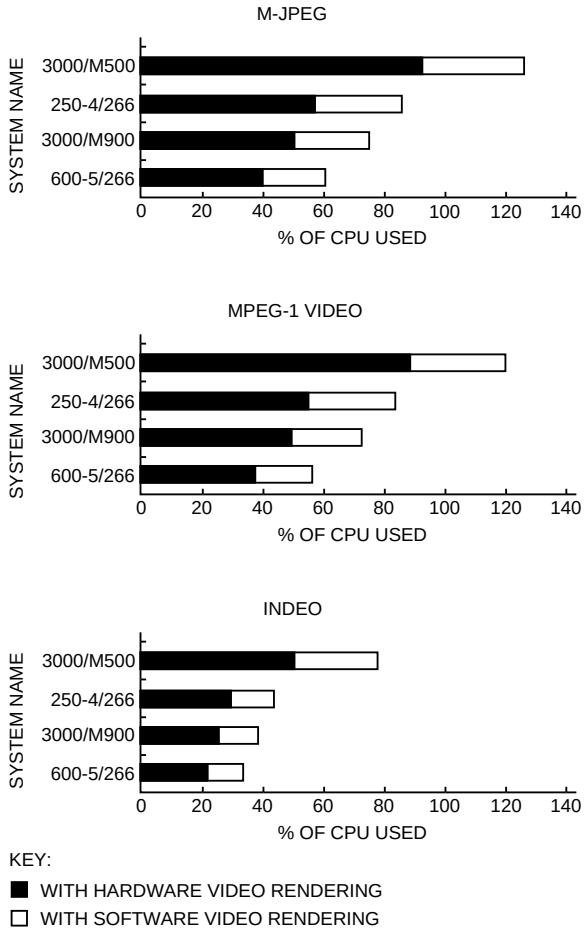


Figure 13
Percentage of CPU Required for Real-time Playback at 30 fps on Four Different Alpha-based Systems

the four machines tested, two SIF INDEO clips could be played back at full speed with CPU capacity left over for other tasks.

The data also shows the advantage of placing the color conversion and rendering of the video in the graphics hardware (see Table 2 and Figure 13). Software rendering accounts for one-third of the total playback time. Since rendering is essentially a table look-up function, it is a good candidate for moving into hardware. If hardware video rendering is available, multiple M-JPEG and MPEG-1 clips can be played back on three of the four machines on which the software was tested.

Software video compression is more time-consuming than decompression. All algorithms discussed in this paper are asymmetric in the amount of processing needed for compression and decompression. Even though the JPEG algorithm is theoretically symmetric, the performance of the JPEG decoder is better than that of the encoder. The difference in performance is due to the sparse nature of the quantized coefficient matrices, which is exploited by the appropriate IDCT optimizations.

For video encoders, we measured the rate of compression for both SIF and quarter SIF (QSIF) formats. Since the overhead due to I/O affects the rate at which the compressor works, we present measured rates collected when the raw video sequence is read from disk and when it is captured in real time. The capture cards used in our experiments were the Sound & Motion J300 (for systems with the TURBOchannel bus) and the FullVideo Supreme (for PCI-based systems). The compressed bit streams were stored as AVI files on local disks. The sequences used in this experiment were the same ones used for obtaining measurement for the various decompressors; their output characteristics are given in Table 3. Table 5 provides performance numbers for the M-JPEG and an unoptimized INDEO compressor. For M-JPEG, rates for both monochrome and color video sequences are provided.

The data in Table 5 indicates that the M-JPEG compression outperforms INDEO (although one has to keep in mind that INDEO was not optimized). This difference occurs because M-JPEG compression, unlike INDEO, does not rely on interframe prediction or motion estimation for compression. Furthermore, when raw video is compressed from disk, the encoder performs better than when it is captured and compressed in real time. This can be explained on the basis of the overhead resulting from context switching in the operating system and the scheduling of sequential capture operation by the applications. Real-time capture and compression of image sizes larger than QSIF still require hardware assistance. It should be noted that in Table 5, the maximum compression rate for real-time capture and compression does not exceed 30 frames per second, which is the limit of the capture hardware. Since there are no such limitations for disk reads, compression rates of greater than 30 frames per second for QSIF sequences are recorded.

With the newer Alpha chip we expect to see improved performance. A factor we neglected in our calculations was prefiltering. Some capture boards are capable of capturing only in CCIR 601 format and do not include decimation filters as part of their hardware. In such cases, the software has to filter each frame down to CIF or QCIF, which adds substantially to the overall compression time. For applications that do not require real-time compression, software digital-video compression may be a viable solution since video can be captured on fast disk arrays and compressed later.

Table 5
Typical Number of Frames Compressed per Second

	M-JPEG (Color)				M-JPEG (Monochrome)				INDEO (Color)			
System	**Compress (fps)**		**Capture and Compress (fps)**		**Compress (fps)**		**Capture and Compress (fps)**		**Compress (fps)**		**Capture and Compress (fps)**	
	SIF	**QSIF**	**SIF**	**QSIF**	**SIF**	**QSIF**	**SIF**	**QSIF**	**SIF**	**QSIF**	**SIF**	**QSIF**
AlphaStation 600 5/266 workstation	21.0	79.4	20.0	30.0	32.8	130	29.0	30.0	8.7	35.4	5.8	23.0
AlphaStation 200 4/266 workstation	10.8	45.1	12.0	30.0	15.8	72.9	20.0	30.0	5.6	22.0	4.2	13.0
DEC 3000/M900 workstation	13.2	56.6	7.9	28.0	21.9	87.8	14.0	29.0	6.0	25.4	4.5	7.6
DEC 3000/M500 workstation	6.7	26.6	7.3	8.1	10.4	40.4	7.4	8.2	2.8	11.8	2.2	8.7

Sample Applications

We implemented several applications to test our architecture (codecs and renderer) and to create a test bed for performance measurements. These programs also served as sample code for software developers incorporating SLIB into other multimedia software layers.

The Video Odyssey Screen Saver

The Video Odyssey screen saver uses software video decompression and 24-bit YCbCr to 8-bit pseudocolor rendering to deliver video images to the screen in a variety of modes. The program is controlled by a control panel, shown in Figure 14.

The user can select from several methods of displaying the decompressed video or let the computer cycle through all methods. The floaters mode, shown in Figure 15, floats one to four copies of the video around the screen with the number of floating windows controlled by a slider in the control panel. The snapshot mode floats one window of the video around the screen, but every second takes a snapshot of a frame and pastes it to the background behind the floating window.

All settings in the control panel are saved in a configuration file in the user's home directory. The user selects a video file with the file button. In the current implementation, any AVI file containing Motion JPEG or raw YUV video is acceptable. The user can set the time interval for the screen saver to take over. Controls for setting brightness, contrast, and saturation are also provided. Video can be played back at normal resolution or with ×2 scaling. Scaling is integrated with the color conversion and dithering for optimization. A pause feature allows the user to leave his or her screen in a locked state with an active screen saver. The screen is unlocked only if the correct password is provided.

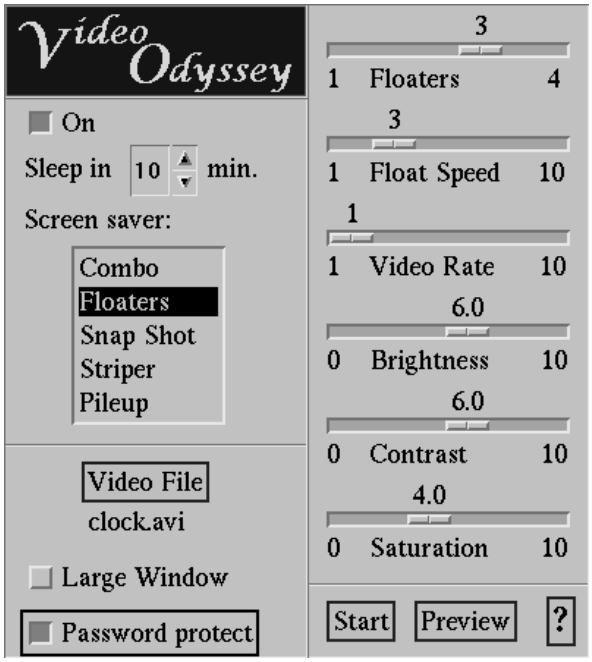


Figure 14
Video Odyssey Control Panel

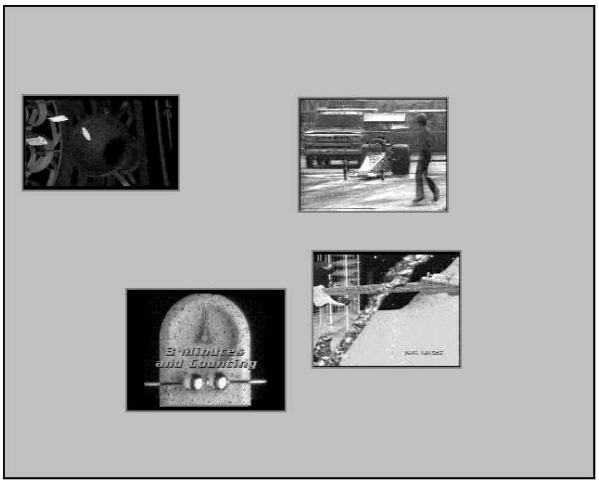


Figure 15
Video Odyssey Screen Saver in Floaters Mode

The Software Video Player

The software video player is an application for viewing video that is similar to a VCR. Like Video Odyssey, the software video player exercises the decompression and rendering portions of SLIB. Unlike Video Odyssey, the software video player allows random access to any portion of the video and permits single-step, reverse, and fast-forward functions. Figure 16 shows the display window of the software video player.

Figure 16
The Software Video Player Display Window

The user moves through the file with a scroll bar and a set of VCR-like buttons. The button on the far left of the display window allows the video to be displayed at normal size or at a magnification of ×2. The far-right button allows adjustment of brightness, contrast, saturation, and number of displayed colors. The quality of the dithering algorithm used in rendering is such that values as low as 25 colors lead to acceptable image quality. Allowable file formats for the software video player are M-JPEG (AVI format and the JPEG file interchange format or JFIF), MPEG-1 (both video and system streams), and raw YUV.

Random access into the file is done in one of two ways, depending on the file format. For formats that contain an index of the frame positions in the file (like AVI files), the index is simply used to seek the desired frame. For formats that do not contain an index, such as MPEG-1 and JFIF, the software video player estimates the location of a frame based on the total length of the video clip and a running average of frame size. This technique is adequate for most video clips and has the advantage of avoiding the time needed to first build an index by scanning through the file.

Interframe compression schemes like MPEG-1 and INDEO pose special problems when trying to access a random frame in a video clip. MPEG-1's B- and P-frames are dependent on preceding frames and cannot be decompressed alone. One technique for handling random access into files with non–key frames and no frame index is to use the file position specified by the user (with a scroll bar or by other means) as a starting point and then to search the bit stream for the next key frame (an I-frame in MPEG-1). At that point, display can proceed normally. Reverse play is also a problem with these formats. The software video player deals with reverse by displaying only the key frames. It could display all frames in reverse by predecompressing all frames in a group and then displaying them in reverse order, but this would require large amounts of memory and would pose problems with processing delays. Rate control functions, including fast-forward and fast-reverse functions, can be done by selectively throwing out non–key frames and processing key or I-frames only.

Other Applications

Several other applications using different components of SLIB were also written. Some of these are (1) Encode—a video encoding application that uses SLIB's compression component to compress raw video to M-JPEG format, (2) Rendit—a viewer for true color images that uses SLIB's rendering component to scale, tone-adjust, dither, quantize, color space convert, and display 24-bit RGB or 16-bit YUV images on frame buffers with limited planes, and (3) routines for viewing compressed on-line video documentation that was incorporated into Digital's videoconferencing product.

Related Work

While considerable effort has been devoted to optimizing video decoders, little has been done for video encoders. Encoding is generally computationally more complex and time-consuming than decoding. As a result, obtaining real-time performance from encoders has not been feasible. Another rationalization for interest in decoders has been that many applications require video playback and only a few are based on video encoding. As a result, "code once, play many times" has been the dominant philosophy. In most papers, researchers have focused on techniques for optimizing the various codecs; very little has been published on providing a uniform architecture and an intuitive API for the video codecs.

In this section, we present results from other papers published on software video codecs. Of the three international standards, MPEG-1 has attracted the most attention, and our presentation is biased slightly toward this standard. We concentrate on work that implements at least one of the three recognized international standards.

The JPEG software was made popular by the Independent Software JPEG Group formed by Tom Lane.[20] He and his colleagues implemented and made available free software that could perform baseline JPEG compression and decompression. Considerable attention was given to software modularity and portability. The main objective of this codec was still-image compression although its modified version has been used for decompression of motion JPEG sequences as well.

The MPEG software video decoder was made popular by the multimedia research group at the University of California, Berkeley. The availability of this free software sparked the interest of many who now had the opportunity to play with and experiment with compressed video. Patel et al. describe the implementation of this software MPEG decoder.[21] The focus in their paper is on an MPEG-1 video player that would be portable and fast. The authors describe various optimizations, including in-line procedures, custom coding frequent bit-twiddling operations, and rendering in the YUV space with color conversion through look-up tables. They observed that the key bottleneck toward real-time performance was not the computation involved but the memory bandwidth. They also concluded that data structure organization and bit-level manipulations were critical for good performance. The authors propose a novel metric for comparing the performance of the decoder on systems marketed by different systems vendors. Their metric, the percentage of required bit rate per second per

thousand dollars (PBSD), takes into account the price of the system on which the decoder is being evaluated.

Bheda and Srinivasan describe the implementation of an MPEG-1 decoder that is portable across platforms because the software is written entirely in a high-level language.[22] The paper describes the various optimizations done to improve the decoder's speed and provides performance numbers in terms of number of frames displayed per second. The authors compare the speed of their decoder on various platforms, including Digital's first Alpha-based PC running Microsoft's Windows NT system. They conclude that their decoder performed best on the Alpha system. It was able to decompress, dither, and display a 320-pixel by 240-line video sequence at a rate of 12.5 frames per second. A very brief description of the API supported by the decoder is also provided. The API is able to support operations such as random access, fast forward, and fast reverse. Optional skipping of B-frames is possible for rate control. The authors conclude that the size of the cache and the performance of the display subsystem are critical for real-time performance.

Bhaskaran and Konstantinides describe a real-time MPEG-1 software decoder that can play both audio and video data on a Hewlett-Packard PA-RISC processor-based workstation.[23] The paper provides step-by-step details on how optimization was carried out at both the algorithmic and the architectural levels. The basic processor was enhanced by including in the instruction set several multimedia instructions capable of performing parallel arithmetic operations that are critical in video codecs. The display subsystem is able to handle color conversion of YCbCr data and up-sampling of image data. The performance of the decoder is compared to software decoders running on different platforms from different manufacturers. The comparison is not truly fair because the authors compare their decoder, which has hardware assistance available to it (i.e., an enhanced graphic subsystem and new processor instructions), to other decoders that are truly software based. Furthermore, since all the codecs were not running on the same machine under similar operating conditions and since the sequence tested on their decoder is not the same as the one used by the others, the comparison is not truly accurate. The paper does not provide any information on the programming interface, the control flow, and the overall software architecture.

There are numerous other descriptions of the MPEG-1 software codecs. Eckart describes a software MPEG video player that is capable of decoding both audio and video in real time on a PC with a 90-megahertz Pentium processor.[24] Software for this decoder is available freely over the Internet. Gong and Rowe describe a parallel implementation of the MPEG-1 encoder that runs on a network of workstations.[25] The performance improvements of greater than 650 percent are reported when the encoding process is performed on 9 networked HP 9000/720 systems as compared to a single system.

Wu et al. describe the implementation and performance of a software-only H.261 video codec on the PowerPC 601 reduced instruction set computer (RISC) processor.[26] This paper is interesting in that it deals with optimizing both the encoder and the decoder to facilitate real-time, full-duplex network connections. The codec plugs under the QuickTime architecture developed by Apple Computer, Inc. and can be invoked by applications that have programmed to the QuickTime interface. The highest display rate is slightly under 18 frames per second for a QSIF video sequence coded at 64 kilobits per second with disk access. With real-time video capture included, the frame rate reduces to between 5 and 10 frames per second. The paper provides an interesting insight by giving a breakdown of the amount of time spent in each stage of coding and decoding on a complex instruction set computer (CISC) versus a RISC system. Although the paper does a good job of describing the optimizations, very little is mentioned about the software architecture, the programming interface, and the control flow.

We end this section by recommending some sources for obtaining additional information on the state of the art in software-only video in particular and in multimedia in general. First, the Society of Photo-Optical Instrumentation Engineers (SPIE) and the Association of Computing Machinery (ACM) sponsor annual multimedia conferences. The proceedings from these conferences provide a comprehensive record of the advances made on a year-to-year basis. In addition, both the Institute of Electrical and Electronics Engineers (IEEE) and ACM regularly publish issues devoted to multimedia. These special issues contain review papers with sufficient technical details.[14,27] Finally, an excellent book on the subject of video compression is the recently published *Digital Pictures* (second edition) by Arun Netravali and Barry Haskel from Plenum Press.

Conclusions

We have shown how popular video compression schemes are composed of an interconnection of distinct functional blocks put together to meet specified design objectives. The objectives are almost always set by the target applications. We have demonstrated that the video rendering subsystem is an important component of a complete playback solution and presented a novel algorithm for mapping out-of-range colors.

We described the design of our software architecture for video compression, decompression, and playback. This architecture has been successfully implemented over multiple platforms, including the Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems. Performance results corroborate our claim that current processors can adequately handle playback of compressed video in real time with little or no hardware assistance. Video compression, on the other hand, still requires some hardware assistance for real-time performance. We believe the widespread use of video on the desktop is possible if high-quality video can be delivered economically. By providing software-only video playback, we have taken a step in this direction.

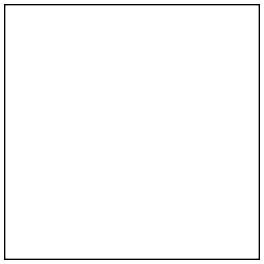
References

1. A. Akansu and R. Haddad, "Signal Decomposition Techniques: Transforms, Subbands, and Wavelets," *Optcon '92, Short Course 28* (November 1992).
2. E. Feig and S. Winograd, "Fast Algorithms for Discrete Cosine Transform," *IEEE Transactions on Signal Processing,* vol. 40, no. 9 (1992): 2174–2193.
3. S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory,* vol. 28 (1982): 129–137.
4. J. Max, "Quantizing for Minimum Distortion," *IRE Transactions on Information Theory,* vol. 6, no. 1 (1960): 7–12.
5. N. Nasrabadi and R. King, "Image Coding using Vector Quantization: A Review," *IEEE Transactions Communications,* vol. 36, no. 8 (1988): 957–971.
6. D. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE,* vol. 40 (1952): 1098–1101.
7. H. Harashima, K. Aizawa, and T. Saito, "Model-based Analysis-Synthesis Coding of Video Telephone Images—Conception and Basic Study of Intelligent Image Coding," *Transactions of IEICE,* vol. E72, no. 5 (1981): 452–458.
8. *Information Technology—Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and Guidelines,* ISO/IEC IS 10918-1:1994 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1994).
9. K. Correll and R. Ulichney, "The J300 Family of Video and Audio Adapters: Architecture and Hardware Design," *Digital Technical Journal,* vol. 7, no. 4 (1995, this issue): 20–33.
10. P. Bahl, "The J300 Family of Video and Audio Adapters: Software Architecture," *Digital Technical Journal,* vol. 7, no. 4 (1995, this issue): 34–51.
11. *Video Codec for Audiovisual Services at p × 64 kbits/s,* ITU-T Recommendation H.261, CDM XV-R 37-E (Geneva: International Telegraph and Telephone Consultative Committee, 1990).
12. *Coding of Moving Pictures and Associated Audio for Digital Storage Media at Up to about 1.5 Mbits/s,* ISO/IEC Standard 11172-2:1993 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1993).
13. *Generic Coding of Moving Pictures and Associated Audio, Recommendation H.262,* ISO/IEC CD 13818-2:1994 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1994).
14. Special Issue on Advances in Image and Video Compression, *Proceedings of the IEEE* (February 1995).
15. R. Ulichney, "Video Rendering," *Digital Technical Journal,* vol. 5, no. 2 (Spring 1993): 9–18.
16. L. Seiler and R. Ulichney, "Integrating Video Rendering into Graphics Accelerator Chips," *Digital Technical Journal,* vol. 7, no. 4 (1995, this issue): 76–88.
17. R. Ulichney, "Method and Apparatus for Mapping a Digital Color Image from a First Color Space to a Second Color Space," U.S. Patent 5,233,684 (1993).
18. Y. Arari, T. Agui, and M. Nakajima, "A Fast DCT-SQ Scheme for Images," *IEEE Transactions IEICE,* E-71 (1988): 1095–1097.
19. K. Froitzheim and K. Wolf, "Knowledge-based Approach to JPEG Acceleration," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE,* vol. 2419 (1995): 2–13.
20. T. Lane, "JPEG Software," Independent JPEG Group, unpublished paper available on the Internet.
21. K. Patel, B. Smith, and L. Rowe, "Performance of a Software MPEG Video Decoder," *Proceedings of ACM Multimedia '93,* Anaheim, Calif. (1993): 75–82.
22. H. Bheda and P. Srinivasan, "A High-performance Cross-platform MPEG Decoder," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE,* vol. 2187 (1994): 241–248.
23. K. Bhaskaran and K. Konstantinides, "Real-Time MPEG-1 Software Decoding on HP Workstations," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE,* vol. 2419 (1995): 466–473.
24. S. Eckart, "High Performance Software MPEG Video Playback for PCs," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE,* vol. 2419 (1995): 446–454.
25. K. Gong and L. Rowe, "Parallel MPEG-1 Video Encoding," *Proceedings of the 1994 Picture Coding Symposium,* Sacramento, Calif. (1994).

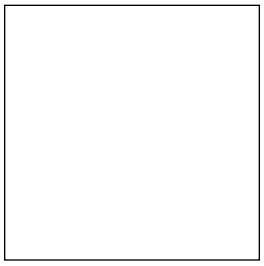
26. H. Wu, K. Wang, J. Normile, D. Ponceleon, K. Chu, and K. Sung, "Performance of Real-time Software-only H.261 Codec on the Power Macintosh," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE,* vol. 2419 (1995): 492–498.

27. Special Issue on Digital Multimedia Systems, *Communications of the ACM,* vol. 34, no. 1 (April 1991).

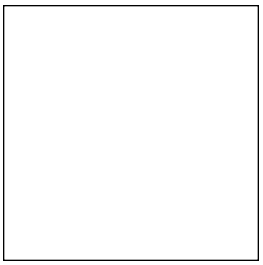
Biographies



Paramvir Bahl
Paramvir Bahl received B.S.E.E. and M.S.E.E. degrees in 1987 and 1988 from the State University of New York at Buffalo. Since joining Digital in 1988, he has contributed to several seminal multimedia products involving both hardware and software for digital video. Recently, he led the development of software-only video compression and video rendering algorithms. A principal engineer in the Systems Business Unit, Paramvir received Digital's Doctoral Engineering Fellowship Award and is completing his Ph.D. at the University of Massachusetts. There, his research has focused on techniques for robust video communications over mobile radio networks. He is the author and coauthor of several scientific publications and a pending patent. He is an active member of the IEEE and ACM, serving on program committees of technical conferences and as a referee for their journals. Paramvir is a member of Tau Beta Pi and a past president of Eta Kappa Nu.



Paul S. Gauthier
Paul Gauthier is the president and founder of Image Softworks, a software development company in Westford, Massachusetts, specializing in image and video processing. He received a Ph.D. in physics from Syracuse University in 1975 and has worked with a variety of companies on medical imaging, machine vision, prepress, and digital video. In 1991 Paul collaborated with the Associated Press on developing an electronic darkroom. During the Gulf War, newspapers used his software to process photographs taken of the nighttime attack on Baghdad by the United States. Working with Digital, he has contributed to adding video to the Alpha desktop.



Robert A. Ulichney
Robert Ulichney received a Ph.D. from the Massachusetts Institute of Technology in electrical engineering and computer science and a B.S. in physics and computer science from the University of Dayton, Ohio. He joined Digital in 1978. He is currently a senior consulting engineer with Digital's Cambridge Research Laboratory, where he leads the Video and Image Processing project. He has filed several patents for contributions to Digital products in the areas of hardware and software-only motion video, graphics controllers, and hard copy. Bob is the author of *Digital Halftoning* and serves as a referee for a number of technical societies, including IEEE, of which he is a senior member.