Image Enhancements for Low-Bitrate Videocoding

by

Brian C. Davison

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and
Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

© Brian C. Davison, MCMXCVI. All Rights Reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
1 May1996

Certified by ....
V. Michael Bove
Associate Professor of Media Technology
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 1 1 1996

Eng.

Image Enhancements for Low-Bitrate Videocoding

by

Brian C. Davison

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 1996, in partial fulfillment of the
requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

This research seeks new algorithms for videocoding that either improve the subjective image quality of coded sequences or decrease the computational complexity required to implement videocoding standards. The research is based on the H.26x international videocoding standards and targets its implementation around a Texas Instruments TMS320C8x Digital Signal Processor (DSP). Algorithm modifications for motion estimation, Discrete Cosine Transform (DCT) quantization, and rate control are devised and compared against test model defaults using objective measures. Results vary, but the motion estimation routines studied result in significant computational savings with only minor performance loss. Pre- and postprocessing routines are also developed and compared against controls via subjective ratings. The postprocessing algorithm, LO-COST, comprises a majority of the work, and attempts to filter reconstructed images based on a model of human perception. The model assumes that the images are composed of distinct objects which represent information to the viewer. The LO-COST algorithm applies both linear and non-linear filters to enhance object edges, mask noise on objects' interior regions, and eliminate artificial halos. The LO-COST algorithm was judged to effectively improve video quality of reconstructed sequences transmitted over low-bitrate channels with a minimal computational complexity.

Thesis Supervisor: V. Michael Bove
Title: Associate Professor of Media Technology, MIT Media Laboratory

Table of Contents

1 Introduction 9
2 Background 11
  2.1 H.26x Videocoding Standards 11
  2.2 The TMS320C8x Digital Signal Processor 15
3 Creating An H.261 Testbed In The C Environment 17
  3.1 Differences between an H.261 Reference Model and the C8x Codec 17
  3.2 Image Sectioning 17
  3.3 Buffer Regulation 18
  3.4 Quantization Control 19
  3.5 Coding-Mode Decisions 19
  3.6 Motion Estimation 21
  3.7 Performance Results of Code Modifications 22
4 Rate Control Studies 25
  4.1 Reasons for rate control 25
  4.2 Levels of Quantization Parameter Modification 25
  4.3 Testing various rate control algorithms 26
  4.4 Results and Conclusion 27
5 Motion Estimation Variations 31
  5.1 Reference Algorithms 31
  5.2 Metric Variations 32
  5.3 Use of Chrominance Information 37
  5.4 Search Windows for Motion Vectors 38
6 Equalizing the Signal-to-Noise Ratio Across Blocks 41
  6.1 Degraded SNR in border pixels 41
  6.2 Virtual Quantization Table 42
7 Prefiltering Sequences 47
  7.1 Encoding Smoothed Images 47
  7.2 Testing Results 49
8 PostProcessing Techniques 53
  8.1 Previous Research 53
  8.2 Noise and Human Perception 54
  8.3 Image Segmentation 55
  8.4 Filters 57
  8.5 Results 59
  8.6 Modifications 63
9 Conclusion 67
10 Acknowledgments 70
Bibliography 71
Appendix A H.261 Modified Code 73
  A.1 File "P64.c" 73
  A.2 File "ME.c" 111

List of Figures

Figure 3.1: Section One (GOBs #1-3) 18
Figure 7.1: Absolute Magnitude of 2-D Fourier Transforms of Prefilters 48
Figure 7.2: Subjective Ratings for Prefilters (Akiyo) 50
Figure 7.3: Subjective Ratings for Prefilters (Foreman) 51
Figure 7.4: Subjective Ratings for Prefilters (Mother/Daughter) 51
Figure 8.1: Non-edge pixels in monotonous regions 56
Figure 8.2: Non-edge pixels near object boundaries 57
Figure 8.3: Edge pixels 57
Figure 8.4: "Claire" before and after LO-COST is applied (H.261 @112 Kbps) 61
Figure 8.5: Enlarged shoulder ("Claire") before and after LO-COST is applied 61
Figure 8.6: Enlarged face ("Claire") before and after LO-COST is applied 62
Figure 8.7: "Akiyo" before and after LO-COST is applied (H.263 @24 Kbps) 62
Figure 8.8: Enlarged shoulder ("Akiyo") before and after LO-COST is applied 63
Figure 8.9: Enlarged face ("Akiyo") before and after LO-COST is applied 63

List of Tables

Table 3.1: Effects of C8x H.261 Algorithm Changes 23
Table 4.1: Performance of Various Rate Control Algorithms 28
Table 5.1: Performance of Motion Estimation Metrics (+/- 15 pel Search Window) 35
Table 5.2: Performance of Motion Estimation Metrics (+/- 7pel Search Window) 35
Table 5.3: Performance of H.263 Motion Estimation Metrics (QCIF Format) 36
Table 5.4: Performance of H.263 Motion Estimation Metrics (CIF Format) 36
Table 5.5: Use of Chrominance Information in Motion Estimation 38
Table 5.6: Varying 8x8 Block MV Search Range in H.263 Adv Pred Mode 38
Table 5.7: Varying PB Delta Component Search Range in H.263 PB Frames Mode 39
Table 5.8: Benefits of Advanced Prediction Mode 40
Table 6.1: Relative weight of DCT coeffs on Border Pixels 43
Table 6.2: Relative weight of DCT coeffs on Middle Pixels 43
Table 6.3: Relative weight of DCT coeffs on Inner Pixels 43
Table 6.4: Mask Applied to Measure Frequency of Coeff. Transmission 43
Table 6.5: Percentage of DCT coeffs transmitted in Default Codec 44
Table 6.6: Percentage of DCT coeffs transmitted after Threshold Mask 44
Table 6.7: Mask Applied as Threshold in Testing 44
Table 6.8: Results of Thresholding Non-masked DCT coeffs 44
Table 7.1: Results of Prefiltering Sequences Before Encoding 49
Table 8.1: Postprocessing Data for "Claire" coded with H.261 @ 112 Kbps 60
Table 8.2: Postprocessing Data for "Akiyo" coded with H.263 @ 24 Kbps 60
Table 8.3: Postprocessing Data for "Claire" coded with H.261 @ 112 Kbps 64

Chapter 1

Introduction

The videoconferencing market has exploded over the last few years, due in large part to advances in personal computers that allow the general public to afford this technology. Videocoding's parent field of image coding has generated heavy research for a number of years, with applications ranging from satellite imaging to medical diagnostics. While one field can be thought of as merely an extension of the other, videocoding and image coding have important differences as well. Image coding can only exploit spatial redundancy to reduce the amount of information needed to reconstruct the original image. Videocoding, however, can take advantage of both temporal and spatial redundancy in its compression techniques. While videocoding has the advantage of temporal redundancy, this benefit is accompanied by greater processing requirements and stricter bandwidth constraints. Instead of encoding one image under little time constraints, videocoding applications often require real-time encoding/decoding of approximately 10 frames/second to even be considered valuable as a consumer product. These applications transmit their encoded images over low-bandwidth channels, requiring videocoding algorithms to achieve very high compression ratios. Thus, tradeoffs must be made for videocoding algorithms to achieve high compression ratios while minimizing the processing requirements for implementation in consumer products. Encoded still images, however, are not usually under such strict time constraints for transmission. They can be sent over low-bandwidth channels with long transmission times, and the complexity of the encoding algorithm is not usually a critical issue. The similarities between these two areas give videocoding researchers a starting point, but there remains much work ahead before videocoding reaches the equivalent efficiency evident in current image coding techniques.

The research described in this report focuses on methods to improve videocoding techniques applied over low-bandwidth channels. The two major uses for this technology are videoconferencing using Integrated Synchronous Digital Network (ISDN) lines and the "videophone", which uses a telephone line as its transmission channel. The H.261 [1] and H.263 [2] standards were created for exactly these purposes[1] and will serve as the basis on which this research is evaluated. The target displays of these standards are typically computer screens or television monitors, as opposed to large conference screens where noise is much more apparent. The typical resolutions used in these applications are Common Intermediate Format (CIF), 352 pixels wide x 288 pixels high, and Quarter-CIF (QCIF), 176 pixels wide x 144 pixels high. The resolution is a crucial factor in videocoding, for it determines the number of pixels that must be re-created and often determines the minimum processing requirements to implement the video codec. Typical channel

1. These standards allow for transmission at variable bitrates up to 1.5 Mbps. However, most consumer applications will involve the lower bitrates since the cost of channel bandwidth is prohibitive for all but business applications.

bandwidths for these applications are 128 Kbps using two ISDN lines or 24 Kbps using a modem and standard telephone line. These are the basic constraints to which the encoding system must adhere.

This research was also targeted toward implementation on specific platforms, the core of each being a Texas Instruments TMS320C8X (C8x) digital signal processor. These processors are used in a wide array of applications from switching boards to modems, but were specifically designed to meet the needs of videocoding. Since the target applications of this research centers around consumer products, there is a tremendous need to limit the complexity of any algorithms, thereby reducing processing requirements (i.e. number of chips required) and ultimately cost. Due to its multiple on-chip parallel processors and their pipelined architecture, the C8x chips are more suited for certain implementations over others. Since the processing capabilities of the chip remains fixed, reductions in complexity in one area of the coding algorithm necessarily frees processing power for other areas, which might result in greater improvement for the final output display. The specific characteristics of the target platform must remain an important consideration when devising algorithms that improve the quality of the decoded video sequences.

The goal of this research was to devise alternative algorithms, at any given stage of these videoconferencing standards, that would improve the image quality of the final decoded video sequence. The videoconferencing standards mentioned above restrict which aspects of the encoding algorithm are open to modification, yet leave much flexibility to implement incremental improvements in many stages. Thus, schemes that call for significant modifications to the standards, such as overlapped block coding [3], are not considered. The rest of the report describes the modifications that were tested and is organized as follows. Chapter 2 describes the videocoding standards that serve as the basis for the research, as well as the processor that serves as the target platform for the codecs. Chapter 3 discusses the modifications that were made to the standards' test models to reflect the current implementation of these standards on the TMS320C8X platforms. Chapter 4 describes the results of rate control studies and chapter 5 presents the results of motion estimation alternatives. Chapter 6 investigates an attempt to equalize image quality throughout each image subblock. Chapter 7 describes the results of prefiltering video sequences prior to encoding, an approach designed for systems that have no means of postprocessing decoded sequences. Chapter 8, the heart of the research, describes the evolution of a low-complexity postprocessing algorithm that masks noise in decoded sequences transmitted over low-bandwidth channels.

Chapter 2

Background

2.1 H.26x Videocoding Standards

2.1.1 The H.261 Standard

The H.261 videoconferencing standard, approved by the ITU in 1991, is the first in a string of international agreements designed to standardize videoconferencing protocols. It is sometimes referred to as the Px64 standard because it was designed to be used in conjunction with ISDN lines, each containing 64 Kbps bandwidth. The standard allows for transmission of the encoded sequences at bitrates from 64 Kbps to 1.5 Mbps. The H.261 standard only describes the protocol for encoding the video images, albeit this represents the overwhelming majority of the transmitted information. Nearly all videoconferencing applications contain multiple layers that divide the responsibilities of encoding audio/video data from other tasks such as control sequences, which are used to negotiate bitrates, etc., and layers that mux these encoded streams into one bitstream for transmission. Thus, when a videoconferencing application secures 4 ISDN lines for communication, each end user must send its encoded information over 2 lines, or 128 Kbps. The audio, control, and other information require their own bandwidth, such that the video data is typically constrained to 112 Kbps[1]. Given this bandwidth, the goal of a video codec is to send the information that can most accurately re-create the video sequence it has captured on camera.

As stated in the introduction, the H.261 standard takes advantage of temporal and spatial redundancies present in video sequences to achieve very high compression ratios. The most typical resolution for this standard is CIF (352x288), which at a capture rate 30 frames/second produces $4.87 * 10^7$ bits of information every second. With a bandwidth constraint of 112 Kbps, this represents a compression ratio of 434 (or 144 if you only consider the target framerate of 10 images transmitted per second). To achieve these compression ratios, the codec assumes that successive images in a video sequence are very nearly identical, or rather contain the same elements, or objects, at different locations within the image. Similar to many image coding techniques, each image is divided into smaller blocks, known as macroblocks. The macroblocks of a transmitted image may be re-created by shifting regions of equivalent size in the last reconstructed (previously transmitted) image. The application receives information in the bitstream, via motion vectors, to determine which regions of the previous image should be used to reconstruct each macroblock. Since it is obvious that every macroblock in a new image cannot be perfectly constructed from the previous image's information (due to new objects entering/leaving the image or rotation of objects), residual error information is also transmitted. Instead of transmitting the errors pixel by pixel, the standard uses the Discrete Cosine Trans-

1. This bandwidth, 112 Kbps, is the default used in this research for tests involving the H.261 standard.

form (DCT) to gather most of the error energy into just a few DCT coefficients. In this way, the error information can be compressed and transmitted more efficiently. If the algorithm deems that two subsequent images, or respective regions within two subsequent images, are substantially different, the macroblocks are encoded as if they were part of a still image. In this case, the DCT is applied to the original pixel data (known as Intra-frame coding), and these transmitted DCT coefficients alone are used to reconstruct the macroblocks. The compression ratio significantly falls in these Intraframe-coded blocks, so preference is usually given to the Interframe-coded method that relies heavily on motion vectors and only uses the DCT for transmitting error information. The elimination of temporal redundancy, via motion vectors, and spatial redundancy, via the DCT, redundancy account for the amazing reduction in information that must be transmitted over the channel.

Determining accurate motion vectors for use in reconstructing encoded images is one of the main tasks of the H.261 algorithm. A motion vector must be computed for each macroblock (16 pixels wide x 16 pixels high) within the image, but the standard does not specify the means by which the motion vector must be chosen. Chapter 5 discusses some of the current motion estimation algorithms and the results of investigating some possible alternatives. The standard does limit the magnitude of the motion vector to +/-15 pixels in both the x and y directions and the resolution of the motion vector to that of integer-pel. These motion vectors are then differentially coded, such that identical motion vectors from successive macroblocks are encoded more efficiently than motion vectors that differ. "Successive macroblocks" refers to the ordering of macroblocks within the standard's encoding pattern. The H.261 standard divides CIF images into 12 Groups of Blocks (GOBs), each containing 33 macroblocks (3 rows x 11 columns). The ordering of the macroblocks depends both on the GOB to which they belong, and their position within the GOB. The standard horizontally traverses each row from left to right in a GOB, beginning with the top row. Although not every macroblock will physically border its successor in the encoded bitstream, differential encoding saves a substantial number of bits.

The H.261 standard includes an optional loop filter which smooths the motion predicted data prior to computing a residual error image. Once the motion vectors are derived, the encoding algorithm uses the MVs to shift regions of the previous image into each macroblock, creating a "predicted" image. Since this predicted image will almost certainly differ from the original source image, including many isolated errors of relatively large magnitudes, the loop filter is used to smooth the predicted image. The loop filter is merely a low-pass filter,

$$LoopFilter = \begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix} / 16, \tag{2.1}$$

which can be implemented very efficiently using shifts and additions. In addition to smoothing isolated errors within the predicted image, the lowpass loop filter makes the residual error image more amenable to

the DCT, which is most efficient when energy is concentrated in low frequencies. This feature was not included in the later H.263 standard, where it was deemed that the increased performance due to greater precision in motion vectors eliminated the benefits of the loop filter.

Once the motion vectors (and loop filter) are used to create a predicted image, this image is subtracted from the source image that is to be encoded. The difference is a residual error image, to which the DCT is applied. Each macroblock is broken into four (8 pixels x 8 pixels) blocks, and the DCT is applied separately to each of these 8x8 blocks. The DCT coefficients for each block are then quantized (divided by a quantization parameter) and included in each macroblock's encoded information. The quantizing of the DCT coefficients effectively applies a threshold test to each coefficient and inserts a quantization error to all those that remain. Since the DCT concentrates most of the energy within low-frequency coefficients, many high-frequency coefficient are not transmitted (quantized to zero). The decoder applies an inverse DCT to the coefficients, and adds this residual error image to its predicted image derived from the motion vectors (and optional loop filter). Once again, the DCT allows the standard to eliminate spatial redundancy among residual errors, substantially reducing the amount of information that must be transmitted. The use of the DCT arose from still-image coding algorithms, and there are questions regarding its effectiveness in videocoding. These issues and other noise characteristics will be explored in Chapter 6.

The previous descriptions ignore issues concerning how the derived information is bundled together and then encoded for transmission over the channel. While these issues are important in themselves, for the purposes of this research most of the details will be ignored. There is, of course, a given amount of overhead, including signalling when information for a new image/GOB/macroblock begins. The basic encoding unit is the macroblock. For each macroblock, information regarding the quantization parameter, the motion vector, the DCT coefficients, and coding modes is bundled together. Note that the quantization parameter can be updated on each macroblock in this standard, but frequent updates will increase the overhead required. As stated earlier, the motion vectors are differentially encoded. The DCT coefficients (in 8x8 blocks) are encoded in a zig-zag pattern using run length encoding. Thus each non-zero coefficient is encoded to the accuracy specified by the quantization parameter while zero coefficients are encoded via the number of consecutive zero coefficients. Once this information is computed, a huffman encoder is used to compress the DCT information even more. The information for several macroblocks is bundled together in a GOB unit, and these are bundled together in an image unit. The exact protocol and overhead contained at each level can be found in the standard itself.

The H.261 videoconferencing standard has proven to be a success based on the wide proliferation of applications that conform to its requirements. Research in low-bitrate videocoding did not cease with the issuance of this standard, though, and continuing improvements in H.261's basic algorithm have led the ITU to issue a new standard, the H.263 ("videophone") standard.

2.1.2 The H.263 Videophone Standard

The follow-up to the H.261 videoconferencing standard, H.263, significantly improves the image quality that can be achieved using the same bitrates. The H.263 standard includes several structural changes to the basic algorithm as well as several optional modes that can used to increase compression/quality even further. Although this standard was targeted for use at lower bitrates, it can also be used over the same bandwidths that H.261 allows. Judging from the results subsequent efforts to improve this algorithm even further (MPEG-4), the H.263 standard may serve as the basis for low-bitrate videocoding for some time.

The major structural change that H.263 incorporates into its standard regards the resolution and magnitude of motion vectors. The loop filter is not an option in the H.263 standard; however, the increased accuracy of motion vectors due to half-pel resolution more than accounts for the gain provided by the loop filter. Note, that while the standard dictates half-pel resolution in defining the units for motion vectors, the method and accuracy of the motion vectors is still left open for various implementations. Included in this change are the exact formulas by which half-pel interpolations between pixels must be calculated to assure compliance between encoder/decoders.

The standard also allows for 4 optional modes: Advanced Prediction Mode, Unrestricted Motion Vector Mode, PB Frames Mode, and Arithmetic Coding Mode. The two modes that contribute the greatest increases in performance are the PB Frames Mode and the Advanced Prediction mode. The PB Frames mode is similar to that of the MPEG-2 standard, allowing bi-directional interpolation of images. P-frames are those images which are reconstructed using only forward motion vectors (vectors relative to previous frames). B-frames use both forward motion vectors and backward motion vectors (vectors relative to future frames). A B-frame is always coupled with a P-frame and sent as one unit. The P-frame, although occurring later in the sequence than the B-frame, is decoded first. The B-frame is then decoded using forward vectors from previous frames and backward vectors based on its coupled P-frame. Both motion vectors for the B-frame are calculated by interpolating the forward motion vectors sent for the P-frame and adding an additional delta component for adjustment. Chapter 5 discusses motion estimation alternatives and includes tests that evaluate the performance of different search windows in calculating this PB delta component. The encoded B-frame also contains DCT coefficients and is reconstructed in much the same manner as a P-frame. The PB-frames mode introduces a greater delay between frames since two frames must be bundled together; however, the reduced overhead generated by transmitting both frames' information within one macroblock packet substantially reduces the number of bits required to encode equivalent information. These savings can be used either to increase the framerate or to increase the image quality while holding the effective framerate constant.

The Advanced Prediction Mode also provides significant improvements in image quality for the H.263 standard. The advanced prediction mode allows macroblocks to be broken down further into four 8x8 blocks. A different motion vector can be sent for each block. However, the greatest improvement arises out

of using the weighted average of three predictions for a given pixel. The three predictions are based on three motion vectors: the block's own motion vector, and the motion vectors associated with the pixel's closest neighboring blocks. For pixels near a block's edge, the neighboring block's motion vector might be more correlated with the pixel's actual motion. Thus, the weighted average benefits from more information when determining the prediction value for each pixel. This weighted average, in some ways, can be thought of as a variation of the loop filter in H.261, since both use a pixel's neighbors to more accurately predict the pixel's value. Generating four motion vectors per macroblock does not necessarily require more processing, since many motion estimation routines use a metric that is proportional to the number of pixels within the block. Also, the implementation certainly has the option to find one motion vector for the entire macroblock and repeat this motion vector for each 8x8 block. However, many test models first find a motion vector for the entire macroblock, and then conduct a search for each block's own vector that is centered around the macroblock motion vector. The benefits associated with various search windows for these 8x8 block motion vectors are discussed at the end of chapter 5. Finally, the AP mode also modifies the differential coding scheme used under H.261. In the AP mode, the basis for the differential coding is based on the median value of three different motion vectors. The method of deciding the sources of these motion vectors are described in detail in the actual H.263 standard. Although this method actually increases the amount of information sent per macroblock, the image quality for given framerates and bitrates is significantly improved.

These two videocoding standards and their associated test models serve as the basis for testing the results of this research. The actual standards contain a much greater description of the protocols in both written and graphical formats. Neither of these standards are concerned with any pre- or postprocessing that is performed on the video sequences. The major component of this research involves the use of pre/postprocessing algorithms to mask the noise introduced into the video sequences when transmitted over limited bandwidth channels. Such enhancements appear likely to remain outside the realm of future revisions of videocoding standards.

2.2 The TMS320C8x Digital Signal Processor

The TMS320C8x, formerly known as the MVP, was designed specifically for image processing applications. Such applications are compatible with highly parallel algorithms which subdivide the image and devote separate processors to each region within an image. For this reason, the C80 contains four Parallel Processors (PPs) together with a central RISC processor that is most commonly used for control algorithms. The chip also contains 50K of on-chip memory that is organized into separate 2K blocks and distributed over the PPs and RISC processor. A memory transfer controller handles movement of data from off-chip memory to on-chip memory, and requires very little overhead for initialization. The chip also includes a video controller that automatically refreshes the data to be displayed on screen. The following section describes important characteristics, for this research, of each component on the C80.

The chip contains both local and global data busses such that on any given cycle, one PP can access one data address through its local bus and another via the global bus. The words are 32-bits each; thus, 8-bit pixel values must be stored/retrieved in groups of four. These memory address units can also implement simple indexing functions, and thus can be used as a secondary unit for additions/subtractions. The PPs' arithmetic units are also highly parallelized and can perform several operations in a given cycle. A 64-bit instruction word allows each PP to receive instructions for multiple operations, both arithmetic and addressing, on each cycle. For example, logical and arithmetic operations can be combined so as to perform a subtraction and an absolute value operation in the same cycle. The ability of the PPs to perform such operations in parallel becomes important in the design of an efficient postprocessing routine discussed in Chapter 8. The ALU contained in each PP has a 32-bit wide data path, and can effectively conduct four 8-bit additions/ subtractions on a given cycle. Each PP also contains a 32-bit multiplier operating in parallel with the ALU and addressing units. The ability to perform 32-bit operations represents a tremendous increase in processing power but also imposes constraints on the alignment of the data. These issues will arise in both the motion estimation studies and the postprocessor design.

The programming of the C80's PPs is very complex and beyond the scope of this report and the research itself. However, benchmarks reveal that convolutions of filters of size NxN can be performed in N*N/2 cycles. Separable, two dimensional median filters are near the range of N*N operations. Current releases of the device run at 40 or 50 MHz, with future releases promising increased clock speeds. A cousin of the original MVP, the C82, is slated as the main workhorse for the H.263 codec. It contains only two PPs and the video controller has also been removed. However, the H.263 standard was intended to encode QCIF sequences over low bandwidths, which significantly reduces the amount of processing required of the chip. Yet with the need for higher compression ratios over these low bandwidths, the efficiency of codec routines is paramount in their design.

The master RISC processor contained on the C8X devices executes programs written in C. It runs at the same speed as the PPs and can access memory via the global data bus. This processor is used for command and control programs such as the mux and control layers used in videoconferencing applications. It initiates routines on each of the PPs by sending interrupt flags and storing instructions in the PP's appropriate interrupt handler. The PPs can also send the master processor signals via interrupt flags. Although not as powerful as a PP, this master processor can often handle some of the signal processing load since it is a pipelined processor itself. Although the PPs and the master processor handle the majority of the videocoding, additional devices are necessary for the chip's interfaces, such as the A/D and D/A converters and the ISDN controller. Together, the master processor and the four PPs make the C8X a powerful core around which a videoconferencing system can be based. However, care must be taken in harnessing the power of the C8X when designing around its efficient routines and memory constraints.

Chapter 3

Creating An H.261 Testbed In The C Environment

3.1 Differences between an H.261 Reference Model and the C8x Codec

Prior to this research, the engineers constructing the videoconferencing code at Texas Instruments had no means of objectively comparing different algorithms or enhancements. The engineers used a live feed/ display system that was capable of only subjective measurements. Therefore, the first task was to incorporate the specific features of the TMS320C8x H.261 codec into an H.261 test model written in C. The test model used was written by Andy Hung of the Stanford University Portable Research Group. Since speed and memory were not constrained in the simulation environment, no attempts were made to economize the code on the chosen platform - a Sun Sparc20 workstation. Altering the code to be consistent with the C8x implementation required modifications to the following areas: subdividing the image, buffer regulation, quantization control, coding-mode decisions, and motion estimation. In the subsequent sections, alterations in each of these areas are discussed in detail.

3.2 Image Sectioning

The C8x implementation of H.261 uses four parallel DSPs to code an image concurrently. For this reason, it divides the image into four sections and spawns four independent encoding loops, one on each processor. It was necessary to model this behavior in the C simulations since the order in which an image is encoded affects the iterative calculations of the quantization parameter. The code that incorporates this feature is found in the *p64EncodeSecX* procedure in the file "p64.c" (Appendix A). Each section is comprised of three consecutive Groups Of Blocks (GOBs). The first three GOBs constitute section one, the second three constitute section two, and so on. Figure 3.1 shows section one (GOBs 1-3) using this division scheme. The *p64EncodeSecX* procedure provides a loop for encoding the GOBs in groups of three. This allows the quantization parameter refinements to use sectional bit usage in the calculations. The effects of this modification are constrained to the *p64EncodeSecX* procedure only.

	GOB #4
GOB #5	GOB #6
....	

Figure 3.1: Section One (GOBs #1-3)

3.3 Buffer Regulation

Buffer regulation restricts the amount of encoded information that can be stored in memory prior to actual transmission over the channel. Since the C8x devices will be severely restricted in available memory, the target buffer size was limited to two frames' apportionment of bits. A frame's apportionment is not a hard cap for any given frame, but the average number of bits that can be allocated to each frame and still achieve the desired output framerate. For a target framerate of 10 frames/second over a 112 Kbps channel, two frames' apportionment amounts to less than 23 Kbits:

$$\text{Target_Buffersize} = 2 \text{ frames} \bullet 112 \text{ Kbits/sec} / (10 \text{ frames/sec}) = 22.4 \text{ Kbits.} \tag{3.1}$$

The target buffersize is used when deciding whether a subsequent frame should be encoded or skipped. The actual buffer size is greater than the target size, but its actual details are more related to overflow issues than the rate control decisions studied in this research. The target buffer size restriction is particularly relevant to intraframe-coded images, in which large numbers of bits are required to reconstruct the image using DCT coefficients alone. Upon coding an intraframe image, several frames must be skipped, thus purging the buffer, before subsequent frames can be analyzed and encoded. The buffer restriction also tends to limit the number of successive frames containing extensive motion that are encoded. Although these interframe-coded images require less bits than intraframe-coded images, large residual errors from extreme motion quickly fill a two-frame buffer, forcing many of the high-motion subsequent frames to be skipped. The code that implements this alteration can be found in procedures *BufferContents* and *p64EncodeFrame* of file "p64.c" in Appendix A. Upon completion of encoding a frame, the number of bits transmitted during the corresponding time interval is subtracted from the buffer. If the buffer still contains more bits than one frame's apportionment, the next frame in line to be coded is skipped and bits are subtracted from the buffer for that time interval. This process continues until the buffer's contents drops below one frame's apportionment. The H.261 reference model included provisions for handling buffer overflows as well. If, at any point during the encoding of a frame, the buffer becomes full, the rest of the frame is skipped. These measures ensure that the C code correctly models the C8x H.261 buffer control algorithm.

3.4 Quantization Control

Quantization control determines how often and to what extent the Quantization Parameter (QP) is adjusted according to bit usage. Quantization control, analogous to fine tuning, and framedropping, analogous to a rough adjustment, act to ensure the encoding meets bandwidth constraints. Two different schemes were written to model the C8x codec implementation and can be found in procedures *p64EncodeSec2* and *p64EncodeSec3* of the file "p64.c" in Appendix A. The first scheme compares the number of bits used by the corresponding section of the previous image to given thresholds and adjusts the QP by +/-1 accordingly. Note that, in this scheme, each section has its own QP that is adjusted independently of the other sections' QPs. The thresholds are set at 20% above and below one section's proportion of the bit allocation for the entire frame. If the desired output framerate were 10 frames/second and the bitrate were 112 Kbps, the corresponding thresholds for the bits used by one section would be 3.36Kbits (120% * 112 Kbps / 10 f/s / 4 sections/frame) and 2.24 Kbits (80% * 112 Kbps / 10 f/s / 4 sections/frame). The formula for modifying the QP using this scheme is given by

$$\text{If } B_{-1}[\text{section}] > \frac{bitrate}{F \bullet 4} \bullet 1.2, \text{ then QP[section] += 1.} \tag{3.2}$$

$$\text{If } B_{-1}[\text{section}] < \frac{bitrate}{F \bullet 4} \bullet 0.8, \text{ then QP[section] -= 1.} \tag{3.3}$$

$B_{-1}[i]$ refers to the bits used in encoding the ith section of the previous frame.

The second scheme compares the number of bits used by the previous section within the same image to its allocated proportion, and likewise adjusts the QP by +/-1. Its formula is given by

$$\text{If } B[\text{section - 1}] > \frac{bitrate}{F \bullet 4} \bullet 1.2, \text{ then QP += 1.} \tag{3.4}$$

$$\text{If } B[\text{section - 1}] < \frac{bitrate}{F \bullet 4} \bullet 0.8, \text{ then QP -= 1.} \tag{3.5}$$

B[i] refers to the bits used in encoding the ith section of the current frame. If i = -1, the bits used to encode the last section of the previous frame are used instead. The choice of which method to use is determined by the *p64EncodeSecX* function call in the procedure *p64EncodeFrame* of the same file. These two schemes were also used to test various rate control schemes. The results of the rate control tests are discussed in chapter 4.

3.5 Coding-Mode Decisions

The coding-mode of a macroblock refers to the amount and kind of information that is encoded for each macroblock. The coding-mode decisions of the H.261 reference model were changed in two ways. First, whenever a macroblock was encoded with motion vector information, the loopback filter was enabled.

The loopback filter applies a lowpass filter to the motion-compensated data before overlaying it on the original image to determine the residual error. The DCT is then applied to this residual error image. The filter mitigates sharp errors that can result largely from imperfect motion information. If used, the loopback filter must be implemented on both the encoding and decoding ends to assure identical reconstructed images on both ends. This change entailed modifying the flag array *FilterMType* in the "p64.c" file.

The second change involved implementing a different scheme, modeled after TMN5 of H.263, of deciding how much information to transmit. For each macroblock, the sum of absolute differences and the absolute sum of absolute differences are calculated. The sum of absolute differences (SAD) is the equal to the sum of the absolute values of the individual pixel differences between motion compensated data from the previously reconstructed image and the current source image to be encoded. Its formula is given by

$$\sum_i \sum_i abs\,(c[i,j] - p[i,j])\,. = SAD, \tag{3.6}$$

where c[i,j] represents the respective macroblock of the current source image and p[i,j] represents the respective macroblock of the previously reconstructed image. The SAD for each macroblock is selected after determining which motion vector (MV) provides the lowest SAD value. In performing the search for the best MV, the zero-displacement SAD, or SAD_0, (associated with a (0,0) motion vector) is preferenced against all other motion vectors by subtracting 100 from its value. If this SAD_0 is selected as the minimal SAD, the preferenced SAD_0 value is used as the SAD value for all further decisions concerning the macroblock. The absolute sum of absolute differences (ASAD) is the sum of the absolute value of the differences between each pixel in the respective macroblock of the current source image and the average of all of the pixels within the current source image. Its formula is given by

$$\sum_i \sum_j abs\,(c[i,j] - \mu) = ASAD, \tag{3.7}$$

where μ is the average of all the pixels in the current image and c[i,j] represents the respective macroblock's values within the current original image. These two values, SAD and ASAD, are then used to determine the mode of encoding the macroblock. The previous calculations are performed in the procedures *MotionEstimation*, *ThreeStep*, and *OneAtATime* of the "me.c" file in Appendix A.

The first step in determining the coding-mode for each macroblock compares the chosen SAD with (ASAD + 500). If the SAD is greater, intraframe mode is selected and a DCT is performed on the pixels of the original image. These DCT coefficients are then used to reconstruct the macroblock without the aid of the previously transmitted image. In this case, interframe coding is assumed to be more costly than intraframe coding, given the high SAD value relative to the variance over the macroblock. Otherwise, the macroblock is interframe-coded, exploiting the temporal redundancy between images using motion compensation. The SAD is then compared with a threshold value, NOT_CODED_THRESH, to determine how

much information should be coded for the macroblock. If the SAD is less than the threshold, only the motion vector is coded for the macroblock - no DCT residual coefficients are calculated. If the SAD is greater than the threshold, both the motion vector and quantized DCT coefficients, which represent the residual errors between a filtered, motion predicted image and the current source image, are encoded. Thus, three coding modes are allowed: intraframe coding, interframe coding with MVs only, and interframe coding with both MVs and DCT coefficients transmitted. These coding options are identical with those used in the C8x H.261 codec. These coding mode changes are incorporated in the *p64EncodeGOBX* procedures of the file "p64.c" in Appendix A.

3.6 Motion Estimation

The motion estimation algorithm of the C8x codec was designed to economize the number of computations that needed to be performed with only minimal image degradation. Two optional routines were available: the "One-At-a-Time" and the "Three-Step" algorithms. The One-At-a-Time algorithm begins with a (0,0) motion vector and compares its SAD with those of adjacent motion vectors. It repeats the process until it finds a MV with a minimum SAD (i.e. a local minimum). First, the zero-displacement SAD is compared with the SADs for motion vectors of (0,1), (0,-1), (-1, 0) and (1,0). If the lowest SAD is associated with one of the four nonzero MVs, its corresponding motion vector is chosen to serve as a new starting point from which to continue the search. If, however, the zero-displacement SAD is lowest, the routine would then check the motion vectors (-1,-1), (-1,1), (1,-1), and (1,1). If the zero-displacement SAD is still the lowest, the routine ends and the zero-displacement MV and SAD are used for that macroblock. Otherwise, the routine chooses the nonzero MV associated with the lowest SAD as its new starting point and repeats the search pattern until a local minimum is found. The number of computations performed is variable and could be in excess of 48 SAD computations (6 steps * 8 SADs/step). However, experimental data place the average number of one pixel steps taken between 3 and 5 (~30 SADs). The full search path (+/- 7 pixels in both x and y directions) used by the H.261 test model required the computation of 225 SADs.

The Three-Step algorithm is actually a series of search paths that employ a logarithmically decreasing step size. Since the desired search range is +/-7 pixels, this requires three steps of +/-4, +/-2, and +/-1 pixels. The first search path, using a step size of 4, compares the zero-displacement SAD with those associated with the motion vectors (-4,-4), (0,-4), (4,-4), (-4,-0), (4,0), (-4, 4), (0,4), and (4,4). The MV corresponding to the lowest SAD is chosen as the starting point for the next search path which uses a step size of +/- 2 instead of +/-4. The search is again repeated with a step size of 1. This Three-Step search also is not guaranteed to find the minimal SAD, but is generally more immune to selecting merely a local minimum, as demonstrated by the objective PSNR data that follows. The Three-Step motion search computes a constant 25 SADs for each macroblock. This represents a savings factor of 9 over the default full search algorithm. These two new motion estimation procedures, found in the file "me.c" in Appendix A, allow the C codec to exactly model the options available in the C8x codec.

3.7 Performance Results of Code Modifications

The results of the these modifications were tested by computing Peak Signal to Noise Ratios (PSNRs) of the decoded output sequences with respect to the source sequences. The PSNRs for each frame is then averaged over the entire sequence to determine an objective measure of the performance of the given algorithm. In computing these measurements, care must be taken to ensure the measurements represent a reasonable measure of picture quality. For example, consider the following three sequences of frames:

original:	0	1	2	3	4	5	6	7
coder 1:	0'	1'	2'	3'	4'	5'	6'	7'
coder 2:	0"	0"	0"	3"	3"	3"	6"	6"

One could compute the signal to noise ratio (another representation of mean squared error),

$$PSNR = \left(\sum_x \sum_y \frac{(r[x,y] - s[x,y])^2}{256} \right) / (xlines \cdot ylines) \quad . \tag{3.8}$$

by subtracting 0' from 0, 1' from 1, 2' from 2, etc., and by subtracting 0" from 0, 0" from 1, and so on. However, while this would accurately describe the performance of coder 1, this would not correlate to visual errors observed by the viewer for coder 2. Thus, in all the measurements given in this report, the SNR measurements are computed by subtracting 0" from 0, then 3" from 3, and so on. Thus, output frames are always compared to their corresponding source frame. When frames are skipped (e.g. an output frame is repeated more than three times for a target framerate of 10 f/s), the measurement for the skipped frame is not computed. Since algorithms that skip frames more frequently allocate more bits to the frames that are actually encoded, we cannot limit our comparisons of algorithms to SNR measurements alone. Thus, the actual output framerate is also given in each test. Small differences in output framerates are usually insignificant (e.g. 8.6 vs. 8.8 f/s), but it is expected that algorithms that achieve a higher output framerate could generate a higher SNR at a lower output framerate. Both measures are important when making comparisons. In chapters 7 and 8, these (objective) measurements will prove less useful than subjective measurements, which convey the viewer's comprehension of information rather than an objective measure of how well the image is exactly reproduced.

Tests were performed after each of the previous algorithm changes were implemented to determine their effects on coding efficiency and quality. The tests were conducted on 150 frames of a CIF-sized sequence encoded at 112 Kbps with a target output framerate of 10 frames/second. The results are given with respect to cumulative changes, i.e. the effects of each change were not tested independently. Table 3.1 summarizes the PSNRs that resulted from these tests. The output framerate is 10 frames/second unless otherwise noted. As the test data show, significant computational savings resulting from the Three-Step motion estimation routine can be achieved with a minimal performance loss. The addition of coding-mode thresholds generally increased performance and, as can be expected, framedropping and quantization parameter

calculation changes significantly improved PSNR while lowering the framerate. These changes represent the baseline, the TMS320C80 H.261 implementation, from which future modifications in this research are judged.

PSNR (dB) *[Framerate]*	Claire	Foreman	Miss America	Students
Test Model	39.10	29.13	38.75	31.97
w/ 3-Step ME	39.12	29.07	38.68	31.95
w/ Loop Filters	39.30	28.99	38.75	30.58
w/ Sectioning	39.29	29.00	38.77	30.62
w/ Coding Thresholds	39.58	28.49	38.83	30.47
w/ FrameDropping and Quant Control	40.23 *[8.5 f/s]*	31.89 *[4.0 f/s]*	39.09 *[8.5 f/s]*	31.55 *[7.9 f/s]*

Table 3.1: Effects of C8x H.261 Algorithm Changes

Chapter 4

Rate Control Studies

4.1 Reasons for rate control

When transmitting encoded video data over very low bandwidths, there are many tradeoffs one must make in terms of image quality, framerate, buffering, etc. An increased appropriation of bits to a single frame will increase the video quality, but has an inverse effect on the framerate. Buffering frames allows one to maintain a consistent framerate, but its memory requirements are often prohibitive on low-cost consumer applications. Each of these tradeoffs are usually regulated by the rate control algorithm, and as can be seen from the recent MPEG 4 testing, can have a significant impact on subjective ratings[1].

The rate control algorithms considered have two mechanisms at their disposal to regulate these tradeoffs: framedropping and the Quantization Parameter (QP). Framedropping provides the encoding algorithm with a coarse adjustment to meet both memory buffering constraints and bandwidth constraints. By controlling the output framerate, framedropping also indirectly determines the maximum image quality that can be attained for a given bandwidth. The quantization parameter is the actual mechanism by which the image quality is set. The QP actually determines the maximum error that can be introduced into the encoded DCT coefficients, thereby controlling the accuracy of the reconstructed image. The QP parameter also determines how many non-zero DCT coefficients, the major component of the encoded bitstream, are encoded. By setting the allowable error contained in each quantized coefficient, the QP sets a threshold for coefficient values, below which, no coefficients are encoded. The rate control algorithm adjusts the QP so as to maximize the image quality (minimize the QP) while adhering to limited bit allocations per frame. Together, these two mechanisms can be used to reconcile tradeoffs between fluid motion between frames and accuracy within frames. The goal of the rate control algorithm is to intelligently adjust the QP and the framerate via framedropping to satisfy consumer preferences while conforming to bandwidth and memory constraints. The algorithms presented here are not extremely complex, but vary significantly in their effects upon the characteristics of the resulting reconstructed sequences.

4.2 Levels of Quantization Parameter Modification

Current videoconferencing standards allow the QP to be adjusted at the macroblock level. Thus, the QP can also be used to achieve a higher resolution at certain areas within the image in addition to regulating the frame's overall bit consumption. There are various theories regarding which areas of the image should

1. In the November 1994 MPEG-4 testing, encoding algorithms that implemented very low framerates (~2-3 frames/second) were often judged superior to encoding schemes using higher framerates (~8-10 f/s), even though it is not possible to compare coding efficiency over such varied framerates.

receive more bits and how often the quantization parameter should be adjusted to accommodate such apportionments. The H.261 reference model sets the QP proportional to the buffer fullness at the macroblock level according to the formula

$$QP = ((BufferContents) / (QDFact)) + QOffset, \tag{4.1}$$

where QDFact = Bitrate/80 and QOffset nominally equals one. To implement this scheme on the C8x, the QP must remain constant over an entire image, since it is impractical for each independent coding thread to access the bit consumption of the other processes. Such a routine, designed to accommodate buffering constraints and duplicating much of the purpose of framedropping, provides very little flexibility in adjusting the QP within an image.

Several other approaches to adjusting the QP track bit consumption at higher levels (by section or frame) or attempt to apportion an equivalent number of bits to each section of an image. Attempts to give equivalent bandwidth to each region within an image are inconsistent with the goal of maximizing the information content of each frame. In such a scheme, many bits will be wasted on encoding insignificant, monotonous regions at very fine resolutions, while detailed sections are therefore necessarily coded at relatively much coarser resolutions. Schemes that regulate the QP at lower levels (at the MacroBlock or Group of Blocks level) are able to accommodate scene changes and severe motion without dropping several subsequent frames by rapid incrementation of the QP. However, this volatility in the QP often causes regions of similar detail within an image to be encoded at different resolutions. Such spatial variations in resolution can be distracting to the viewer. Schemes that regulate the QP at higher levels within the image provide a more consistent resolution within the image, but can often cause several frames to be dropped when highly detailed regions within a new image quickly fill the buffer. In an effort to resolve these tradeoffs, several routines that incorporate these QP adjustment models were constructed and tested to evaluate their relative performance.

4.3 Testing various rate control algorithms

The rate control algorithms tested differed according to three criterion: the level at which QP adjustment occurred, whether frames were dropped, and whether bit usage comparisons were intraframe or interframe. The C8x's H.261 implementation divides an image into four sections to allow each of its parallel PP's to encode a section independently. This sectional division of the frame served as one level at which bit usage could be analyzed in addition to the frame level and the Group of Blocks level. Based on the level chosen, the QP adjustment formula would be invoked at the beginning a new GOB, section, or frame. The algorithms also varied according to how they compared bit consumption with allocations. One method compared the number of bits used for the previous region within the same frame to its bit allocation when deciding whether to adjust the QP for the subsequent region. The formulas for this scheme are given by

$$\text{If } B_{-1}\,[\text{section}] > \frac{bitrate}{F \bullet 4} \bullet 1.2, \text{ then QP[section] += 1.} \tag{4.2}$$

$$\text{If } B_{-1}\,[\text{section}] < \frac{bitrate}{F \bullet 4} \bullet 0.8, \text{ then QP[section] -= 1.} \tag{4.3}$$

Another method tracked a QP for each region within the frame independently, adjusting each QP based on the number of bits used to encode the respective region in the previous frame. This scheme's formulas are given by

$$\text{If } B\,[\text{section - 1}] > \frac{bitrate}{F \bullet 4} \bullet 1.2, \text{ then QP += 1.} \tag{4.4}$$

$$\text{If } B\,[\text{section - 1}] < \frac{bitrate}{F \bullet 4} \bullet 0.8, \text{ then QP -= 1.} \tag{4.5}$$

This second scheme apportioned nearly equivalent bandwidths to each region within the frame. For each scheme, the bits consumed in encoding a region were compared to the region's proportional share of the bits allocated for the entire frame at a desired framerate. If the number of bits consumed in a region was considerably greater than its share, the QP would be increased by one (represents coarser resolution); and likewise if the used bits were much less, the QP was decreased by one. "Considerably greater or less than" is defined as greater/less than 120/80% of the region's proportion of bits.

Each of these schemes was tested both with and without the use of framedropping. Disabling framedropping allows schemes to be compared using equivalent framerates. However, this does not represent real conditions and hurts various schemes disproportionately. Thus, both output framerate and PSNR must be considered when comparing the results of these tests. The default C8x implementation also further restricts the QP to an upper limit of 14 instead of 31 as stated in the accepted standards. It was determined that above a QP level of 14, the picture quality is degraded to such an extent that consumers would not value the information contained in the reconstructed sequences. The C8x implementation also attempts to encode only every nth frame; thus, when a single frame is dropped, the distance between subsequent encoded images is 2*n frames. For these tests, every third frame was considered for encoding.

4.4 Results and Conclusion

The tests were conducted on 150 frames of CIF-sized sequences encoded using the H.261 standard at 112 Kbps. The output framerate and an average of the PSNR of the reconstructed sequences are given for each of the schemes described in Table 4.1. The schemes are differentiated according to the level at which the bit consumption/allocation decision is made (frame, section of GOB), whether the decision is based on regions within the same frame or between successive frames, and whether framedropping was enabled. The

first scheme allows the QP to fluctuate between 2 and 31, whereas all other schemes limit the QP to a maximum value of 14.

PSNR (dB) ***Framerate (f/s)***	Claire	Students
1. Section, Within Frame, No Dropping, 2 < QP < 31	39.88 8.8 f/s	30.17 5.0 f/s
2. Section, Within Frame, No Dropping	39.72 9.0 f/s	29.24 7.6 f/s
3. Section, Within Frame, Frame Dropping	39.30 10 f/s	28.63 10 f/s
4. Frame, Within Frame, No Dropping	39.09 10 f/s	28.48 10 f/s
5. Section, Between Frame, Frame Dropping	38.66 9.6 f/s	28.49 9.4 f/s
6. Section, Between Frames, No Dropping	38.56 10 f/s	28.42 10 f/s
7/ GOB, Within Frame, Frame Dropping	39.34 9.6 f/s	28.32 8.6 f/s
8. GOB, Within Frame, No Dropping	36.65 10 f/s	28.52 10 f/s
9. GOB, Between Frame, Frame Dropping	38.49 9.6 f/s	28.62 9.4 f/s
10. GOB, Between Frame, No Dropping	38.34 10 f/s	28.52 10 f/s

Table 4.1: Performance of Various Rate Control Algorithms

The scheme with the highest performance (#2) also resulted in the lowest framerate. However, when we only consider tests in which framedropping was disabled, the same QP adjustment scheme (#3) results in the highest PSNR values. (Note that even when adjusting the QP at the frame level, the results are fairly comparable to the most effective scheme.) Scheme #2 adjusts the QP based on the number of bits used in the previous section within the same image, balancing the trade-off between consistent resolution within an image and accommodation of high motion or detail. The results appear reasonable, since the preferred scheme naturally allocates more bits to regions of greater detail by maintaining a constant QP value over large regions of an image rather than equalizing the bits used by regions of high and low detail. By varying

the QP at the section level, this scheme also allows the QP to change fairly rapidly to account for extreme motion, but not to the extent that different regions of the image are encoded with significantly different QP values, as occurs when QP changes are made at the GOB level. The use of nonuniform quantization levels and randomized scanning of macroblocks [4] offer alternative solutions to these problems. However, such schemes would require a change in the current videocoding standards, and were not considered for the purpose of this research. Other tests might be conducted in which the maximum level of the QP is varied or different framedropping decisions are implemented. However, this trade-off between output framerate and image quality relates more to psychovisual studies than to pure algorithm improvements. Note that subjective tests were not recorded in assessing these schemes, in part because in order for the viewer to differentiate in performance, the viewer would need to be attuned to the differences produced by the algorithms that were being compared. This would naturally inject some bias into the testing. Objective testing served sufficiently to determine the characteristics of each scheme.

Chapter 5

Motion Estimation Variations

5.1 Reference Algorithms

5.1.1 Standard Motion Estimation Algorithm

The function of a motion estimation (ME) algorithm is to provide motion vectors that may be used to accurately predict the motion of image blocks when reconstructing an encoded sequence. A standard ME model is needed before one may assess the performance of various ME algorithms. A widely accepted standard evaluates the Sum of Absolute Differences (SAD) metric for each motion vector within a given window size. The SAD is simply a sum over a 16x16 macroblock (or 8x8 block if using the advanced prediction mode of H.263) of the absolute difference between corresponding pixels of successive frames. The value of the SAD metric is given by the formula

$$S = \sum_{j=-15}^{15} \sum_{i=-15}^{15} \left| c[m,n] - p'[m+j, n+i] \right|, \tag{5.1}$$

where c is the current frame to be coded and p' is the latest previously-reconstructed frame. This model is commonly known as a "full search SAD" algorithm, where "full search" is defined as the testing of every possible motion vector within an NxN search window, normally centered about the motion vector (0,0).

This standard is arbitrary, for given the available processing power, one might substitute a Least Squares (LS) metric (equivalent to using variance) to minimize the energy left in the residual error image. However, the LS metric favors vectors that minimize large errors between pixels, errors that are often left uncorrected over low bit-rates. This occurs because many high frequency DCT coefficients must be coded to correct large single-pixel errors, yet these high frequency coefficients are either coarsely quantized or not coded due to limited bandwidth. Furthermore, large single-pixel errors can often be mitigated with postprocessing routines, as will be shown in chapter 8, and minimizing such errors should not be the only focus of motion estimation routines. The LS metric also increases complexity, requiring either a multiplication, lookup or some other method to calculate the square for each difference that is computed. In our target platform, the TMS320C8x, the current implementation computes the differences between four pairs of pixels during one instruction. Computing and squaring each difference separately would eliminate the efficiency gained from the 32-bit data path. Therefore, although the LS metric is widely used in other fields, the SAD metric is preferable for this application.

5.1.2 TMS320C8x H.261 Motion Estimation Algorithm

The ME algorithm included in TI's C8x implementation of the H.261 codec conducts a series of successive searches to determine an accurate motion vector for each macroblock. Instead of completing a full-

search for a given search window, each successive search evaluates nine motion vectors that are offset by a constant "step size". In each successive search, this step size is halved, thereby increasing the resolution of the ME to the desired level. In this routine, the zero displacement SAD (motion vector of (0,0)) is first computed. The SAD for motion vectors of +/-N pixels, where N is a power of 2, are also computed (i.e. [-N,-N], [0,-N], [+N,-N], ...). The lowest of the nine SADs computed is selected and its motion vector are used as the origin for the remaining searches and the baseline SAD. The step size is reduced by a factor of two, to N/2, and the process is repeated until the desired resolution is reached (either integer or half-pel). A typical +/- 7 pixel search window requires three passes, thus explaining the initial name of "3-Step" motion search.

The C8x H.261 motion estimation routine also contains two threshold tests to determine if a non-zero motion vector is warranted, or should even be sought. If the zero displacement SAD, SAD_0, is greater than NO_MOTEST_THRESH, the motion search is conducted. Otherwise, the (0,0) motion vector is presumed sufficient and is recorded along with SAD_0 for that macroblock. If the motion search is conducted, the SAD corresponding to the chosen motion vector (minimum SAD found) is compared to the quantity SAD_0 minus 100. The lower of these two values and its corresponding motion vector are recorded for the given macroblock. These thresholds give preference to the zero displacement SAD for two reasons. Coding efficiency can be greatly improved if several successive motion vectors are coded (0,0) since vectors are coded differentially and a zero motion vector is the most common motion vector. Second, in the presence of little actual motion and low spatial frequency, this motion estimation routine sometimes chooses inaccurate motion vectors due to camera noise, etc. Objective tests were used to set the NO_MOTEST_THRESH at approximately 200 (for a 16x16 macroblock). The offset bias of 100 in the second comparison was chosen from the H.263 Test Model Number 5 (TMN5) algorithm.

This C8x default algorithm, logarithmic step search path combined with SAD metric, serves as the basis for comparing alternative algorithms. In the following sections, several different metrics will be described and tested in conjunction with the default algorithm. In the metrics described so far, only the luminance information for each pixel has been used in the computations. Two different methods of incorporating the pixels' chrominance information will also be investigated. Finally, the results of varying search windows for H.263's advanced prediction and PB frames modes will be discussed.

5.2 Metric Variations

With the goal of reducing the computational complexity required for motion estimation, five alternative metrics were devised and tested. Each variation uses the default C8x motion estimation algorithm described above but replaces the SAD computation with an alternative metric. Each metric, in essence, converts 2x2 subblocks into one functional value, effectively reducing the size of the macroblock by a factor of four. The sum of absolute differences is computed for these functional values, and serves as the new metric for selecting a motion vector. The formula for this metric is given by

$$S_x = \sum_{i=0}^{7} \sum_{j=0}^{7} |f_{i,j} - f'_{i,j}|, \qquad (5.2)$$

where $f_{i,j}$ represents functional values for the current image and $f'_{i,j}$ represents functional values for the previously constructed image. These metrics can only be applied to passes in which the step size is greater than or equal to two, since the functional values are stored at half the resolution of the original image. Therefore, when the integer-pel pass (step size of one) is conducted, the normal SAD metric is used. Since the functional values reduce the effective size of the macroblock and thus the computational complexity of computing SADs by four, but need only be computed and stored once, the cost of conducting a motion search can be significantly reduced. The methods by which each metric's functional values are computed are described below.

5.2.1 Downsampling

In the downsampled metric, each 2x2 pixel subblock is replaced by the top-left pixel's value. The formula for its functional value is given by

$$f_{i,j} = b[m + 2i, n + 2j] \times 4 \qquad (5.3)$$

where b is the target image, [m, n] give the initial coordinates of the given macroblock and [i, j] are the indices of the functional value. This metric requires no processing other than reordering the pixels values in memory. Essentially, this metric reduces the image to one-half its resolution by aliasing high and low frequencies. On CIF-sized images, it performs comparably to the default algorithm.

5.2.2 Low Frequency

The low frequency metric uses the average of the 2x2 subblock for its functional value. Since our main intent is to match the DC value within local areas, this approach should work quite well. However, pronounced edges are not matched as readily using only low frequencies. The formula for this metric's functional value is given by

$$f_{i,j} = \sum_{k=0}^{1} \sum_{l=0}^{1} b[m + 2i + k, n + 2j + l] \qquad (5.4)$$

5.2.3 Edge Detection

The above descriptions refer to the frequency components contained within macroblocks. Since these images are composed of objects, an equally valid analysis notes the location of distinct objects' edges. Matching the location and angle of edges is a primary function, if not the primary function, of motion estimation. This metric tests for edges in diagonal directions, as can be seen by its functional value formula,

$$f_{i,j} = \sum_{k=0}^{1} \sum_{l=0}^{1} b[m+2i+k, n+2j+l] \times (-1)^{k+l}. \tag{5.5}$$

If a purely horizontal or vertical edge exists (i.e. $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$ or $\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$), the functional value computed for the 2x2 subblock is nearly equal to zero. Thus, this metric primarily matches the orientation of edges.

5.2.4 Normalized Edge Detection

The edge detection metric does not account for the DC value within the local area, only for the presence of edges with a given orientation. To include consideration for the absolute luminance, the normalized edge detection adds the edge detection and downsampled functional values together. Thus, the average of this functional value is normalized close to the average of the local area within the macroblock. The formula for this metric's functional value is given by

$$f_{i,j} = \left(\sum_{k=0}^{1} \sum_{l=0}^{1} b[m+2i+k, n+2j+l] \times (-1)^{k+l} \right) + 4 \times b[m+2i, n+2j]. \tag{5.6}$$

5.2.5 Gradient

The gradient metric determines the existence and intensity of an edge using only two of the four subblock values. It can identify both horizontal and vertical edges and edges that lie in one, but not both, of the diagonal directions. The computational cost of its functional value is slight, as given by

$$f_{i,j} = b[m+2i, n+2j] - b[m+2i+1, n+2j+1]. \tag{5.7}$$

5.2.6 Results

These metrics were substituted for the SAD metric in the C8x default motion estimation algorithm as noted above. The C8x H.261 codec was used to code 150 frames of a CIF-sized sequence at 112 Kbps. The maximum quantization parameter value was set to 12 and the minimum to 2. The tables below give the average PSNR for all coded frames and the corresponding framerates for each metric. Table 5.1 limited the search window to +/- 15pixels while Table 5.2 used a window of +/- 7. Both calculate motion vectors to integer-pel resolution.

PSNR (dB) *Framerate (f/s)*	'Akiyo'	'Claire'	'Foreman'	'Moth/D'	'Students'
SAD	37.28 *8.2*	40.42 *9.0*	32.71 *3.2*	36.42 *8.2*	33.19 *7.8*
Downsampled	37.26 *8.2*	40.40 *9.0*	32.70 *3.2*	36.43 *8.0*	33.10 *7.4*
Low Freq	37.16 *8.2*	40.29 *9.2*	32.71 *3.0*	36.50 *7.8*	33.08 *7.8*
Edge Detection	36.64 *8.4*	39.92 *9.0*	32.75 *2.4*	35.64 *7.8*	32.95 *5.8*
Norm Edge	37.00 *8.4*	40.27 *9.0*	32.66 *3.0*	36.03 *8.2*	32.99 *7.2*
Gradient	37.19 *8.0*	40.50 *8.6*	32.72 *3.0*	36.36 *8.0*	33.08 *7.8*

Table 5.1: Performance of Motion Estimation Metrics (+/- 15 pel Search Window)

PSNR (dB) *Framerate (f/s)*	'Akiyo'	'Claire'	'Foreman'	'Moth/D'	'Students'
SAD	37.31 *8.2*	40.44 *9.2*	32.72 *3.0*	36.55 *8.0*	33.20 *7.8*
Downsampled	37.19 *8.4*	40.43 *9.0*	32.69 *3.0*	36.54 *7.8*	33.20 *7.8*
Low Freq	37.24 *8.4*	40.43 *9.0*	32.73 *3.0*	36.47 *8.0*	33.17 *7.6*
Edge Detection	36.91 *8.6*	40.29 *8.8*	32.74 *2.6*	35.96 *8.0*	32.98 *6.8*
Norm Edge	37.09 *8.2*	40.37 *9.0*	32.68 *3.0*	36.37 *8.0*	33.06 *7.4*
Gradient	37.18 *8.2*	40.41 *9.0*	32.70 *3.0*	36.49 *8.0*	33.16 *7.6*

Table 5.2: Performance of Motion Estimation Metrics (+/- 7pel Search Window)

The data demonstrate that minimal performance loss results from either the downsampling metric or the gradient metric under these conditions. Since CIF images are susceptible to much larger motion vectors than QCIF (half the resolution), these metrics are one way of extending a motion estimation routine's search range while limiting computational costs. In computing one macroblock's motion vector using a +/-15 search window, the default routine must compute the SAD metric 24 times during its first three steps. Using one of the alternative metrics, this processing would be reduced by a factor of four - equivalent to 6 SAD metric computations. Thus, for the full +/- 15 search, the default method requires 33 SAD metric computations compared to an equivalent of 15 (plus the initial functional value computations) for the alternative metrics. For a +/- 7 pixel search range, the relative costs are 25 to 17. This represents a savings of 30-54%.

Tests were also conducted using the H.263 Telenor Research codec. Its default motion estimation algorithm used a full search along with the SAD metric. The two most promising alternative metrics were inserted in place of the SAD and used for the entire motion vector search. Both CIF (@ 60 Kbps) and QCIF (@ 24 Kbps) sequences were coded and the results are found in Tables 5.3 and 5.4. The alternative metrics did not perform as well in this experiment as above. This is probably due to the fact that the logarithmic search requires a SAD computation for the integer-pel pass whereas the H.263 test codec's full motion search used the alternative metrics over its entire search. Differences in the H.26x algorithms might also explain the greater margins. However, these alternative metrics still compare favorably to the default algorithms due to their considerable savings in processing requirements.

PSNR (dB)	Akiyo	Foreman	M/D	Miss America
SAD	34.47	27.66	33.46	39.01
Decimation	34.37	27.20	33.35	38.84
Low Frequency	33.46	27.07	32.53	38.52

Table 5.3: Performance of H.263 Motion Estimation Metrics (QCIF Format)

PSNR (dB)	Akiyo	Foreman	M/D
SAD	36.66	30.24	35.77
Decimation	36.49	30.02	35.65
Low Frequency	35.61	29.95	35.15

Table 5.4: Performance of H.263 Motion Estimation Metrics (CIF Format)

5.3 Use of Chrominance Information

Modern videoconferencing protocols separate luminance and chrominance information before processing and encoding the data. Even though far fewer bits are spent on transmitting the chrominance information, it retains a much higher signal to noise ratio - approximately 2-5 dB in low-bitrate codecs. Most of the energy is contained in the luminance field, and consequently luminance data is normally used exclusively for motion estimation, postprocessing, etc. One of the failure modes of motion estimation routines occurs when objects of equal intensity but of different color interact. Since most motion estimation routines use only luminance information, it becomes difficult to accurately predict motion vectors near such objects. In this section, the attempt to incorporate chrominance information into the algorithm's motion vector decisions will be described and tested.

Two variations of the C8x H.261 default motion estimation routine were created to incorporate chrominance information in selecting motion vectors. In the first approach, chrominance information was used exclusively during the passes of step size two or greater. The luminance information is always used for the integer pel pass since the chrominance data is stored at half the resolution of luminance data in low bitrate codecs. Interpolating chrominance data to use for integer pel resolution was deemed inefficient and its impact negligible. The second approach used a combination of the luminance and chrominance information for passes of step size two or greater. Again, the final pass of integer pel resolution was conducted using only luminance information.

The testing was conducted on CIF-sized sequences of 150 frames using the C8x H.261 codec at 112 Kbps. The following table gives the PSNR and framerates for the default motion estimation routine along with the two variants. As the data indicate, the chrominance information used alone did not improve the accuracy of the motion vectors, but actually lowered performance. The combined use of chrominance and luminance information is similar to the default in both luminance and chrominance PSNR, but its computational costs do not justify implementation. This conclusion does not rule out using chrominance information in any motion estimation routine. Simply, it notes that two straightforward implementations that used chrominance information over low bit-rates were unsuccessful at significantly improving video quality. Given the low ratio of noise in the chrominance fields compared to the luminance field, other implementations would most likely have only marginal effects.

Lum PSNR (dB) **Cb PSNR (dB)** **Cr PSNR (dB)** ***Framerate (f/s)***	Akiyo	Claire	Foreman	Moth/Daugh
Default	37.31 40.69 42.60 *8.2*	40.42 41.35 43.99 *9.2*	32.72 38.75 40.57 *3.0*	36.58 41.06 42.57 *8.2*
Chrom Only	36.42 40.33 42.30 *7.8*	39.51 40.84 43.86 *9.0*	32.70 39.31 41.22 *2.2*	35.42 40.71 42.48 *8.2*
Chrom + Lum	37.23 40.67 42.55 *8.2*	40.48 41.47 44.05 *9.0*	32.70 38.74 40.55 *3.0*	36.57 41.09 42.60 *8.2*

Table 5.5: Use of Chrominance Information in Motion Estimation

5.4 Search Windows for Motion Vectors

The H.263 standard's optional modes provide significant improvement over the basic implementation. The advanced prediction mode's 8x8 block MVs and overlapped motion compensation are the source of much of this gain. Some have suggested that these individual block (8x8) MVs be restrained very closely to the MV chosen for the entire 16x16 macroblock. The same predictions have been made regarding the PB frames mode's delta vector component. Restricting the search windows for these two components could provide substantial computational savings compared to a full search implementation for each individual motion vector. To verify these results, tests were conducted to compare PSNR performance for various window sizes for both the PB frames mode's delta component and the 8x8 block MVs. Telenor Research's H.263 code was used to encode 100 QCIF-sized frames at 24 Kbps with the advanced prediction mode, PB frames mode, and the SAC mode all turned on.

P-frame PSNR (dB) **B-frame PSNR (dB)** ***Bitrate (Kbps)***	Akiyo	Foreman	M/D	Miss America
Range = +/- 1	34.76 34.44 *23.84*	27.55 26.07 *24.11*	33.98 33.41 *23.85*	39.47 38.73 *23.64*

Table 5.6: Varying 8x8 Block MV Search Range in H.263 Adv Pred Mode

P-frame PSNR (dB) **B-frame PSNR (dB)** ***Bitrate (Kbps)***	Akiyo	Foreman	M/D	Miss America
Range = +/- 2	34.77 34.46 *23.90*	27.59 26.29 *24.21*	33.97 33.39 *23.90*	39.49 38.78 *23.72*
Range = +/- 4	34.86 34.51 *23.67*	27.43 26.26 *24.10*	34.03 33.41 *23.79*	39.41 38.69 *23.69*
Range = +/- 8	34.79 34.46 *23.74*	27.43 26.33 *24.24*	34.00 33.42 *23.66*	39.47 38.72 *23.69*

Table 5.6: Varying 8x8 Block MV Search Range in H.263 Adv Pred Mode

P-frame PSNR (dB) **B-frame PSNR (dB)** ***Bitrate (Kbps)*** ***Framerate (f/s)***	Akiyo	Foreman	M/D	Miss America
Range = +/- 1	34.73 34.41 *23.89* *12.63*	27.55 25.73 *24.28* *6.67*	33.99 33.37 *23.68* *10.74*	39.43 38.67 *23.66* *13.26*
Range = +/- 2	34.77 34.46 *23.90* *12.63*	27.59 26.29 *24.21* *6.77*	33.97 33.39 *23.90* *10.74*	39.49 38.78 *23.72* *13.26*
Range = +/- 4	34.89 34.55 *23.88* *12.63*	27.58 26.44 *24.05* *6.67*	34.03 33.46 *23.79* *10.74*	39.48 38.74 *23.67* *13.12*
Range = +/- 8	34.89 34.55 *23.89* *12.63*	27.50 26.55 *23.95* *6.88*	34.01 33.48 *23.77* *10.85*	39.48 38.74 *23.76* *13.12*

Table 5.7: Varying PB Delta Component Search Range in H.263 PB Frames Mode

The data are not entirely consistent, but indicate that a search window size no greater than 2 for both components produces consistently high performance in all the sequences. Above this value, improved performance is not necessarily evident. Due to little variation in performance when the 8x8 block MV search window is altered, one may question the benefit of the advanced prediction mode. To demonstrate that other features of this mode provide significant increases in performance, sequences were encoded with either a zero search range for the 8x8 block MVs or with the advanced prediction mode turned off. Note that before

the 8x8 block MV searches are implemented, a 16x16 block MV search is always conducted first. This 16x16 block MV is required in the event that the encoding process decides to send an identical value for each 8x8 block MV. The 16x16 block MV also provides a good initial guess for the 8x8 block MVs. Table 5.8 demonstrates the benefits of the advanced prediction mode, primarily the overlapped block motion compensation. It is quite clear that the Advanced Prediction mode is beneficial, regardless of the size of the search window for the 8x8 block MVs.

P-frame PSNR (dB) **B-frame PSNR (dB)** ***Framerate (f/s)***	Akiyo	Foreman	M/D	Miss America
Search Range of 0	34.76 34.44 *12.63*	27.55 26.07 *6.67*	33.98 33.41 *10.74*	39.47 38.73 *13.26*
Adv Pred Mode Off	34.55 34.26 *12.37*	27.17 25.60 *6.67*	33.77 33.25 *10.74*	39.09 38.49 *12.90*

Table 5.8: Benefits of Advanced Prediction Mode

Chapter 6

Equalizing the Signal-to-Noise Ratio Across Blocks

6.1 Degraded SNR in border pixels

A phenomenon of block-based video codecs is the relatively poor signal to noise ratio in pixels near block boundaries compared with pixels close to the blocks' center. This difference can often be as great as 1.5 dB over low-bitrate channels. There are several causes for the increased noise present in such border pixels, including uncorrelated motion vectors and quantization of DCT coefficients. Motion vectors attempt to find the greatest correlation between the current image's target block and a block of similar size on the previous frame. However, since these blocks are not always composed of a single physical object, different regions of the block might be best represented by different motion vectors. The calculated motion vectors will therefore have greater correlation with the block's interior pixels, which are more likely to belong to a single physical object present in the macroblock than pixels near the block's borders. The block's border pixels, defined as the pixels immediately adjacent to a block's boundary, have greater dispersion and their movement is less likely to be correlated with each other or with interior pixels. Since only one motion vector can be sent per macroblock in codecs such as H.261, the MVs chosen often do not achieve the highest correlation with border pixels.

Another cause for lower SNR in border pixels arises from the quantization of DCT coefficients. Sharp discontinuities often exist across blocks as motion vectors are selected to maximize correlation with the movement of the entire block. Compensation for these errors would require a corrective signal to include many high frequency components in order to represent a step. However, the DCT is widely used in these codecs because of its characteristic of compressing most of the signal's energy into low frequency coefficients. After coarse quantization, most high frequency coefficients are discarded due to limited bandwidth. In coding entire images, most of the information relevant to the human observer is contained in low frequencies, however, residual error images have much different statistical distributions. Thus, the DCT may not be optimally suited to correcting these discontinuities across block boundaries.

Solutions have been proposed to mitigate these effects in recent codec standards. H.263's overlapped motion compensation recognizes that the motion of border pixels may be more closely related to the motion vector of its neighboring block. Therefore, these border pixels are reconstructed by averaging estimates made by using three different blocks' motion vectors. This greatly reduces the disparity in SNR between border and inner pixels but does not solve the biases in the DCT. Since this research focused on enhancing algorithms based on current standards, all possible solutions would need to be compliant with such standards. It would not be feasible to use a different method of coding residuals or modifying the quantization parameter protocol contained in these approved standards. Instead, indirect modifications are made to the

quantization of DCT coefficients by thresholding their values before quantization occurs. By allowing only the DCT coefficients that have a greater impact upon border pixels to be encoded, the gap in SNR between border and inner pixels might be lowered. Even if the SNR for the overall image decreased as a result, by reducing the noise in the "weak link", the overall subjective quality might be improved.

6.2 Virtual Quantization Table

Analysis of the DCT equation (6.1-2 below) shows that the spatial weighting causes different frequency coefficients to have varied effects on pixels across a block. For example, using an 8x8 block transform, the coefficient of index (1,1) has a spatial weighting of |2.75| for pixel [0,0] and a spatial weighting of |0.65| for pixel [6,3]. When a coefficient's spatial weighting is squared and then averaged over a region of pixels, the square root of this average gives a measure of the spatial weighting of this coefficient on the given region of pixels. Of course, if this average was computed for the entire 8x8 block, the result would be equal to one. But since we have noticed a disparity between border pixels and interior pixels, this information might be used to preference coefficients that have a greater impact on border pixels. Tables 6.1-6.3 shows the relative weighting that each DCT coefficient has for three different regions within an 8x8 block: inner, middle, and border pixels. Inner pixels are defined as the 16 pixels in a 4x4 array at the center of the block. The border pixels include the 28 pixels that surround the block. The middle pixels are the 20 pixels between these first two groups. In Table 6.4, a mask was produced that gave preference to low frequency DCT coefficients that most greatly affect border pixel values. Tables 6.5 and 6.6 show the frequency with which each DCT coefficient was transmitted both before and after applying the mask. By creating a higher threshold for the non-masked DCT coefficients and thus limiting their transmission, more bits might be saved, lowering the quantization parameter, and consequently quantizing the masked DCT coefficients at a finer resolution.

$$F(u,v) = (1/4)\,C(u)\,C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) \times \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \quad (6.1)$$

$$f(x,y) = (1/4)\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)\,C(v)\,F(u,v) \times \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \quad (6.2)$$

(C(u), C(v) = 1/sqrt(2) for u, v = 0 and C(u), C(v) = 1 otherwise)

1.00	1.40	1.30	1.16	1.00	0.84	0.70	0.60
1.40	1.67	1.61	1.51	1.40	1.28	1.19	1.12
1.30	1.61	1.53	1.43	1.30	1.18	1.07	1.00
1.16	1.51	1.43	1.30	1.16	1.02	0.90	0.82
1.00	1.40	1.30	1.16	1.00	0.84	0.70	0.60
0.84	1.28	1.18	1.02	0.84	0.65	0.49	0.39
0.70	1.19	1.07	0.90	0.70	0.49	0.32	0.21
0.60	1.12	1.00	0.82	0.60	0.39	0.21	0.09

Table 6.1: Relative weight of DCT coeffs on Border Pixels

1.00	0.97	0.58	0.55	1.00	1.45	1.42	1.03
0.97	0.77	0.67	0.73	0.97	1.20	1.26	1.17
0.58	0.67	0.25	0.18	0.58	0.97	0.90	0.48
0.55	0.73	0.18	0.08	0.55	1.02	0.92	0.37
1.00	0.97	0.58	0.55	1.00	1.45	1.42	1.03
1.45	1.20	0.97	1.02	1.45	1.86	1.92	1.69
1.42	1.26	0.90	0.92	1.42	1.92	1.95	1.59
1.03	1.17	0.48	0.37	1.03	1.69	1.59	0.89

Table 6.2: Relative weight of DCT coeffs on Middle Pixels

1.00	0.35	1.00	1.27	1.00	0.73	1.00	1.65
0.35	0.12	0.35	0.44	0.34	0.25	0.35	0.57
1.00	0.35	1.00	1.27	1.00	0.73	1.00	1.65
1.27	0.44	1.27	1.61	1.27	0.93	1.27	2.10
1.00	0.34	1.00	1.27	1.00	0.73	1.00	1.65
0.73	0.25	0.73	0.93	0.73	0.53	0.73	1.21
1.00	0.35	1.00	1.27	1.00	0.73	1.00	1.65
1.65	0.57	1.65	2.10	1.65	1.21	1.65	2.73

Table 6.3: Relative weight of DCT coeffs on Inner Pixels

X	X	X					
X	X	X	X	X	X	X	X
X	X			X			
	X						
	X	X					
	X						
	X						
	X						

Table 6.4: Mask Applied to Measure Frequency of Coeff. Transmission

62	33	22	15	11	7.9	5.4	3.2
28	21	15	11	8.6	6.3	4.5	2.5
17	14	10	7.8	5.9	4.3	2.8	1.4
11	9.0	7.4	5.3	3.4	2.5	1.5	0.8
8.0	6.6	5.2	3.0	2.3	1.5	0.7	0.3
5.5	4.7	3.7	1.9	1.2	0.4	0.1	-
3.6	3.0	2.1	1.0	0.5	0.1	-	-
1.5	1.4	1.0	0.4	0.1	-	-	-

Table 6.5: Percentage of DCT coeffs transmitted in Default Codec

64	34	24	5.8	4.3	2.8	1.7	0.9
30	23	18	14	9.6	7.2	5.0	2.7
18	15	2.7	1.9	7.1	0.9	0.4	0.2
3.9	10	1.9	1.2	0.7	0.4	0.1	-
2.5	7.6	6.7	0.6	0.3	0.1	-	-
1.8	5.7	0.8	0.3	0.1	-	-	-
0.9	3.5	0.4	0.1	-	-	-	-
0.3	1.6	0.2	-	-	-	-	-

Table 6.6: Percentage of DCT coeffs transmitted after Threshold Mask

To test this scheme, the DCT coefficient quantization algorithm was modified to zero all non-masked DCT coefficients whose value was less than twice the quantization parameter. The mask used for these tests is shown in Table 6.7. The frequency at which each DCT coefficient was transmitted, as well as the PSNR, was recorded for each region within the 8x8 blocks using both the default and modified quantization schemes. Sequences containing 150 frames were encoded using H.261 algorithm at 112 Kbps. The results of these tests are shown in Table 6.8.

X	X	X					
X	X	X	X	X	X	X	X
X	X	X	X	X			
	X	X	X				
	X	X					
	X						
	X						
	X						

Table 6.7: Mask Applied as Threshold in Testing

PSNR (dB)	*Overall*	*Border*	*Middle*	*Inner*
Claire (orig)	40.41	39.72	40.91	41.19
Claire (T2)	39.97	39.59	40.21	40.39
Students (orig)	32.69	32.06	33.18	33.37
Students (T2)	32.21	31.96	32.43	32.41

Table 6.8: Results of Thresholding Non-masked DCT coeffs

The data indicate that the disparity between the SNR for the inner and border pixels was narrowed; however, the PSNR for each region was also degraded. If the overall PSNR decreased while the border pixel region's PSNR increased, the modification might have increased subjective quality. However, with a decrease in PSNR for each region, this scheme certainly could not be expected to improve image quality. Although this particular scheme did not improve performance, modification of the quantization table for DCT coefficients remains a possible source of enhanced image quality. Indeed, JPEG codecs contain quantization tables that vary for each DCT coefficient, since humans notice certain frequencies more readily. Such mechanisms might prove worthwhile in video codecs, although the statistical properties of residual error images are considerably different than those of source images.

Chapter 7

Prefiltering Sequences

7.1 Encoding Smoothed Images

As described in the previous chapter, an image's high frequency components are often quantized to zero when transmitting encoded data over very low bandwidths. Sharp edges and noise are just two common causes of such high frequency components occurring in video sequences. In order to ensure the accurate encoding of the low frequency components, one might seek to mitigate such high frequency components before applying the encoding algorithm. The pre-filtered sequence may be blurred slightly as a result of smoothed edges, etc., but the SNR of the decoded sequence relative to the encoder's input sequence is greatly improved. It is difficult to compare reconstructed sequences originating from different sources. Should the SNR be computed with respect to the original source sequence or to the prefiltered source sequence? Since each "objective" comparison will result in a different conclusion, a purely subjective comparison of the reconstructed sequences is more suited for this analysis.

The filters examined in this research are designed for efficient implementation on a digital signal processor, namely the TMS320C8x. Nearly all of the filters can be implemented using only shifts and additions, eliminating the need for any multiplications. The coefficients of each of the 3x3 filters used are listed below. Their respective frequency plots are given in Figure 7.1. Each filter has a DC component equal to unity gain and varied degrees of attenuation at the higher frequencies.

$$PF_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} / 6 \quad PF_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} / 8 \tag{7.1}$$

$$PF_3 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 8 & 1 \\ 0 & 1 & 0 \end{bmatrix} / 12 \quad PF_4 = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} / 16 \tag{7.2}$$

$$PF_5 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 12 \quad PF_6 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 16 \tag{7.3}$$

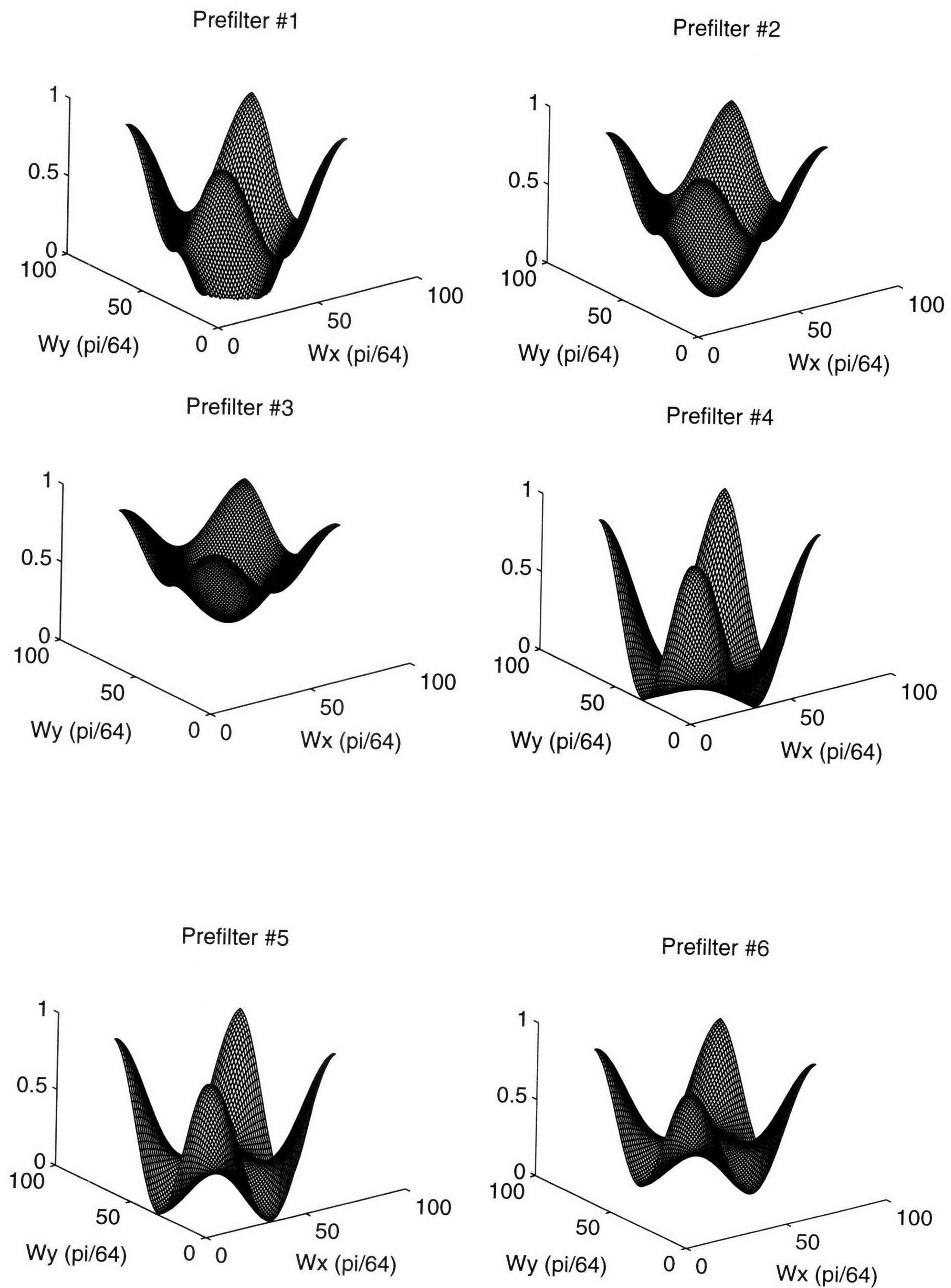

Figure 7.1: Absolute Magnitude of 2-D Fourier Transforms of Prefilters

7.2 Testing Results

In testing these filters, four sequences were prefiltered with each of the six lowpass filters. Signal to noise ratios were computed for the prefiltered sequences with respect to the original sequence. The prefiltered sequences were then encoded using the H.26x codecs, and signal to noise ratios were computed for the reconstructed sequences with respect to both the original source and prefiltered source sequences. These PSNRs are given in Table 7.1. As stated above, such comparisons are not the best measure of performance in these experiments. Therefore, subjective tests were conducted in which four versions (using different prefilters) of each reconstructed sequence were displayed adjacently on a video screen and synchronously played at normal speed. Both expert viewers and novices gave their preferences in ranked order. The results of the subjective testing is given in Figures7.2-7.4. The reconstructed sequences with the lowest scores were preferred to those with higher scores. A second test was also conducted in which reconstructed sequences based on prefilters #2,#3 and #6 (the most promising filters) were compared to the reconstructed sequence based on the source sequence. Both of these scores are included in the figures.

Pre-Filtered PSNR (dB) **P-Frame/B-Frame PSNR (dB)** Framerate (f/s) PSNR (dB) relative to Originals	Akiyo	Foreman	Mother & Daughter
No Prefilter	N/A 34.77/34.46 12.63 39.94	N/A 26.60/26.24 10.11 32.52	N/A 33.85/33.58 10.00 39.78
Filter 1	39.97 38.99/38.71 14.04 37.36	36.55 29.34/28.92 12.00 31.29	41.91 37.52/37.04 10.00 38.21
Filter 2	42.43 38.08/37.78 13.75 38.23	39.03 28.67/28.22 11.63 31.56	44.39 36.65/36.26 10.00 38.76
Filter 3	45.82 36.94/36.64 13.40 39.05	42.49 27.95/27.50 11.02 31.87	47.89 35.62/35.53 10.00 39.20
Filter 4	37.73 40.42/40.15 14.53 36.16	34.95 30.12/29.61 12.63 30.86	39.53 38.62/38.22 10.00 37.22

Table 7.1: Results of Prefiltering Sequences Before Encoding

Pre-Filtered PSNR (dB) **P-Frame/B-Frame PSNR (dB)** Framerate (f/s) PSNR (dB) relative to Originals	Akiyo	Foreman	Mother & Daughter
Filter 5	38.14 40.06/39.81 14.38 36.25	35.61 29.99/29.53 12.63 30.98	39.88 38.38/38.00 10.00 37.26
Filter 6	40.61 38.58/38.35 14.04 37.39	38.09 29.02/28.60 12.13 31.32	42.37 37.17/36.76 10.00 38.02

Table 7.1: Results of Prefiltering Sequences Before Encoding

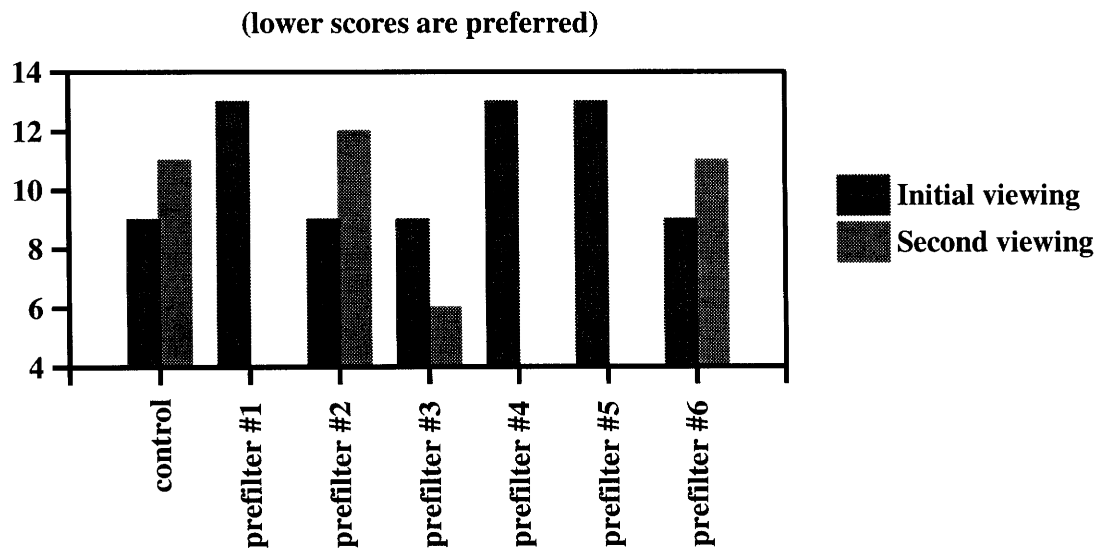

Figure 7.2: Subjective Ratings for Prefilters (Akiyo)

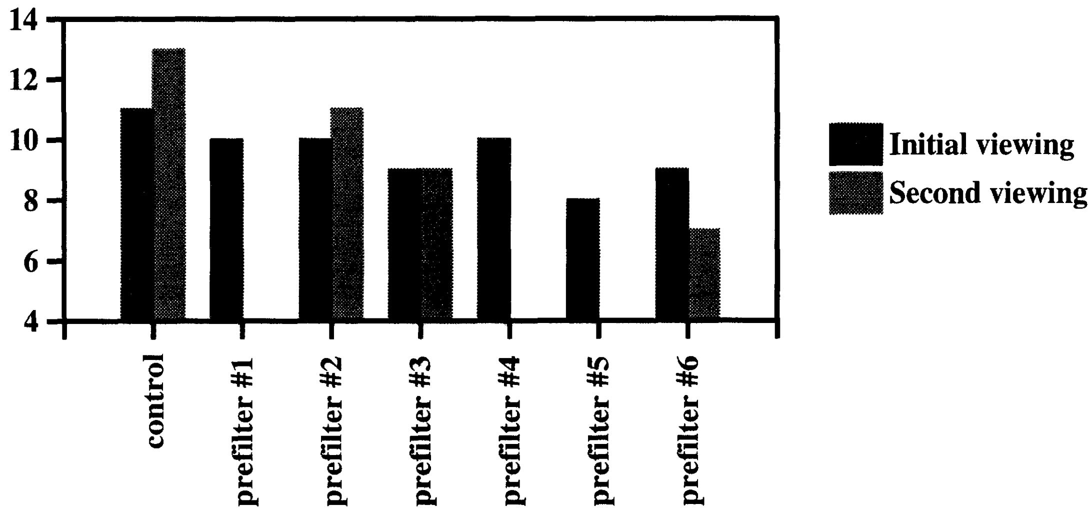

Figure 7.3: Subjective Ratings for Prefilters (Foreman)

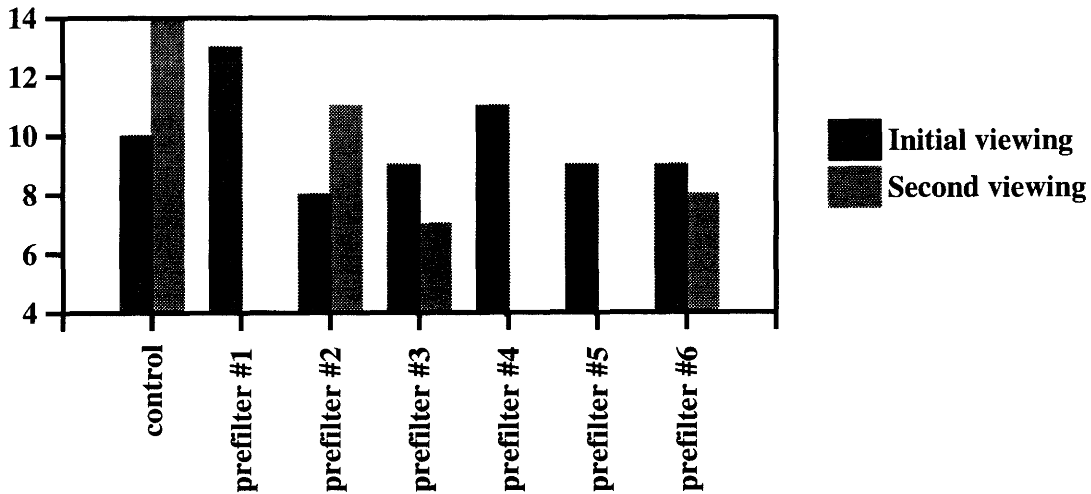

Figure 7.4: Subjective Ratings for Prefilters (Mother/Daughter)

The data show that preference was given to certain prefiltered source sequences relative to the non-filtered source. These sequences were displayed at CIF resolution when making comparisons. Lower resolutions might eliminate the need for prefiltering (noise is not as apparent), while higher resolutions (4CIF or full monitor size) would increase the importance of using prefilters. Although prefiltering data is often not as effective as postfiltering, it can be used to enhance image quality in videoconferencing when postfiltering is not possible. Postfiltering is implemented on the receiver end and is usually most effective when the encoder source sequence contains the least amount of distortion. Thus, this prefiltering process is not advised if a postprocessing algorithm is available. However, if one is transmitting encoded data to a target that does not

have postprocessing available, prefiltering provides a means to improve the quality of the target's reconstructed video sequence. Together, pre- and postprocessing can improve the quality of both the receiver's and target's decoded sequences.

Chapter 8

PostProcessing Techniques

8.1 Previous Research

There has been significant research in the area of postprocessing routines designed to mitigate the effects of noise present in coded images and sequences. Although much of the work is specific to still-image coding and non real-time applications, many algorithms provide insight into the development of a low complexity postprocessing routine for low bitrate videocoding. Lee [5] developed a means of using local statistics to reduce additive and multiplicative noise, but the noise present in most encoded video sequences isn't accurately characterized as additive or multiplicative white noise. Zakhor [6] and Stevenson [7] developed iterative approaches based on bandlimited DCT coefficients and stochastic models of source and decoded images, respectively. Such iterative techniques are too complex to implement in current real-time applications, as demonstrated by Stevenson's algorithm requiring 90 seconds processing time for a single image on a Sparc2 workstation. Minami and Zakhor [8] demonstrated that blocking artifacts could be reduced using an optimization technique on DCT coefficients when the block boundaries are clearly defined. This optimization scheme significantly reduces blocking, but relies on a full set of DCT coefficients for the entire image. Video codec standards that employ motion prediction, such as H.261 and H.263, do not INTRA-code every frame. Thus, this approach is not directly applicable to video sequences. Their scheme also ignores the fact that signals are not always slowly varying across a block boundary, and it may be favorable to retain a sharp edge along such a boundary. Tzou [9] proposed a method that uses an anisotropic filters to determine which frequencies should be suppressed across a boundary. The scheme was effective, but the goal of this research is to find a more universal algorithm that can be applied to both boundary and interior block pixels.

Ramamurthi and Gersho [10] developed a scheme that first classified each block as an edge or monotone block, and then determined the orientation (at 45 degree intervals) of detected edges. Various linear filters were then applied to each block depending on its classification. Liu and Jayant's approach [11] to mitigating these artifacts was to apply various processing techniques depending on the classification of each pixel in an image. Each pixel is first classified as an edge or non-edge pixel. Then, a multi-level median filter is applied to edge regions, a double median filter is applied to smooth non-edge regions, and a median filter is applied to the non-smooth non-edge regions. This classification scheme appropriately segments the image for different filters, but the filters are too complex for implementation on low-cost consumer applications. I have adopted similar classifications (essentially edge and non-edge), but have used different methods for classification and applied different filters to each region.

In the LO-COST algorithm, a mild lowpass filter is first applied to the entire image to mitigate blocking and generic noise. Then, both edge and object-boundary detection schemes are applied for each pixel.

For edge pixels, a sharpening, or highpass, filter is applied to enhance the edge. For non-edge pixels near an object boundary, a median filter is applied to eliminate the mosquito noise. The remainder of the non-edge pixels are not modified beyond the initial lowpass filtering. This scheme is very effective, yet it is also economical to implement for consumer applications. In subjective testing, viewers detected a significant improvement in image quality when the LO-COST algorithm was applied to decoded sequences transmitted over very low bit-rates.

8.2 Noise and Human Perception

8.2.1 Noise characteristics

It is beneficial to characterize the coding noise present in video sequences before devising a method to compensate for it. Blocking refers to the artificial edges that appear along block boundaries. Blocking occurs in motion-predicted codecs because motion vectors (MV) are not optimal motion predictors for every pixel in the associated block. "Optimal" motion vectors derived for a 16x16 pixel macroblock (MB), for example, will naturally have greater correlation with the motion of the center pixels than with that of the pixels found near the boundaries of the MB. While the H.263 standard includes optional 8x8 pixel block MVs and overlapped-block-motion compensation (OBMC) to reduce blocking artifacts, the H.261 standard requires the standard 16x16 pixel MB vectors to be used. The inability of quantized DCT coefficients to correct large errors near block boundaries causes abrupt, artificial edges that must be masked for postprocessing routines to be effective.

The other major artifacts, mosquito noise and ringing, are also caused by the quantized DCT coefficients. Mosquito noise refers to the single pixel errors that arise near sharp edges within an image. Mosquito noise is not localized to within a couple of pixels of an edge, but appears throughout the entire 8x8 pixel block when a sharp, temporally non-stationary edge passes through the block. Furthermore, these errors are not stationary between images, but appear to swarm around sharp edges over time and lead to its name -- "mosquito noise". Thus, it is the temporal and spatial non-stationary characteristics of mosquito noise that cause it to be so distracting. To gain consistency between frames and between blocks within the same frame, it is wise to apply some type of filter to every pixel of every image. Ringing occurs when a DCT coefficient is coarsely quantized causing slight halos to appear around edges. This ringing can usually be removed by applying median filters to the pixels near a sharp edge.

8.2.2 Human perception

The propensity of the human viewer to discern certain artifacts more easily than others plays a key role in the approach described in this paper. Karunasekera and Kingsbury [12] demonstrated that indeed human viewers perceive certain edge artifacts more readily than others, and developed a metric for measuring the distortion of blocking artifacts. With this motivation, I contend that the human viewer does not perceive the decoded images pixel by pixel (as SNR measures evaluate images), but rather as discernible,

interacting objects. Humans use edges to recognize these objects, but cannot easily detect noise along an object's edges. Noise surrounding these edges or within smooth monotonous regions, however, significantly distracts the viewer's attention. This noise should be masked so that the viewer can concentrate on the more important information contained in the object's edges. This seems rather intuitive and straightforward, but has significant ramifications for the LO-COST algorithm. Consistent masking is applied to mitigate noise that occurs in monotonous regions or surrounds an object, while a sharpening filter is applied to amplify the object's edges.

8.3 Image Segmentation

8.3.1 Edge detection

Edge detection presents a problem in terms of both accuracy and complexity. True edges, those naturally occurring in the image, may appear either as a sharp cutoff between two neighboring pixels or as a more slowly varying gradient. Artificial edges, created by blocking, occur between adjacent pixels of neighboring blocks. For the cases considered here, reconstructed pixel values rarely (less than 1.5%) differ by more than 15 units (over a range of 0-255) from the actual values in the original image. Thus, artificial edges created solely by block boundaries should rarely show a difference greater than 30 units between neighboring pixels. For these reasons, the nominal value of 30 was chosen as the threshold for detecting edges. However, the threshold was not compared with the difference between adjacent pixels, since true edges with moderate gradients might not show such a steep drop-off. Instead, differences were calculated between the current pixel and pixels (2 units away in the horizontal and vertical directions according to

$$e_1[i,j] = abs(s[i+2,j] - s[i,j]) \tag{8.1}$$

$$e_{2(1)}[i,j] = abs(s[i-2,j] - s[i,j]) \tag{8.2}$$

$$e_1[i,j] = abs(s[i,j+2] - s[i,j]) \tag{8.3}$$

$$e_1[i,j] = abs(s[i,j-2] - s[i,j]) \tag{8.4}$$

If any of the differences are greater than the threshold value 30, an edge is detected. Other methods were tried, including differences in neighboring pixels, but this method demonstrated the best resiliency at avoiding artificial edges while detecting the true ones. In terms of complexity, the edge detection scheme chosen is relatively simple and can be performed on an isolated region of the image.

8.3.2 Object boundary detection

The more difficult task of object boundary detection presents a much greater challenge. Objects are defined to be large connected regions within the image, such as people, tables, etc. Details within such

objects, such as eyes, ears, teeth, etc. are not defined as separate objects. Objects might be detected by defining regions as left/right of an edge, etc. However, this analysis must be performed on a relatively large region and might entail very complex algorithms. Parallel processors used in real-time applications, which often process regions of an image separately, would be unable to perform such global operations, and the complexity of the algorithm would be too costly for limited resources. Therefore, a very crude measure is used to detect actual object boundaries. The difference between pixels in diagonal directions is computed according to

$$o_{+}[i,j] = abs(s[i+n,j+n] - s[i,j]) \tag{8.5}$$

$$o_{+}[i,j] = abs(s[i+n,j+n] - s[i,j]) \tag{8.6}$$

where n is nominally 8 for CIF and 4 for QCIF formats. If either measure is greater than the threshold, roughly 50% greater than the edge detection threshold, an object boundary is assumed to exist. A threshold value of 40 produced good results in testing. For CIF formats, this computation requires 16 lines of the image to be buffered in memory. To achieve an equivalent distance using pixels chosen in horizontal or vertical directions would require 22 lines to be buffered. Thus, the pixels are chosen from diagonal directions to reduce the buffering requirements.

Figure 8.1: Non-edge pixels in monotonous regions

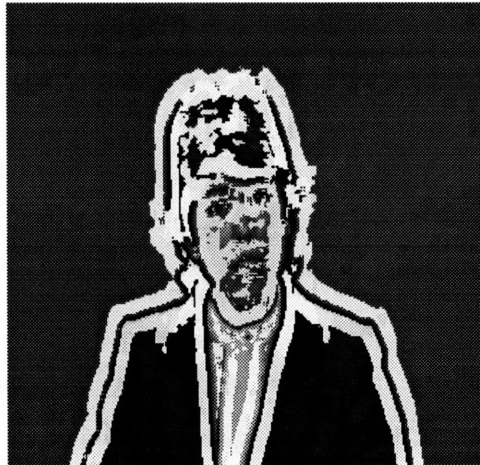

Figure 8.2: Non-edge pixels near object boundaries

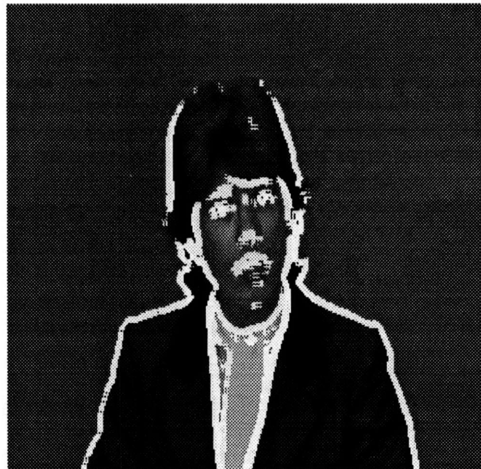

Figure 8.3: Edge pixels

With these two measures, each pixel can be classified into one of three categories: 1) not near an edge, not near an object boundary (monotonous region) 2) not near an edge, near an object boundary or 3) near an edge. Each of these classifications can be seen in the highlighted (white) portions of the images shown in Figures 8.1-8.3. Note, that the third classification locates most of the sharp details within the image and the second classification surrounds nearly all of these objects. However, since only luminance information is processed in the LO-COST algorithm, edges caused by color contrast only are not detected (i.e. black lapel on dark blue blazer).

8.4 Filters

8.4.1 Smoothing

The first step in the algorithm, smoothing the image with a lowpass filter, is performed for two reasons. The first reason is to prepare the data for the edge/object detection schemes. By smoothing the data, isolated discrepancies are less likely to signal a false edge or boundary. The new gradients between pixels

are not as steep, but the (2 pel distance used in the edge detection scheme will detect edges with smaller slopes. The second and most important reason is that of consistency. As described in subsection 2.1, human viewers tend to notice when noise is either temporally or spatially non-stationary. Even small deviations can be very obvious if their location within the picture changes ever so slightly. Therefore, it is very important to process every pixel on every frame in a consistent manner. Although not eliminated entirely, artificial edges and roaming speckle noise become much less obvious after the lowpass filter is applied.

The lowpass filter chosen for the smoothing process, L (shown below), does not possess a very sharp cutoff. Consequently, blurring can rarely be detected in the processed images without direct comparison to the original sequence, especially in the monotonous regions. Blurring becomes most noticeable near edges and object boundaries. Therefore, edge pixels and object boundaries receive further processing to mitigate the effects of the lowpass filter.

$$L_{lp} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 16 \tag{8.7}$$

Note that since this lowpass filter's coefficients contain only multiples of two, the output value can be calculated using only additions and shifts.

8.4.2 Near object boundaries

When an object boundary is detected but no immediate edge is present, a median filter is applied to further reduce mosquito noise. Originally, a 3x3 or 5x5 median filter was successfully used to reduce the mosquito artifacts. These filters require the sorting of at least nine values, a very computationally intensive task for a pipelined DSP. Due to processing constraints, this filter was replaced by either a 3x3 separable median filter or a 5-point, cross-shaped median filter (shown below). The 3x3 separable median filter may be efficiently computed by sorting each 3-pixel column separately, and then taking the median value of the three column medians. To compute separable median values for each pixel in a raster scan, only two 3-value sorts are required per pixel: one sorting of a new 3-pixel column and one additional sorting of the median values of the three columns. Alternatively, the 5-point median filter requires accessing and sorting five values for each pixel. The 3x3 separable median filter provides better subjective noise masking, while the 5-point cross was superior in SNR measurements.

$$M = median \begin{bmatrix} & X & \\ X & X & X \\ & X & \end{bmatrix} \tag{8.8}$$

This filter also serves to compensate for some the blurring caused by the lowpass filter. Subjective testing has revealed that viewers are less likely to notice blurring near object boundaries when median filters were applied to the surrounding pixels. An explanation could be that the median filtering restores some con-

trast near an edge of which the gradient was smoothed by the lowpass filter. The median filter is not applied to all non-edge pixels, because it enhances certain block boundaries and requires more computation.

8.4.3 Near edges

Edges must be treated differently than their surrounding areas because viewers use edges to discern objects. Since blurring is especially obvious when edges are smoothed, we apply a "sharpening" (highpass) filter, H, to the edges to gain contrast. The combination of the lowpass and the sharpening filters effectively results in a 5x5 filter (shown below). The sharpening filter amplifies both the edge signal and noise, but is rarely applied to artificial edges created by block boundaries due to the choice of the edge-detection threshold. Overall, the sharpening of edges provides the viewer with greater contrast to discern objects; any amplified noise is only mildly objectionable in such detailed areas.

$$H = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 16 & -1 \\ -1 & -1 & -1 \end{bmatrix} / 8 \tag{8.9}$$

$$L*H = \begin{bmatrix} -1 & -2 & -3 & -2 & -1 \\ -2 & 6 & 4 & 6 & -2 \\ -3 & 4 & 120 & 4 & -3 \\ -2 & 6 & 4 & 6 & -2 \\ -1 & -2 & -3 & -2 & -1 \end{bmatrix} / 128 \tag{8.10}$$

Note that the sharpening filter, H, can also be implemented using only additions and shifts.

8.5 Results

8.5.1 Objective data

It is difficult to measure an algorithm's success in the enhancement of video sequences. Peak SNR measurements serve as an initial benchmark to determine whether a routine has greatly altered the error rate of a decoded sequence. However, viewers are more concerned with information content than the exact matching between original and decoded images. We have already conjectured that noise found along edges is often not detected, yet significantly lowers the PSNR measure due to the larger range of error. To gain a deeper understanding of the effects this algorithm, the error distributions were tabulated for the LO-COST algorithm and several filters used for comparison, including the D-filter introduced by Kundu and Wu [13]. This allows the differences among PSNR data to be analyzed with greater insight. Both measurements are included in Tables 8.1 and 8.2, shown below.

Claire Sequence w/ Given Filter	Average PSNR (dB)	Percentage of pixels with a change in luminance of					
		[0]	[1]	[2-3]	[4-8]	[9-16]	17+
Decoded	40.67	24.68	38.23	27.08	8.97	1.01	0.03
3x3 Median (S)	39.65	25.17	38.90	26.82	7.87	1.02	0.23
3x3 Median (N)	39.39	25.49	39.25	26.39	7.51	1.07	0.28
5x5 Median (S)	37.19	25.46	39.19	25.78	7.53	1.43	0.62
3x3 D-filter	35.03	25.55	39.07	24.97	7.19	1.97	1.24
3x3 LowPass	38.94	25.14	38.80	26.13	7.85	1.58	0.50
5-pt Median	40.43	25.01	38.72	27.11	8.11	0.93	0.12
LO-COST	40.72	25.48	39.19	26.38	7.79	1.10	0.06
LO-COST but no sharpening	40.98	25.51	39.34	26.55	7.62	0.93	0.05

N = non-separable S = separable

Table 8.1: Postprocessing Data for "Claire" coded with H.261 @ 112 Kbps

Akiyo Sequence w/ Given Filter	Average PSNR (dB)	Percentage of pixels with a change in luminance of					
		[0]	[1]	[2-3]	[4-8]	[9-16]	17+
Decoded	34.58	15.23	26.66	29.21	21.64	6.00	1.26
3x3 Median (S)	30.71	15.51	27.18	28.89	19.04	5.89	3.50
3x3 Median (N)	30.62	15.69	27.38	28.74	18.38	6.01	3.80
5x5 Median (S)	27.69	15.34	26.74	27.56	18.07	6.51	5.78
3x3 D-filter	27.52	14.76	26.18	27.91	17.54	6.38	7.23
3x3 LowPass	31.44	15.56	27.29	28.81	17.79	6.27	4.28
5-pt Median	32.89	15.47	27.21	29.24	19.99	5.78	2.30
LO-COST	34.14	15.96	27.72	29.04	19.33	6.28	1.68
LO-COST but no sharpening	34.31	16.06	27.94	29.35	19.22	5.82	1.61

Table 8.2: Postprocessing Data for "Akiyo" coded with H.263 @ 24 Kbps

Each of the filters increase the percentage of pixels that are within one intensity unit of the original image. However, each of the filters/algorithms also increase the percentage of pixels with errors greater than 16 units. Such large errors are undesirable, but are often located adjacent to an edge and are practically undetectable. The LO-COST algorithm's SNR performance is superior to any of the filters applied individually, and is superior to that of the decoded image in the "Claire" sequence. In these sequences, the SNR measurements of the block-border pixels is greater than 1 dB below that of the interior pixels. The LO-COST algorithm lowers that gap substantially, thus reducing blocking artifacts. The data also show that the sharpening filter detracts from overall SNR, but it is an important component because it adds clarity and key visual information for the viewer. Objective analysis alone cannot measure the success of this algorithm, since viewers ultimately make the final decision.

8.5.2 Subjective performance

Figures 8.4-8.9 demonstrate the improvements over the decoded images. The improvement in temporal consistency cannot be shown here, but is evident when viewing the video sequences. The images shown

in Figures 8.4-8.6 are taken from a video sequence encoded/decoded using the H.261 standard at 112 Kbps. The images shown in Figures 8.7-8.9 are taken from a video sequence encoded using the H.263 standard at 24 Kbps. Several independent observers were asked to judge the overall image quality of sequences displayed side by side with different postprocessing filters applied. Every viewer preferred the LO-COST algorithm to each of the filters listed above as well as the unaltered decoded sequence.

Notice that the outlines of all the objects have been cleaned of mosquito noise. The speckle noise and blocking that appear in highly detailed regions (i.e. face) are masked, allowing the viewer to discern the key features. The edges, including those contained in small details (i.e. earrings, eyes, etc.), appear sharp and accurate. The economical LO-COST algorithm has retained the visual information while masking the distracting noise.

Figure 8.4: "Claire" before and after LO-COST is applied (H.261 @112 Kbps)

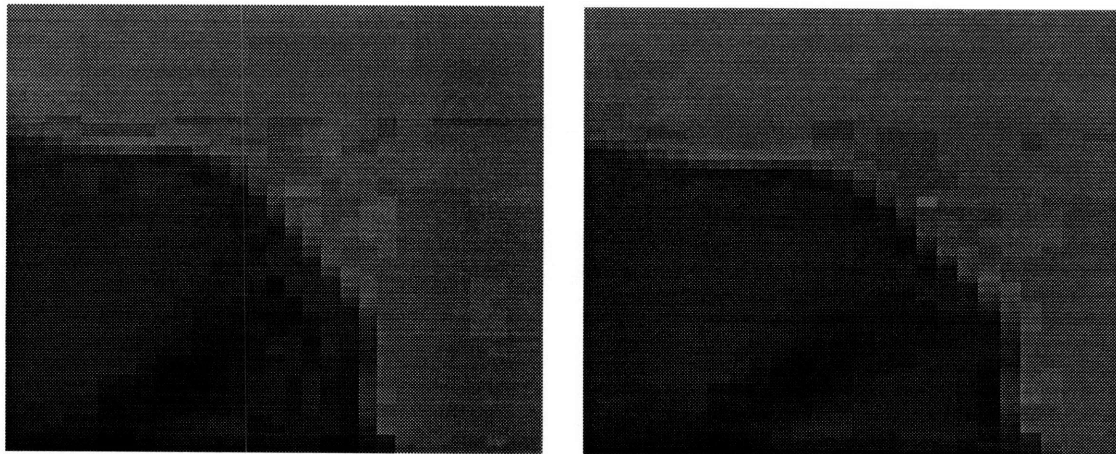

Figure 8.5: Enlarged shoulder ("Claire") before and after LO-COST is applied

Figure 8.6: Enlarged face ("Claire") before and after LO-COST is applied

Figure 8.7: "Akiyo" before and after LO-COST is applied (H.263 @24 Kbps)

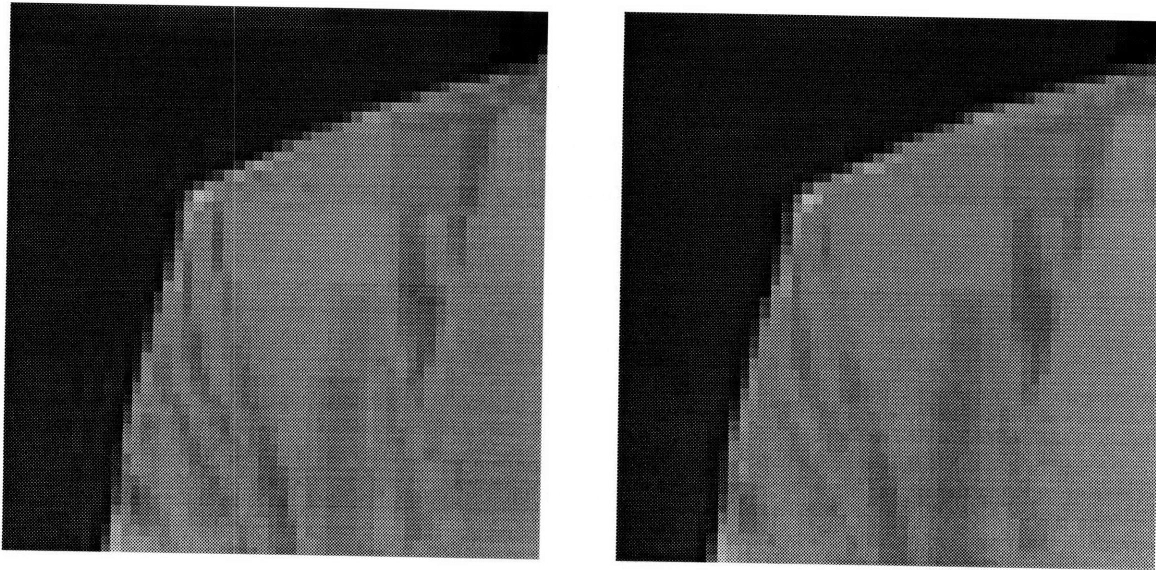

Figure 8.8: Enlarged shoulder ("Akiyo") before and after LO-COST is applied

Figure 8.9: Enlarged face ("Akiyo") before and after LO-COST is applied

8.6 Modifications

Since complexity was a key issue in the design of this algorithm, certain simplifications of the algorithm were also considered. The first simplification involved eliminating the object boundary detection step and applying the median filter to all pixels for which an edge was not detected. This eliminated the need to retain 16 lines of data in memory and to perform the object boundary decision. One of the benefits of the selective application of the median filter was that it need not be computed for every pixel (never more than

30% in the sequences tested). However, if the 3x3 separable median filter routine was written such that it depended on computing a value for every pixel (i.e. sorting one new 3-value column for every pixel rather than sorting three 3-value columns for selected pixels), this modification merely eliminates a decision step and the need to retain 16 lines of data. Both of these are critical issues when facing program and data cache constraints. The only drawback occurs when blocking becomes more noticeable.

Another simplification involves the removal of the smoothing filter and modification of the sharpening filter, in addition to the removal of the object boundary detection. In this scenario, a 5x5 linear filter, L*H, is applied to the edge pixels, and a 3x3 separable median filter is applied to non-edge pixels. This still eliminates much of the mosquito noise, but does not provide a consistent temporal masking of the noise located near edges or across block boundaries.

A third simplification to the algorithm's implementation involved checking for edges only on every other pixel of every other line. For each pixel used in the decision scheme, two edge measures would be calculated, e1 and e3, and used in conjunction with previous pixels' edge tests. After initial smoothing with the lowpass filter, each decision process would classify four adjacent pixels as edge or non-edge pixels. The edge pixels are then processed with the sharpening filter and non-edge pixels are processed with the cross-shaped median filter. A small array is required to retain the results of the vertical tests from the previous line. This modification reduces the number of edge detection tests by a factor of 8 with similar performance.

A final modification was made with respect to the location of pixels within the blocks. As noted in section 5.1, blocks' border pixels contain greater errors than interior pixels. To compensate, a stronger lowpass filter, L2, was applied to the 28 border pixels of each 8x8 block, the common DCT block size for applicable videoconferencing standards. The original filter, L, is still applied to the remaining 36 interior pixels. This alternative filter with a lower cutoff frequency provided greater masking of the blocking artifacts. Objective data for each of the modifications is shown in Table 8.3.

Claire Sequence w/ Given Filter	Average PSNR (dB)	Percentage of pixels with a difference in intensity of					
		[0]	*[1]*	*[2-3]*	*[4-8]*	*[9-16]*	*17+*
Locost w/o object detection	40.70	25.56	39.24	26.25	7.76	1.12	0.06
LO-COST w/o smoothing	40.70	24.97	38.54	26.95	8.44	1.05	0.04
LO-COST w/ 4-pixel classification	40.52	25.55	39.28	26.28	7.68	1.11	0.10
LO-COST w/ border pixel filter	40.57	25.72	39.34	26.03	7.64	1.18	0.09

Table 8.3: Postprocessing Data for "Claire" coded with H.261 @ 112 Kbps

The LO-COST algorithm provides effective masking for various types of noise found in low bit-rate encoded video sequences. It can be easily implemented on parallel digital signal processors, which contain

both cache and processing constraints, for use with consumer videoconferencing applications. This algorithm provides substantial improvement in visual quality with only modest computational complexity.

Chapter 9

Conclusion

Each algorithm proposed above, except for the standardizations of Chapter 3, were designed to either improve subjective image quality or decrease the computational complexity of implementing the videocoding standards. The effectiveness of the proposed algorithms varies significantly and will therefore be treated separately.

The attempt to equalize the signal-to-noise ratio within coded blocks (Chapter 6) was the least successful experiment conducted. While the differences in SNR between regions within a block were narrowed, the SNR of both the entire block and each individual region of the block declined. The constraints placed on the experiment by the videocoding standard could have contributed to this failure, and the goal of equalizing image quality throughout each blocks remains. The question is whether to modify the coding algorithm to accomplish this goal or correct the quality differential via postprocessing. A change in the coding standards' quantization algorithm is certainly a possibility, since such "quantization tables" have been used for JPEG compression for some time. Without modifying the standard, however, attempts to preference certain DCT coefficients appear to offer little, if any, gain in performance.

The rate control studies demonstrated the importance of defining a clear goal for videocoding. Generally, one does not want to provide equal numbers of bits to code different regions of the image, since many areas are stationary and provide little information to the viewer. However, equivalent quantization parameters for each region of an image can lead to pronounced delays when a scene change consumes an unusually large number of bits. The compromise, presented above, between these approaches modifies the quantization parameter several times during the encoding of an image by comparing bit usage within the current image to estimated consumption. This approach successfully accommodates scene changes by allowing fairly rapid adjustments in the QP when consecutive images differ while appropriating more bits to the regions with greater spatial and temporal frequencies. One dilemma that was not considered in the research occurs when the portions of a frame with the highest spatial frequencies are not important to the viewer (background distractions). If these are stationary, it is sometimes possible to avoid coding these by distinguishing which areas have significant motion/changes. However, if these background regions are not stationary, it becomes extremely difficult to limit their consumption of bits relative to the important information. One solution to this problem might be to let the viewers segment the portion of the frame that the encoder should encode (or preference in coding). The areas outside this "preferred" region would either be encoded with a much coarser QP or not be encoded at all. This would require significant modifications to the videocoding applications, but would allow the viewer, who contains the ultimate knowledge regarding what is important, to decide how to allocate the available bits. Encoding the relevant information within an image rather than exact reproduction is a more accurate description of the goal of videocoding.

The motion estimation studies demonstrated that considerably fewer computations are needed to derive accurate motion vectors for each macroblock. The metrics introduced reduced the number of computations required of motion estimation routines by up to 50% while only slightly decreasing objective image quality. Thus, applications or devices that are limited in processing capabilities or power consumption can significantly benefit from the proposed algorithms. Furthermore, these studies corroborate that much H.263's improved video quality arises from using the weighted average of predicted pixel values rather than the improved accuracy of using individual block motion vectors or PB delta components. These findings significantly reduce the number of motion searches that should be performed by the encoding algorithm, allowing for more useful routines (i.e. pre/postprocessing) to be included in application. Finally, the use of chrominance information in motion estimation algorithms did not appear to improve objective image quality. Although these findings do not preclude routines that use chrominance information to their advantage, the author believe that chrominance information will provide only minor improvements to image quality using these standards.

The pre- and postprocessing experiments demonstrated that much of the potential improvement in image quality can be gained outside the encoding algorithm altogether. Many complex preprocessing schemes have been proposed which attempt to eliminate noise present in the source, such as the 3-D nonlinear filters presented by Bauer [14]. However, given the C8x's constraints on memory and processing power, the proposed preprocessors presented in Chapter 7 were limited to linear lowpass filters; and the intent was not to reduce noise in the source, but to make the source more amenable to coding. The results of the preprocessing experiments demonstrate that lowpass filters can be used as preprocessing routines to improve the quality of the decoded sequences when no postprocessing is available. However, postprocessing routines appear to be most effective when the coder uses the most accurate source sequence possible.[1] Therefore, the preprocessing filters proposed are not recommended if postprocessing is available. The postprocessing algorithm realized the greatest improvement in subjective image quality of all the proposed modifications. It effectively masked much of the blocking, mosquito, and ringing effects evident in decoded sequences encoded using very low bitrates. Given the limited processing resources, the postprocessing algorithm used very simple means to detect objects and apply appropriate nonlinear filters. The imminent rise in processing power per dollar will allow much more complex schemes to be used in consumer applications in the near future, but the viewer-oriented approach to postprocessing will likely grow stronger. Preprocessing routines might be most effective when eliminating noise present in a source sequence, but the goal of postprocessing routines is to convey information to the viewer. Thus, stochastic models and least square error measurements are often misguided in their aims at objective improvement. Psychovisual experiments will most certainly play an increased role in this area of research.

1. Note that temporal preprocessing filters that reduce additive or multiplicative noise will most likely improve the effectiveness of postprocessing routines.

These experiments raise many more questions than they answer. What information in an image is relevant? How should objects be defined, and how can they be differentiated from mere details within an object? Should the user interactively participate in deciding which parts of the frames are most relevant? These questions are the focus of many new videocoding proposals currently under investigation. The LOCOST postprocessing algorithm attempted to answer these questions via a crude viewer-oriented model that can be implemented very inexpensively. Its most novel idea was that various filters should be applied to different regions based on how the viewer is most likely to comprehend these regions (i.e. as an object, an edge, etc.) rather than solely on objective local characteristics. The segmentation of regions was based on local objective measures, but these measures served merely to define regions from a viewer's perspective. More sophisticated routines in all aspects of the encoding process along with the luxury of greater processing capabilities are needed before very-low-bitrate videocoding achieves prominent use in the consumer sector.

Acknowledgments

This research was conducted during an internship with the Compression Lab of the DSP R&D Center at Texas Instruments Inc., Dallas, Texas. I wish to thank Jeremiah Golston for giving me the opportunity to work on such an interesting research topic, Wanda Gass for her professional support and encouragement, and Jennifer Webb and Judy Liao for many enlightening discussions and feedback on my work. Thanks also to Amy Singer for helping me to proofread and clarify explanations throughout the paper. My master's thesis advisor at MIT, Professor V. Michael Bove, provided guidance through his many thoughtful suggestions; his career consulting and sincere enthusiasm continue to serve as a source of inspiration for me. I also wish to thank the US Navy for providing the Scholarship Leave program to allow commissioned officers to conduct scientific research. This research is not only beneficial to the country, but sharpens the officers, and in turn, provides well-educated leaders for the men and women of our armed forces.

None of this would be possible without the support of two exceptional parents, Larry and Harriet, a generous brother and (newly acquired) sister-in-law, and countless teachers throughout the years. I would also like to recognize several friends who have taught me many lessons about life itself. Todd Hay has shown me the importance of high standards and continual learning. Manuel Munoz has taught me the importance of dreams, while Fionn Mayes has moved me through his uncompromising character and determination despite adversity. David Steel has taught me the importance of friendship and of living life to its fullest. Lieutenant Waller demonstrated the power of uncompromising integrity, and set a high standard for professionalism and leadership that I hope to one day achieve. And Angela Lee, whose selfless devotion to assisting the less fortunate has provided a moral bearing along this most rocky journey. These are among the many who have shaped the person I have become. Thank you once again.

References

[1] Liou, M. "Overview of the Px64 kbit/s video coding standard," *Communications of the ACM*. Vol. 34, No. 4, April 1991.

[2] Schaphorst, R. "Draft H.263," ITU Telecommunication Standardization Sector, Leidschendam, November 1995.

[3] Reeves, H. C. and J. S. Lim. "Reduction of Blocking Effect in Image Coding," *Optical Engineering*. Vol. 23, No. 1, pp. 34-37, January 1984.

[4] Ngan, K. N., D. W. Lin, and M. L. Liou. "Enhancement of Image Quality for Low Bit Rate Video Coding," *IEEE Transactions of Circuits and Systems*. Vol. 38, No. 10, pp. 1221-1225, October 1991.

[5] Lee, J. S. "Digital Image Enhancement and Noise Filtering by Use of Local Statistics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 2, No. 2, pp. 165-168, March 1980.

[6] Zakhor, A. "Iterative Procedures for Reduction of Blocking Effects in Transform Image Coding," *IEEE Transactions on Circuits and Systems*. Vol. 2, No. 1, pp. 91-95, March 1992.

[7] Stevenson, R. L. "Reduction of Coding Artifacts in Transform Image Coding," *IEEE Proceedings for the International Conference on Acoustics, Speech, and Signal Processing*. Vol. 5, pp. 401-404, 1993.

[8] Minami, S., and A. Zakhor. "An Optimization Approach For Removing Blocking Effects in Transform Coding," *IEEE Transactions on Circuits and Systems For Video Technology*. Vol. 5, No. 2, pp. 74-82, April 1995.

[9] Tzou, K. H. "Post-filtering of Transform-coded Images," *Applications of Digital Image Processing XI, Proceedings of SPIE*. Vol. 974, pp. 121-126, 1988.

[10] Ramamurthi, B., and A. Gersho. "Nonlinear Space-Variant Postprocessing of Block Coded Images," *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol 34, No. 5, pp. 1258-1267, October 1986.

[11] Liu, T. S., and N. Jayant. "Adaptive Postprocessing Algorithms For Low Bit Rate Video Signals," *IEEE Transactions on Image Processing*. Vol. 4, No. 7, pp. 1032-1035, July 1995.

[12] Karunasekera, S. A. and N. G. Kingsbury. "A Distortion Measure For Blocking Artifacts in Images Based on Human Visual Sensitivity," *SPIE Proceedings on Visual Communications and Image Processing*. Vol. 2094, pp. 474-486, November 1993.

[13] Kundu, A., and W.R. Wu. "Double-window Hodges-Lehman (D) Filter and Hybrid D-median Filter For Robust Image Smoothing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 37, No. 8, pp. 1293-1298, August 1989.

[14] Bauer. "3-D Nonlinear Recursive Digital Filter For Video Image Processing," *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*. pp. 494-497, May 1991.

Appendix A

H.261 Modified Code

A.1 File "P64.c"

```
/***
File changed by:
   Brian Davison
   21 September 1995

Changes:
   1) Created p64EncodeSec() to allow quantization to be evaluated
      only after every three GOBs (1 section) similar to MVP code.
   2) Created p64EncodeGOB2() to account for change #1.
   3) Option for ThreeStep or OneAtATime motion estimation inserted.
   4) Added a ASAD variable to DEFs and inserted new decision
         thresholds to the p64EncodeGOB2() routine.
   5) Added Frame-Dropping to Rate Control.  Changed buffer size to
         3 * (average # of bits transmitted during each frame encoding).
 It drops the next frame if Buffer Contents become greater than
 2 * (aver # of bits ...) immediately after a frame is encoded.
   6) Added new quantization control in p64EncodeSec2() and
         p64EncodeSec3().  Also added new array to record the
         quantization level for each section of the previous frame,
         SecQuant[], and a variable to record the # of encoded bits in
 the previous section of the current frame, PrevSec_enc_bits.
 SecQuant is initialized to GQuant after the first Quantization
 estimation.
   7) Added Defs for p64EncodeSec2 and p64EncodeSec3.

   8) Sends residual images to output file.
   12) Added ability to control MaxQuant (-u).
***/


/*************************************************************
Copyright (C) 1990, 1991, 1993 Andy C. Hung, all rights reserved.
PUBLIC DOMAIN LICENSE: Stanford University Portable Video Research
Group. If you use this software, you agree to the following: This
program package is purely experimental, and is licensed "as is".
Permission is granted to use, modify, and distribute this program
without charge for any purpose, provided this license/ disclaimer
notice appears in the copies.  No warranty or maintenance is given,
either expressed or implied.  In no event shall the author(s) be
liable to you or a third party for any special, incidental,
consequential, or other damages, arising out of the use or inability
```

```
to use the program for any purpose (or the loss of data), even if we
have been advised of such possibilities.  Any public reference or
advertisement of this source code should refer to it as the Portable
Video Research Group (PVRG) code, and not by any author(s) (or
Stanford University) name.
*************************************************************/
/*
************************************************************
p64.c

This is the file that has the "main" routine and all of the associated
high-level routines.  This has been kludged from the JPEG encoder, so
there is more extensibility than is really necessary.

************************************************************
*/

/*LABEL p64_1g.c */

#include <stdio.h>
#include "globals.h"

/*PUBLIC*/

int main();
extern void p64EncodeSequence();
extern void p64EncodeFrame();
extern void p64EncodeSec();
extern void p64EncodeSec2();
extern void p64EncodeSec3();
extern void p64EncodeGOB();
extern void p64EncodeGOB2();
extern void p64DecodeSequence();
extern int p64DecodeGOB();
extern void PrintImage();
extern void PrintFrame();
extern void MakeImage();
extern void MakeFrame();
extern void MakeFstore();
extern void MakeStat();
extern void MakeRate();
extern void SetCCITT();
extern void Help();
extern void MakeFileNames();
extern void VerifyFiles();

static void ExecuteQuantization();
static void p64EncodeMDU();
static void ReadCompressMDU();
static void WriteMDU();
```

```
static void DecodeSaveMDU();
static int DecompressMDU();
static void StoreResidual();
static void StoreResidualOut();
static void WriteResidual();
static void WriteResidualOut();

/*PRIVATE*/

#define  SwapFS(fs1,fs2)  {FSTORE  *ftemp;ftemp  =  fs1;fs1  =  fs2;fs2  =
ftemp;}

IMAGE *CImage=NULL;
FRAME *CFrame=NULL;
FSTORE *CFS=NULL;
FSTORE *OFS=NULL;
STAT *CStat=NULL;
STAT *RStat=NULL;
RATE *RCStore=NULL;

int BlockJ[] = {0,0,0,0,1,2};
int BlockV[] = {0,0,1,1,0,0};
int BlockH[] = {0,1,0,1,0,0};

char *DefaultSuffix[]={".Y",".U",".V"};

/* CCITT p*64 Marker information */

int TemporalReference=1;
int TemporalOffset=0;
int PType=0x0;
int Type2=0x0;
int MType=0x0;
int GQuant=8;
int MQuant=8;
int MVDH=0;
int MVDV=0;
int VAR=0;
int VAROR=0;
int MWOR=0;
int ASAD=0;
int LastMVDV=0;
int LastMVDH=0;
int CBP=0x3f;
int ParityEnable=0;
int PSpareEnable=0;
int GSpareEnable=0;
int Parity=0;
int PSpare=0;
int GSpare=0;
```

```
int GRead=0;
int MBA=0;
int LastMBA=0;

int LastMType=0; /* Last MType */

/* System Definitions */

int ImageType=IT_NTSC;

int NumberMDU=0;
int CurrentMDU=0;
int NumberGOB=0;
int CurrentGOB=0;

int CurrentFrame=0;
int StartFrame=0;
int LastFrame=0;
int PreviousFrame=0;
int NumberFrames=0;
int TransmittedFrames=0;
int FrameRate=30;

int FrameSkip=1;

/* Stuff for RateControl */

int FileSizeBits=0;
int Rate=0;
int BufferOffset=0;  /*Number of bits assumed for initial buffer. */
int QDFact=1;
int QOffs=1;
int QUpdateFrequency=11;
int QUse=0;
int QSum=0;

/* Some internal parameters for rate control */

#define DEFAULT_QUANTIZATION 8

int InitialQuant=0;

/* New Changes for Quantization Control on MVP */

int skipped = 0;
int SecQuant[];
int SecBits[];
int PrevSec_enc_bits;
int MaxQuant = 14;
int MaxSpike = 16;
```

```
/* Debugging Info */

int Debug = 0;
char *DebugFilePrefix = NULL;
char DecFileName[30];
char Res1FileName[30];
char Res2FileName[30];
FILE *RF1, *RF2;
short **yResidualImage, **uResidualImage, **vResidualImage;
short **yResidualImageOut, **uResidualImageOut, **vResidualImageOut;
int height, width;

int qinfo = 0;
unsigned char QuantP[18][2];
FILE *QF;
char QuantFileName[30];

/* SAD Thresholds for coding blocks */

int no_motest_thresh = No_MotEst_Thresh;
int not_coded_thresh = Not_Coded_Thresh;

/* DCT Coefficient Thresholds for coding blocks */

int CBPThreshold=1;  /* abs threshold before we use CBP */

/* MC Threshold for coding blocks through filter*/

#define D_FILTERTHRESHOLD 6

/* Intra forced every so many blocks */

#define SEQUENCE_INTRA_THRESHOLD 131

/* Parser stuff */

extern double Memory[];

/* Stuff for encoding (temporary integer storage) */

static int inputbuf[10][64];
static int output[64];

/* Stuff for Motion Compensation */

extern int SearchLimit;
extern int bit_set_mask[];
extern int MeX[];
extern int MeY[];
```

```
extern int MeVal[];
extern int MeOVal[];
extern int MeVAR[1024];
extern int MeVAROR[1024];
extern int MeMWOR[1024];
extern int MeASAD[1024];
extern int MeSPK[1024];
extern int MeOSPK[1024];
int ME_ThreeStep=0;
int spike = 0;
int orspike = 0;

/* Book-keeping stuff */

int ErrorValue=0;
int Loud=MUTE;
int Trace=NULL;
int Verbose=0;
int ForceCIF=0; /* Forces CIF format - superset will hold any other for-
mat */

int MQuantEnable;
int UseQuant;

/* Statistics */

int NumberNZ=0;
int FirstFrameBits=0;
int NumberOvfl=0;
extern int MotionVectorBits;
extern int MacroAttributeBits;
extern int CodedBlockBits;
int YCoefBits=0;
int UCoefBits=0;
int VCoefBits=0;
extern int EOBBits;
int TotalBits,LastBits;
int MacroTypeFrequency[10];
int YTypeFrequency[10];
int UVTypeFrequency[10];

unsigned char **LastIntra;   /* Used for intra forcing (once per 132
frames) */

/* Coding control */

int QuantMType[] =    {0,0,0,0,0,0,0,0,0,0}; /* Quantization used */
int CBPMType[] =      {0,0,1,0,0,1,1,0,1,1}; /* CBP used in coding */
int IntraMType[] =    {1,1,0,0,0,0,0,0,0,0}; /* Intra coded macroblock */
int MFMType[] =       {0,0,0,0,1,1,1,1,1,1}; /* Motion forward vector
```

```
used */
int FilterMType[] =    {0,0,0,0,1,1,1,1,1,1}; /* Filter flags =  MV case
*/
int TCoeffMType[] =   {1,1,1,0,0,1,1,0,1,1}; /* Transform coeff. coded */

/* int QuantMType[] =    {0,1,0,1,0,0,1,0,0,1};  */ /* Quantization used
*/
/* int FilterMType[] =   {0,0,0,0,0,0,0,1,1,1};  */ /* Filter flags */
/* int CBPMType[] =     {0,0,1,1,0,1,1,0,1,1};  */ /* CBP used in coding
*/
/* int TCoeffMType[] =   {1,1,1,1,0,1,1,0,1,1};   */ /* Transform coeff.
coded */

/* DCT Stuff */
/* Functional Declarations */

vFunc *UseDct = ChenDct;
vFunc *UseIDct = ChenIDct;

#define DefaultDct (*UseDct)
#define DefaultIDct (*UseIDct)

#define BufferContents() (mwtell() + BufferOffset -\
  (((CurrentGOB*NumberMDU)+CurrentMDU)\
   *Rate*FrameSkip\
  /(NumberGOB*NumberMDU*FrameRate)))
#define BufferSize() (100*Rate*FrameSkip/FrameRate) /*In bits */

/*START*/
/*BFUNC

main() is the first routine called by program activation. It parses
the input command line and sets parameters accordingly.

EFUNC*/

int main(argc,argv)
     int argc;
     char **argv;
{
  BEGIN("main");
  int i,p,s,r;

  MakeImage();    /* Initialize storage */
  MakeFrame();
  MakeFstore();
  inithuff();     /* Put Huffman tables on */
  initmc();       /* Put Motion Comp stuff on */
  if (argc==1)
    {
```

```
      Help();
      exit(-1);
    }
  for(s=0,p=0,r=0,i=1;i<argc;i++)
    {
      if (!strcmp("-NTSC",argv[i]))
{
  ImageType = IT_NTSC;
}
      else if (!strcmp("-CIF",argv[i]))
{
  ImageType = IT_CIF;
}
      else if (!strcmp("-QCIF",argv[i]))
{
  ImageType = IT_QCIF;
}
      else if (*(argv[i]) == '-')
 {
  switch(*(++argv[i]))
     {
    case 'a':
      CurrentFrame = atoi(argv[++i]);
      StartFrame=CurrentFrame;
      break;
    case 'b':
      LastFrame = atoi(argv[++i]);
      break;
     case 'c':
      ForceCIF=1;
       break;
     case 'd':
      CImage->p64Mode |= P_DECODER;
       break;
    case 'D':
      Debug = atoi(argv[++i]);
      DebugFilePrefix = argv[++i];
      break;
    case 'f':
      FrameRate = atoi(argv[++i]);
      break;
    case 'g':
      qinfo = 1;
      break;
    case 'i':
      SearchLimit = atoi(argv[++i]);
      BoundValue(SearchLimit,1,31,"SearchLimit");
      break;
    case 'k':
      FrameSkip = atoi(argv[++i]);
```

```
        break;
      case 'l':
        Loud = atoi(argv[++i]);
        break;
      case 'n':
        not_coded_thresh = (atoi(argv[++i]));
        no_motest_thresh = (atoi(argv[++i]));
        MaxSpike = (atoi(argv[++i]));
        break;
      case 'p':
        ParityEnable=1;
        break;
      case 'q':
        InitialQuant=atoi(argv[++i]);
        BoundValue(InitialQuant,1,31,"InitialQuant");
        break;
      case 'r':
        Rate = (atoi(argv[++i]));
        break;
      case 's':
        CImage->StreamFileName = argv[++i];
        break;
      case 't':
        ME_ThreeStep = 1;
        break;
      case 'u':
        MaxQuant = (atoi(argv[++i]));
        break;
      case 'v':
        Verbose=1;
        break;
      case 'x':
        FileSizeBits = (atoi(argv[++i]));
        break;
      case 'y':
        UseDct = ReferenceDct;
        UseIDct = ReferenceIDct;
        break;
      default:
        WHEREAMI();
        printf("Illegal Option %c\n",*argv[i]);
        exit(ERROR_BOUNDS);
        break;
      }
  }
        else
  {
    strcpy(CFrame->ComponentFilePrefix,argv[i]);
  }
      }
```

```
  if (qinfo)
    {
      sprintf(QuantFileName, "%s.q", CImage->StreamFileName);
      if ((QF = fopen(QuantFileName, "w")) == NULL)
{
  printf("Can't open quant output file: %s", QuantFileName);
  qinfo = 0;
}
      printf("Quantization Info FileName: %s\n", QuantFileName);
    }

  if (!CImage->StreamFileName)
    {
      if (!(CImage->StreamFileName =
    (char *) calloc(strlen(CFrame->ComponentFilePrefix)+6,
    sizeof(char))))
{
  WHEREAMI();
  printf("Cannot allocate string for StreamFileName.\n");
  exit(ERROR_MEMORY);
}
      sprintf(CImage->StreamFileName,
      "%s.bit",CFrame->ComponentFilePrefix);
    }
  else
    sprintf(CImage->StreamFileName, "%s.bit", CImage->StreamFileName);

  switch(ImageType)
    {
    case IT_NTSC:
      PType=0x04;
      PSpareEnable=1;
      PSpare=0x8c;
      break;
    case IT_CIF:
      PType=0x04;
      break;
    case IT_QCIF:
      PType=0x00;
      break;
    default:
      WHEREAMI();
      printf("Image Type not supported: %d\n",ImageType);
      break;
    }

  if (ImageType == IT_CIF)
    {height = 288;
     width = 352;}
```

```
  else if (ImageType == IT_QCIF)
    {height = 144;
     width = 176;}

  if (Debug)
    {
      if (((Debug/10)%10) == 1)
{
  yResidualImage          = (short **) calloc(height, sizeof(short *));
  for(i=0; i<height; i++)
    yResidualImage[i]     =  (short *) calloc(width, sizeof(short));
  uResidualImage          = (short **) calloc(height, sizeof(short *));
  for(i=0; i<height; i++)
    uResidualImage[i]     =  (short *) calloc(width, sizeof(short));
  vResidualImage          = (short **) calloc(height, sizeof(short *));
  for(i=0; i<height; i++)
    vResidualImage[i]     =  (short *) calloc(width, sizeof(short));
}
      if (((Debug/100)%10) == 1)
{
  yResidualImageOut       = (short **) calloc(height, sizeof(short *));
  for(i=0; i<height; i++)
    yResidualImageOut[i] =  (short *) calloc(width, sizeof(short));
  uResidualImageOut       = (short **) calloc(height, sizeof(short *));
  for(i=0; i<height; i++)
    uResidualImageOut[i] =  (short *) calloc(width, sizeof(short));
  vResidualImageOut       = (short **) calloc(height, sizeof(short *));
  for(i=0; i<height; i++)
    vResidualImageOut[i] =  (short *) calloc(width, sizeof(short));
}
    }

  if(!(GetFlag(CImage->p64Mode,P_DECODER)))
    {
      SetCCITT();
      if (CurrentFrame>LastFrame)
{
  WHEREAMI();
  printf("Need positive number of frames.\n");
  exit(ERROR_BOUNDS);
}
      NumberFrames = LastFrame-CurrentFrame+1;
      p64EncodeSequence();
    }
  else
    {
      p64DecodeSequence();
    }

  exit(ErrorValue);
```

```
}

/*BFUNC

ExecuteQuantization() is used to find the appropriate quantization for
the current sequence.  It calls the oracle if the flag is set.

EFUNC*/

static void ExecuteQuantization()
{
  BEGIN("ExecuteQuantization");
  int CurrentSize;

  CurrentSize=BufferContents();
  GQuant = (CurrentSize/QDFact) + QOffs;
  MQuant=GQuant;
  if (Verbose)
    {
      printf("BufferContents: %d  New Q1: %d\n",
     CurrentSize,GQuant);
    }
  if (GQuant<1)  GQuant=1;
  if (GQuant>31) GQuant=31;
  if (MQuant<1)  MQuant=1;
  if (MQuant>31) MQuant=31;
}

/*BFUNC

p64EncodeSequence() encodes the sequence defined by the CImage and
CFrame structures.

EFUNC*/

void p64EncodeSequence()
{
  int i;
  BEGIN("p64EncodeSequence");

  MakeStat();
  MakeRate();
  MakeIob(READ_IOB);
  InitFS(CFS);
  ClearFS(CFS);
  InitFS(OFS);
  ClearFS(OFS);
  swopen(CImage->StreamFileName);

  if (Loud > MUTE)
```

```
    {
      PrintImage();
      PrintFrame();
    }
  if (FileSizeBits)  /* Rate is determined by bits/second. */
            Rate=(FileSizeBits*FrameRate)/(FrameSkip*(LastFrame-Current-
Frame+1));
  if (Rate)
    {
      QDFact = (Rate/120);
      QOffs = 1;
      if (!InitialQuant)
{
  InitialQuant = 10000000/Rate;
  if (InitialQuant>31) InitialQuant=31;
  else if (InitialQuant<1) InitialQuant=1;
  printf("Rate: %d   QDFact: %d  QOffs: %d\n",
 Rate,QDFact,QOffs);
  printf("Starting Quantization: %d\n",InitialQuant);
}
    }
  if (!InitialQuant)  InitialQuant=DEFAULT_QUANTIZATION;
  GQuant=MQuant=MaxQuant;

  for (i=0; i<NumberGOB; i++) /* New Quantization Control Init */
    {
      SecQuant[i] = GQuant;
      SecBits[i] = (Rate * FrameSkip / FrameRate);
    }

  BufferOffset=0;
  TotalBits=0;
  NumberOvfl=0;
  FirstFrameBits=0;
  printf("START>SEQUENCE\n");
  MakeFileNames();
  VerifyFiles();
  TransmittedFrames=0;
  while(CurrentFrame <= LastFrame)
    {
      p64EncodeFrame();
    }
  /*limit file growth*/
  if (CurrentFrame>LastFrame+1) {CurrentFrame=LastFrame+1;}
  TemporalReference = CurrentFrame % 32;
  WritePictureHeader();
  swclose();
  if (qinfo)
    fclose(QF);
  printf("END>SEQUENCE\n");
```

```
  printf("Bits for first frame: %d   Number of buffer overflows: %d\n",
 FirstFrameBits,NumberOvfl);
}


/*BFUNC

p64EncodeFrame() encodes a single image frame.

EFUNC*/

void p64EncodeFrame()
{
  BEGIN("p64EncodeFrame");
  int x, i;

  printf("START>Frame: %d\n",CurrentFrame);
  ReadIob(CurrentFrame);
  InstallFS(0,CFS);
  if (CurrentFrame!=StartFrame)
    GlobalMC();
  TemporalReference = CurrentFrame % 32;
  WritePictureHeader();

  for(x=0;x<10;x++) /* Initialize Statistics */
    {
      MacroTypeFrequency[x]=0;
      YTypeFrequency[x]=0;
      UVTypeFrequency[x]=0;
    }
  MotionVectorBits=MacroAttributeBits=0;
  YCoefBits=UCoefBits=VCoefBits=EOBBits=0;
  QUse=QSum=0;
  NumberNZ=0;

  p64EncodeSec3();

  RCStore[CurrentFrame].position=TotalBits;
  RCStore[CurrentFrame].baseq = GQuant;
  x = mwtell();
  LastBits = x - TotalBits;
  TotalBits = x;
  printf("Total No of Bits: %8d  Bits for Frame: %8d\n",
 TotalBits,LastBits);
  if (Rate)
    {
      printf("Buffer Contents: %8d  out of: %8d\n",
     BufferContents(),
     BufferSize());
    }
```

```
  printf("MB Attribute Bits: %6d  MV Bits: %6d   EOB Bits: %6d\n",
 MacroAttributeBits,MotionVectorBits,EOBBits);
  printf("Y Bits: %7d  U Bits: %7d  V Bits: %7d  Total Bits: %7d\n",
 YCoefBits,UCoefBits,VCoefBits,(YCoefBits+UCoefBits+VCoefBits));
  printf("MV StepSize: %f  MV NumberNonZero: %f  MV NumberZero: %f\n",
 (double) ((double) QSum)/((double)(QUse)),
 (double) ((double) NumberNZ)/
 ((double)(NumberGOB*NumberMDU*6)),
 (double) ((double) (NumberGOB*NumberMDU*6*64)- NumberNZ)/
 ((double)(NumberGOB*NumberMDU*6)));
  RCStore[CurrentFrame].size = LastBits;
  printf("Code MType: ");
  for(x=0;x<10;x++) printf("%5d",x);
  printf("\n");
  printf("Macro Freq: ");
  for(x=0;x<10;x++) printf("%5d",MacroTypeFrequency[x]);
  printf("\n");
  printf("Y     Freq: ");
  for(x=0;x<10;x++) printf("%5d",YTypeFrequency[x]);
  printf("\n");
  printf("UV    Freq: ");
  for(x=0;x<10;x++) printf("%5d",UVTypeFrequency[x]);
  printf("\n");
  SwapFS(CFS,OFS);
  if (Debug)
    {
      if ((Debug%10) == 1)
{
  sprintf(DecFileName,"%s_dec.%d", DebugFilePrefix, CurrentFrame);
  SaveMem(DecFileName, CFS->fs, 3);
}

      if (((Debug/10)%10) == 1)
{
  sprintf(Res1FileName, "%s_res1.%d", DebugFilePrefix, CurrentFrame);
  if ((RF1 = fopen(Res1FileName, "w")) == NULL)
    {
      printf("Can't open residual output file: %s", Res1FileName);
      Debug = ((Debug/100) * 100) + (Debug%10);
    }
  else
    WriteResidual();
  fflush(RF1);
  fclose(RF1);
}

      if (((Debug/100)%10) == 1)
{
  sprintf(Res2FileName, "%s_res2.%d", DebugFilePrefix, CurrentFrame);
  if ((RF2 = fopen(Res2FileName, "w")) == NULL)
```

```
    {
      printf("Can't open residual output file: %s", Res2FileName);
      Debug = ((Debug/10) * 10) + (Debug%10);
    }
  WriteResidualOut();
  fflush(RF2);
  fclose(RF2);
}
    }

  if (qinfo)
    {
      for(i=0; i<(height/16); i++)
if (fwrite(QuantP[i], sizeof(unsigned char), (width/176), QF)
    != width/176)
  {
    printf("Can't write quantization info to output file %s\n",
   QuantFileName);
    exit(2);
  }
    }

  Statistics();
  printf("END>Frame: %d\n",CurrentFrame);
  printf("GQuant = %d\n", GQuant);
  if (Rate)
    {
      if (CurrentFrame==StartFrame)
{/* Begin Buffer at 0.5 size */
  FirstFrameBits = TotalBits;

  printf("First Frame Reset Buffer by delta bits: %d\n",
 BufferOffset);
}
      /* Take off standard deduction afterwards. */
      BufferOffset -= (Rate*FrameSkip/FrameRate);
    }
  else if (CurrentFrame==StartFrame)
    FirstFrameBits = TotalBits;
  CurrentGOB=0;TransmittedFrames++;
  CurrentFrame+=FrameSkip;     /* Change GOB & Frame at same time */
  skipped = 0;                 /* Turn Rate Control back on       */
  while (BufferContents() > (Rate * FrameSkip/FrameRate))
    {
      printf("SKIP>Frame: %d\n", CurrentFrame);
      skipped = 1;             /* Turn Rate Control off           */
/*      if (GQuant < MaxQuant)
GQuant+=1; */
      CurrentFrame+=FrameSkip;  /* Drop Frame if Buffer Full      */
      BufferOffset -= (Rate*FrameSkip/FrameRate);
```

```
    }
}


/*BFUNC

p64EncodeSec() encodes one section (3 GOBs) within a frame.

EFUNC*/

void p64EncodeSec()
{
  BEGIN("p64EncodeSec");
  if ((CurrentGOB % 3) == 0)
    {
      MQuantEnable=0;                          /* Disable MQuant */
      if ((Rate) && (CurrentFrame!=StartFrame)) /* Change Quantization */
  ExecuteQuantization();
    }
  p64EncodeGOB2();
}


/* BFUNC

   p64EncodeSec2 encodes one section (3 GOBs) within a frame and deter-
mines
whether the quantization level should be changed for each section.

EFUNC */

void p64EncodeSec2()
{
  int Tbits, w, y, aver_sec_bit_alloc, CurrentSec;

  BEGIN("p64EncodeSec2");
  w = mwtell();
  aver_sec_bit_alloc = (Rate * FrameSkip / FrameRate) / (NumberGOB / 3);

  for (CurrentGOB=0; CurrentGOB<NumberGOB; CurrentMDU=0, CurrentGOB++)
    {
      if ((CurrentGOB % 3) == 0)
{
  CurrentSec = CurrentGOB / 3;
  if ((Rate) && (CurrentFrame!=StartFrame)) /* Change Quant */
     {
       if (SecBits[CurrentSec] > (1.2 * aver_sec_bit_alloc))
 {
   if (SecQuant[CurrentSec] < MaxQuant)
     SecQuant[CurrentSec] += 1;
```

```
 }
       else if (SecBits[CurrentSec] < (0.8 * aver_sec_bit_alloc))
    {
      if ((SecQuant[CurrentSec] > 2) && (skipped == 0))
        SecQuant[CurrentSec] -= 1;
    }
        GQuant = SecQuant[CurrentSec];
      }
    p64EncodeGOB2();
    y = mwtell();
    SecBits[CurrentSec] = y - w;
    w = y;
 }
      else { p64EncodeGOB2(); }
     }
   if ((CurrentFrame == StartFrame) && (GQuant > MaxQuant))
     GQuant = MaxQuant;
}


/* BFUNC

  p64EncodeSec3 encodes one section (3 GOBs) within a frame and deter-
mines
whether the quantization level should be changed for each section.  In
this
fcn, it looks at the bits encoded for the previous section (in current
frame)

EFUNC */

void p64EncodeSec3()
{
  int w, y, aver_sec_bit_alloc;

  BEGIN("p64EncodeSec3");
  w = mwtell();
  aver_sec_bit_alloc = (Rate * FrameSkip / FrameRate) / (NumberGOB / 3);

  for (CurrentGOB=0; CurrentGOB<NumberGOB; CurrentMDU=0, CurrentGOB++)
    {
      if ((CurrentGOB % 3) == 0)
{
  if ((Rate) && (CurrentFrame!=StartFrame)) /* Change Quant */
    {
      if (PrevSec_enc_bits > (1.2 * aver_sec_bit_alloc))
{
  if (GQuant < MaxQuant)
    GQuant += 1;
}
```

```
      else if (PrevSec_enc_bits < (0.8 * aver_sec_bit_alloc))
{
  if ((GQuant > 2) && (skipped == 0))
    GQuant -= 1;
}
    }
  p64EncodeGOB2();
  y = mwtell();
  PrevSec_enc_bits = y - w;
  w = y;
 }
      else { p64EncodeGOB2(); }
    }
  if ((CurrentFrame == StartFrame) && (GQuant > MaxQuant))
    GQuant = MaxQuant;
}


/*BFUNC

p64EncodeGOB2() encodes a group of blocks within a frame.

EFUNC*/

void p64EncodeGOB2()
{
  BEGIN("p64EncodeGOB2");
  int Val,OVal,i;

  MQuantEnable = 0;
  switch (ImageType)
    {
    case IT_NTSC:
    case IT_CIF:
      GRead=CurrentGOB;
      break;
    case IT_QCIF:
      GRead=(CurrentGOB<<1);
      break;
    default:
      WHEREAMI();
      printf("Unknown Image Type: %d\n",ImageType);
      break;
    }
  WriteGOBHeader();

  LastMBA = -1; MType=0;

  if (qinfo)
    {
```

```
      if (ImageType == IT_CIF)
for (i=0; i<3; i++)
  QuantP[3*(CurrentGOB/2)+i][CurrentGOB%2] = (unsigned char) GQuant;
      else
if (ImageType = IT_QCIF)
  for (i=0; i<3; i++)
    QuantP[3*CurrentGOB][1] = (unsigned char) GQuant;
    }

  for(;CurrentMDU<NumberMDU;CurrentMDU++)
    { /* MAIN LOOP */
      LastMType=MType;

      OVal = MeOVal[Bpos(CurrentGOB,CurrentMDU,0,0)];
      Val = MeVal[Bpos(CurrentGOB,CurrentMDU,0,0)];
      MVDH = MeX[Bpos(CurrentGOB,CurrentMDU,0,0)];
      MVDV = MeY[Bpos(CurrentGOB,CurrentMDU,0,0)];
      VAR  = MeVAR[Bpos(CurrentGOB,CurrentMDU,0,0)];
      VAROR = MeVAROR[Bpos(CurrentGOB,CurrentMDU,0,0)];
      MWOR = MeMWOR[Bpos(CurrentGOB,CurrentMDU,0,0)];
      ASAD = MeASAD[Bpos(CurrentGOB,CurrentMDU,0,0)];
      spike = MeSPK[Bpos(CurrentGOB,CurrentMDU,0,0)];
      orspike = MeOSPK[Bpos(CurrentGOB,CurrentMDU,0,0)];
      if (CurrentFrame!=StartFrame)   /* Intra vs. Inter decision */
{
  if (ASAD >= (Val - 500))
    {                               /* (MC+Inter)mode */
      if ((Val < not_coded_thresh) && (spike < MaxSpike))
  MType = 7;
      else
if (Val == OVal)
  MType = 2;                      /* Inter mode */
else                              /* MC mode */
  MType = 8;
    }
  else MType = 0; /*Intramode */
  if (MQuantEnable)
    {
      MType++;
    }
}
      else
MType = 0; /* We always start with Intramode */
      if (LastIntra[CurrentGOB][CurrentMDU]>SEQUENCE_INTRA_THRESHOLD)
MType=0;  /* Code intra every 132 blocks */

      /* printf("[State %d]",MType);*/
      if ((Rate)&&(BufferContents()>BufferSize()))
{
  MVDH=MVDV=0; /* Motion vectors 0 */
```

```
  MType=4;      /* No coefficient transmission */
  NumberOvfl++;
  WHEREAMI();
  printf("Buffer Overflow!\n");
}
      p64EncodeMDU();
    }
}


/*BFUNC

p64EncodeMDU(MDU,)
     ) encodes the MDU by read/compressing the MDU; encodes the MDU by
read/compressing the MDU, then
writing it, then decoding it and accumulating statistics.

EFUNC*/

static void p64EncodeMDU()
{
  BEGIN("p64EncodeMDU");

  ReadCompressMDU();
  WriteMDU();
  DecodeSaveMDU();

  QUse++;                  /* Accumulate statistics */
  QSum+=UseQuant;
  if (MType < 10)
    MacroTypeFrequency[MType]++;
  else
    {
      WHEREAMI();
      printf("Illegal MType: %d\n",MType);
    }
}

/*BFUNC

ReadCompressMDU(MDU,)
     ) reads in the MDU; reads in the MDU, and attempts to compress it.
If the chosen MType is invalid, it finds the closest match.

EFUNC*/

static void ReadCompressMDU()
{
  BEGIN("ReadCompressMDU");
  int c,j,h,v,x;
```

```
  int *input;
  int total,accum,pmask;

  while(1)                            /* READ AND COMPRESS */
    {
      if (QuantMType[MType])
{
  UseQuant=MQuant;
  GQuant=MQuant; /* Future MB Quant is now MQuant */
}
      else UseQuant=GQuant;
      for(c=0;c<6;c++)
{
  input = &inputbuf[c][0];
  j = BlockJ[c];
  v = BlockV[c];
  h = BlockH[c];
  if (TCoeffMType[MType])
    {
      InstallIob(j);
      MoveTo(CurrentGOB,CurrentMDU,h,v);
      ReadBlock(input);
      if (!IntraMType[MType])
{
  InstallFS(j,CFS);
  MoveTo(CurrentGOB,CurrentMDU,h,v);
  if (FilterMType[MType])
    {
      if (j)
HalfSubFCompensate(input);
      else
SubFCompensate(input);
    }
  else if (MFMType[MType])
    {
      if (j)
HalfSubCompensate(input);
      else
SubCompensate(input);
    }
  else
    SubOverlay(input);
}
      if (((Debug/10)%10) == 1)
StoreResidual(input, h, v, j);
      DefaultDct(input,output);
      BoundDctMatrix(output);
      if (IntraMType[MType])
{
  CCITTFlatQuantize(output,8,UseQuant);
```

```
  FlatBoundQuantizeMatrix(output);
}
      else
{
  CCITTQuantize(output,UseQuant,UseQuant);
  BoundQuantizeMatrix(output);
}
      ZigzagMatrix(output,input);
    }
  else
    for(x=0;x<64;x++) input[x] = 0;
}
      if (!CBPMType[MType]) CBP = 0x3f;  /* VERIFY MType CBP */
      else
{
  for(pmask=0,CBP=0,total=0,c=0;c<6;c++)
    {
      input = &inputbuf[c][0];
      for(accum=0,x=0;x<64;x++) accum += abs(input[x]);
      if ((accum)&&(pmask==0)) pmask|=bit_set_mask[5-c];
      if (accum>CBPThreshold) CBP |= bit_set_mask[5-c];
      total+= accum;
    }
  if (!CBP)
    {
      if (pmask) CBP=pmask;
      else
if (!MFMType[MType])
  {CBP=0;MType=3;continue;}
else
  {CBP=0;MType=7;continue;}
    }
}
      if (IntraMType[MType]) LastIntra[CurrentGOB][CurrentMDU]=0;
      else LastIntra[CurrentGOB][CurrentMDU]++;
      return;                             /* IF HERE, THEN EXIT LOOP */
    }                                    /* GOOD ENCODING TYPE */
}


/*BFUNC

StoreResidual() stores all residual components in a matrix that
will be written out to an image file.

EFUNC*/

static void StoreResidual(matrix, h, v, type)
     int *matrix, h, v, type;
{
```

```
  BEGIN("StoreResidual");
  int i,j, m, n, scale;
  scale = (type > 0) + 1;
  if (ImageType == IT_CIF)
    {
      m = (((((CurrentGOB/2) * 3) + (CurrentMDU / 11)) * 16) + (8 * v))/
scale;
      n = (((((CurrentGOB%2) * 11) + (CurrentMDU % 11)) * 16) + (8 * h))/
scale;
    }
  else if (ImageType == IT_QCIF)
    {
      m = ((((CurrentGOB * 3) + (CurrentMDU / 11)) * 16) + (8 * v))/scale;
      n = (((CurrentMDU % 11) * 16) + (8 * h))/scale;
    }

  for(i=0;i<BLOCKHEIGHT;i++)
    for(j=0;j<BLOCKWIDTH;j++)
      {
if (type == 0)
  yResidualImage[m + i][n + j] = (short) *matrix++;
else if (type == 1)
  uResidualImage[m + i][n + j] = (short) *matrix++;
else if (type == 2)
  vResidualImage[m + i][n + j] = (short) *matrix++;
      }
}

/*BFUNC

StoreResidualOut() stores all residual components in a matrix that
will be written out to an image file.

EFUNC*/

static void StoreResidualOut(matrix, h, v, type)
     int *matrix, h, v, type;
{
  BEGIN("StoreResidualOut");
  int i,j, m, n, scale;
  scale = (type > 0) + 1;
  if (ImageType == IT_CIF)
    {
      m = (((((CurrentGOB/2) * 3) + (CurrentMDU / 11)) * 16) + (8 * v))/
scale;
      n = (((((CurrentGOB%2) * 11) + (CurrentMDU % 11)) * 16) + (8 * h))/
scale;
    }
  else if (ImageType == IT_QCIF)
    {
```

```
      m = ((((CurrentGOB * 3) + (CurrentMDU / 11)) * 16) + (8 * v))/scale;
       n = (((CurrentMDU % 11) * 16) + (8 * h))/scale;
     }

  for(i=0;i<BLOCKHEIGHT;i++)
    for(j=0;j<BLOCKWIDTH;j++)
      {
if (type == 0)
  yResidualImageOut[m + i][n + j] = (short) *matrix++;
else if (type == 1)
  uResidualImageOut[m + i][n + j] = (short) *matrix++;
else if (type == 2)
  vResidualImageOut[m + i][n + j] = (short) *matrix++;
      }
}


/*BFUNC

WriteResidual() outputs the components of the residual image to
an image file.

EFUNC*/

static void WriteResidual()
{
  BEGIN("WriteResidual");
  int i;
  for(i=0; i<height; i++)
fwrite(yResidualImage[i], sizeof(short), width, RF1);
  for(i=0; i<(height/2); i++)
fwrite(uResidualImage[i], sizeof(short), (width/2), RF1);
  for(i=0; i<(height/2); i++)
fwrite(vResidualImage[i], sizeof(short), (width/2), RF1);
}

/*BFUNC

WriteResidualOut() outputs the components of the residual image to
an image file.

EFUNC*/

static void WriteResidualOut()
{
  BEGIN("WriteResidualOut");
  int i;
  for(i=0; i<height; i++)
fwrite(yResidualImageOut[i], sizeof(short), width, RF2);
  for(i=0; i<(height/2); i++)
```

```
fwrite(uResidualImageOut[i], sizeof(short), (width/2), RF2);
  for(i=0; i<(height/2); i++)
fwrite(vResidualImageOut[i], sizeof(short), (width/2), RF2);
}


/*BFUNC

WriteMDU() writes out the MDU to the stream. The input buffer and
MType must already be set once this function is called.

EFUNC*/

static void WriteMDU()
{
  BEGIN("WriteMDU");
  int c,j,x;
  int *input;

  MBA = (CurrentMDU-LastMBA);      /* WRITE */
  WriteMBHeader();
  LastMBA = CurrentMDU;
  for(c=0;c<6;c++)
    {
      j = BlockJ[c];
      input = &inputbuf[c][0];
      if ((CBP & bit_set_mask[5-c])&&(TCoeffMType[MType]))
{
  if(j) {UVTypeFrequency[MType]++;}
  else {YTypeFrequency[MType]++;}
  CodedBlockBits=0;
  if (CBPMType[MType])
    {
      CBPEncodeAC(0,input);
    }
  else
    {
      EncodeDC(*input);
      EncodeAC(1,input);
    }
  if(!j){YCoefBits+=CodedBlockBits;}
  else if(j==1){UCoefBits+=CodedBlockBits;}
  else{VCoefBits+=CodedBlockBits;}
  IZigzagMatrix(input,output);
  if (IntraMType[MType])
    ICCITTFlatQuantize(output,8,UseQuant);
  else
    ICCITTQuantize(output,UseQuant,UseQuant);
  DefaultIDct(output,input);
 }
```

```
      else for(x=0;x<64;x++) input[x]=0;
    }
}

/*BFUNC

DecodeSaveMDU() does a decode on the MDU that was just encoded/decoded
and left on the inputbuf array.  The device is OFS if encoding mode is
on, else it is the Iob if decoding mode is on.

EFUNC*/

static void DecodeSaveMDU()
{
  BEGIN("DecodeSaveMDU");
  int c,j,h,v;
  int *input;

  for(c=0;c<6;c++)
    {
      j = BlockJ[c];
      v = BlockV[c];
      h = BlockH[c];
      input = &inputbuf[c][0];

      if (((Debug/100)%10) == 1)
  StoreResidualOut(input, h, v, j);

      if (!IntraMType[MType])     /* DECODE */
{
  InstallFS(j,CFS);
  MoveTo(CurrentGOB,CurrentMDU,h,v);
  if (FilterMType[MType])
    {
      if (j)
HalfAddFCompensate(input);
      else
AddFCompensate(input);
    }
  else if (MFMType[MType])
    {
      if (j)
HalfAddCompensate(input);
      else
AddCompensate(input);
    }
  else
    AddOverlay(input);
}
      BoundIDctMatrix(input);       /* SAVE */
```

```
      if (!(GetFlag(CImage->p64Mode,P_DECODER)))
InstallFS(j,OFS);
      else
InstallIob(j);
      MoveTo(CurrentGOB,CurrentMDU,h,v);
      WriteBlock(input);
    }
}

/*BFUNC

p64DecodeSequence() decodes the sequence defined in the CImage and
CFrame structures.

EFUNC*/

void p64DecodeSequence()
{
  BEGIN("p64DecodeSequence");
  int SelfParity;
  int Active;
  int EndFrame=0;

  sropen(CImage->StreamFileName);
  if (ReadHeaderHeader()) /* nonzero on error or eof */
    {
      srclose();
      exit(ErrorValue);
    }
  MakeFileNames();
  Active=0;
  while(1)
    {
      if (!EndFrame)
ReadHeaderTrailer();
      if ((GRead < 0)||(EndFrame))  /* End Of Frame */
{
  if (!EndFrame)
    ReadPictureHeader();
  else
    TemporalReference++;
  if (Active)
    {
      CopyIob2FS(CFS);
      while(((CurrentFrame+TemporalOffset)%32) !=
    TemporalReference)
{
  printf("END> Frame: %d\n",CurrentFrame);
  WriteIob();
  CurrentFrame++;
```

```
}
      /* Might still be "Filler Frame" sent at the end of file */
      if (ParityEnable)
{
  SelfParity = ParityFS(CFS);
  if (Parity != SelfParity)
    {
      printf("Bad Parity: Self: %x  Sent: %x\n",
     SelfParity,Parity);
    }
}
    }
  else
    {      /* First Frame */
      if (ForceCIF)
ImageType=IT_CIF;
      else
{
  if (PType&0x04)
    {
      if (PSpareEnable&&PSpare==0x8c) ImageType=IT_NTSC;
      else ImageType=IT_CIF;
    }
  else ImageType=IT_QCIF;
}
      SetCCITT();
      if (Loud > MUTE)
{
  PrintImage();
  PrintFrame();
}
      MakeIob(WRITE_IOB);
      InitFS(CFS);
      ClearFS(CFS);
      TemporalOffset=(TemporalReference-CurrentFrame)%32;
      Active=1;
    }
  if ((EndFrame)||(ReadHeaderHeader())) /* nonzero on error or eof */
    break; /* Could be end of file */
  printf("START>Frame: %d\n",CurrentFrame); /* Frame is for real */
  continue;
}
      EndFrame = p64DecodeGOB();                      /* Else decode the
GOB */
    }
  srclose();
}

/*BFUNC
```

```
p64DecodeGOB() decodes the GOB block of the current frame.

EFUNC*/

int p64DecodeGOB()
{
  BEGIN("p64DecodeGOB");

  ReadGOBHeader();              /* Read the group of blocks header  */
  switch(ImageType)
    {
    case IT_NTSC:
    case IT_CIF:
      CurrentGOB = GRead;
      break;
    case IT_QCIF:
      CurrentGOB = (GRead>>1);
      break;
    default:
      WHEREAMI();
      printf("Unknown Image Type: %d.\n",ImageType);
      break;
    }
  if (CurrentGOB > NumberGOB)
    {
      WHEREAMI();
      printf("Buffer Overflow: Current:%d  Number:%d\n",
     CurrentGOB, NumberGOB);
      return;
    }
  LastMBA = -1;                 /* Reset the MBA and the other predictors
*/
  LastMVDH = 0;
  LastMVDV = 0;
  while(ReadMBHeader()==0)
    {
      if (DecompressMDU()) return(1);
      DecodeSaveMDU();
    }
  return(0);
}

/*BFUNC

DecompressMDU() decompresses the current MDU of which the header has
already been read off of the stream.  It leaves the decoded result in
the inputbuf array.  It returns a 1 if an end of file has occurred.

EFUNC*/
```

```
static int DecompressMDU()
{
  BEGIN("DecompressMDU");
  int c,j,x;
  int *input;

  LastMBA = LastMBA + MBA;
  CurrentMDU = LastMBA;
  if (CurrentMDU >= NumberMDU)
    {
      if ((CurrentGOB == NumberGOB-1)&&(seof()))
return(1);

      WHEREAMI();
      printf("Apparent MDU out of range: %d > %d.\n",CurrentMDU,Number-
MDU);
      printf("CurrentGOB: %d LastMBA %d, MBA: %d\n",
     CurrentGOB,LastMBA,MBA);
      printf("at bit position %d in stream\n",mrtell());
      CurrentMDU=0;
      LastMBA=0;
      return(0);
    }
  if (!CBPMType[MType]) CBP = 0x3f;
  if (QuantMType[MType])
    {
      UseQuant=MQuant;
      GQuant=MQuant;
    }
  else UseQuant=GQuant;
  for(c=0;c<6;c++)
    {
      j=BlockJ[c];
      input = &inputbuf[c][0];

      if ((CBP & bit_set_mask[5-c])&&(TCoeffMType[MType]))
{
  if (CBPMType[MType])
    CBPDecodeAC(0,input);
  else
    {
      *input = DecodeDC();
      DecodeAC(1,input);
    }
  if (Loud > TALK)
    {
      printf("Cooked Input\n");
      PrintMatrix(input);
    }
  IZigzagMatrix(input,output);
```

```
  if (IntraMType[MType])
    ICCITTFlatQuantize(output,8,UseQuant);
  else
    ICCITTQuantize(output,UseQuant,UseQuant);
  DefaultIDct(output,input);
}
      else  for(x=0;x<64;x++) input[x]=0;
    }
  return(0);
}

/*BFUNC

PrintImage() prints the image structure to stdout.

EFUNC*/

void PrintImage()
{
  BEGIN("PrintImage");

  printf("*** Image ID: %x ***\n",CImage);
  if (CImage)
    {
      if (CImage->StreamFileName)
{
  printf("StreamFileName %s\n",CImage->StreamFileName);
}
      printf("InternalMode: %d   Height: %d   Width: %d\n",
     CImage->p64Mode,CImage->Height,CImage->Width);
    }
}

/*BFUNC

PrintFrame() prints the frame structure to stdout.

EFUNC*/

void PrintFrame()
{
  BEGIN("PrintFrame");
  int i;

  printf("*** Frame ID: %x ***\n",CFrame);
  if (CFrame)
    {
      printf("NumberComponents %d\n",
     CFrame->NumberComponents);
      for(i=0;i<CFrame->NumberComponents;i++)
```

```
{
  printf("Component: FilePrefix: %s\n",
 ((*CFrame->ComponentFilePrefix) ?
  CFrame->ComponentFilePrefix : "Null"));
  printf("Height: %d  Width: %d\n",
 CFrame->Height[i],CFrame->Width[i]);
  printf("HorizontalFrequency: %d  VerticalFrequency: %d\n",
 CFrame->hf[i],CFrame->vf[i]);
  InstallIob(i);
  PrintIob();
}
    }
}

/*BFUNC

MakeImage() makes an image structure and installs it as the current
image.

EFUNC*/

void MakeImage()
{
  BEGIN("MakeImage");

  if (!(CImage = MakeStructure(IMAGE)))
    {
      WHEREAMI();
      printf("Cannot make an image structure.\n");
    }
  CImage->StreamFileName = NULL;
  CImage->p64Mode = 0;
  CImage->Height = 0;
  CImage->Width = 0;
}

/*BFUNC

MakeFrame() makes a frame structure and installs it as the current
frame structure.

EFUNC*/

void MakeFrame()
{
  BEGIN("MakeFrame");
  int i;

  if (!(CFrame = MakeStructure(FRAME)))
    {
```

```
      WHEREAMI();
      printf("Cannot make an frame structure.\n");
    }
  CFrame->NumberComponents = 3;
  *CFrame->ComponentFileName='\0';
  *CFrame->ComponentFilePrefix='\0';
  for(i=0;i<MAXIMUM_SOURCES;i++)
    {
      CFrame->Height[i] = 0;
      CFrame->Width[i] = 0;
      CFrame->hf[i] = 1;
      CFrame->vf[i] = 1;
    }
}

/*BFUNC

MakeFstore() makes and installs the frame stores for the motion
estimation and compensation.

EFUNC*/


void MakeFstore()
{
  int i;

  CFS = (FSTORE *) malloc(sizeof(FSTORE));
  CFS->NumberComponents = 0;
  for(i=0;i<MAXIMUM_SOURCES;i++)
    {
      CFS->fs[i] = NULL;
    }
  OFS = (FSTORE *) malloc(sizeof(FSTORE));
  OFS->NumberComponents = 0;
  for(i=0;i<MAXIMUM_SOURCES;i++)
    {
      OFS->fs[i] = NULL;
    }
}

/*BFUNC

MakeStat() makes the statistics structure to hold all of the current
statistics. (CStat and RStat).

EFUNC*/


void MakeStat()
```

```
{
  CStat = MakeStructure(STAT);
  RStat = MakeStructure(STAT);
}

/*BFUNC

MakeRate() makes some statistics and book-keeping structures for
advanced rate tracking through the frames.

EFUNC*/

void MakeRate()
{
  RCStore = (RATE *) calloc(NumberFrames,sizeof(RATE));
}

/*BFUNC

SetCCITT() sets the CImage and CFrame parameters for CCITT coding.

EFUNC*/

void SetCCITT()
{
  BEGIN("SetCCITT");
  int i;

  if (*CFrame->ComponentFilePrefix=='\0')
    {
      WHEREAMI();
      printf("A file prefix should be specified.\n");
      exit(ERROR_BOUNDS);
    }
  CFS->NumberComponents = 3;
  OFS->NumberComponents = 3;
  CFrame->NumberComponents = 3;
  CFrame->hf[0] = 2;
  CFrame->vf[0] = 2;
  CFrame->hf[1] = 1;
  CFrame->vf[1] = 1;
  CFrame->hf[2] = 1;
  CFrame->vf[2] = 1;
  switch(ImageType)
    {
    case IT_NTSC:
      NumberGOB = 10;  /* Parameters for NTSC design */
      NumberMDU = 33;
      CImage->Width = 352;
      CImage->Height = 240;
```

```
      CFrame->Width[0] = 352;
      CFrame->Height[0] = 240;
      CFrame->Width[1] = 176;
      CFrame->Height[1] = 120;
      CFrame->Width[2] = 176;
      CFrame->Height[2] = 120;
      break;
    case IT_CIF:
      NumberGOB = 12;  /* Parameters for NTSC design */
      NumberMDU = 33;
      CImage->Width = 352;
      CImage->Height = 288;
      CFrame->Width[0] = 352;
      CFrame->Height[0] = 288;
      CFrame->Width[1] = 176;
      CFrame->Height[1] = 144;
      CFrame->Width[2] = 176;
      CFrame->Height[2] = 144;
      break;
    case IT_QCIF:
      NumberGOB = 3;  /* Parameters for NTSC design */
      NumberMDU = 33;
      CImage->Width = 176;
      CImage->Height = 144;
      CFrame->Width[0] = 176;
      CFrame->Height[0] = 144;
      CFrame->Width[1] = 88;
      CFrame->Height[1] = 72;
      CFrame->Width[2] = 88;
      CFrame->Height[2] = 72;
      break;
    default:
      WHEREAMI();
      printf("Unknown ImageType: %d\n",ImageType);
      exit(ERROR_BOUNDS);
      break;
    }

  LastIntra = (unsigned char **) calloc(NumberGOB,sizeof(unsigned char
*));
  for(i=0;i<NumberGOB;i++)
    {
      /* Should be assigned to all zeroes */
      LastIntra[i] = (unsigned char *) calloc(NumberMDU,sizeof(unsigned
char));
      memset(LastIntra[i],0,NumberMDU);  /* just in case */
    }
}

/*BFUNC
```

```
Help() prints out help information about the p64 program.

EFUNC*/

void Help()
{
  BEGIN("Help");

  printf("p64  [-d [-c]] [-NTSC] [-CIF] [-QCIF]\n");
  printf("     [-a StartNumber] [-b EndNumber]\n");
  printf("     [-D Debug_Code Debug_File_Prefix]\n");
  printf("     [-f FrameRate] [-k FrameskipNumber] [-o] [-p]\n");
  printf("     [-i MCSearchLimit] [-q Quantization] [-v] [-y]\n");
  printf("     [-r Target Rate] [-x Target Filesize]\n");
  printf("     [-s StreamFile] \n");
  printf("     ComponentFilePrefix \n");
  printf("-NTSC (352x240)  -CIF (352x288) -QCIF (176x144) base file-
sizes.\n");
  printf("-a is the start filename index. [inclusive] Defaults to 0.\n");
  printf("-b is the end filename index. [inclusive] Defaults to 0.\n");
  printf("-c forces cif large-frame decoding (can be used with all input
modes).\n");
  printf("-d enables the decoder\n");
  printf("-D Debug Code (111 all three modes, 110 only residuals
etc).\n");
  printf("-f gives the frame rate (default 30).\n");
  printf("-g produces a file with each GOB's GQuant (prefix_dec.q).\n");
  printf("-i gives the MC search area: between 1 and 31 (default
15).\n");
  printf("-k is the frame skip index. Frames/s = FrameRate/
FrameSkip.\n");
  printf("-n Not_Coded_Thresh No_MotEst_Thresh MaxSpike.\n");
  printf("-o enables the interpreter.\n");
  printf("-p enables parity checking (disabled for 1992 CCITT
specs).\n");
  printf("-q denotes Quantization, between 1 and 31.\n");
  printf("-r gives the target rate in bps.\n");
  printf("-s denotes StreamFile, which defaults to ComponentFilePre-
fix.p64\n");
  printf("-t denotes ThreeStep Motion Estimation.\n");
  printf("-u MaxQuant sets the maximum level for the Q parameter.\n");
  printf("-v denotes verbose mode, showing quantization changes.\n");
  printf("-x gives the target filesize in kilobits. (overrides -r
option.)\n");
  printf("-y enables Reference DCT.\n");
}


/*BFUNC
```

```
MakeFileNames() creates the filenames for the component files
from the appropriate prefix and suffix commands.

EFUNC*/

void MakeFileNames()
{
  BEGIN("MakeFileNames");
  int i;

  if (ImageType == IT_CIF)
         sprintf(CFrame->ComponentFileName,"%s.cif",  CFrame->Component-
FilePrefix);
  else
        sprintf(CFrame->ComponentFileName,"%s.qcif",  CFrame->Component-
FilePrefix);
}


/*BFUNC

VerifyFiles() checks to see if the component files are present and
of the correct length.

EFUNC*/

void VerifyFiles()
{
  BEGIN("VerifyFiles");
  int i,FileSize;
  FILE *test;

  if ((test = fopen(CFrame->ComponentFileName,"r")) == NULL)
    {
      WHEREAMI();
      printf("Cannot Open FileName %s\n",
     CFrame->ComponentFileName);
      exit(ERROR_BOUNDS);
    }
  fseek(test,0,2);
  FileSize = ftell(test);
  rewind(test);
  if (CFrame->Height[0] == 0)
    {
      if (CFrame->Width[0] == 0)
{
  WHEREAMI();
  printf("Bad File Specification for file %s\n",
 CFrame->ComponentFileName);
```

```
}
      else
{
  CFrame->Height[0] = FileSize / CFrame->Width[0];
  printf("Autosizing Height to %d\n",
 CFrame->Height[0]);
}
    }
  if (FileSize < ((LastFrame - StartFrame + 1) / FrameSkip * 1.5 *
  CFrame->Width[0] * CFrame->Height[0]))
    {
      WHEREAMI();
      printf("Inaccurate File Sizes: Estimated %d: %s: %d \n",
     ((LastFrame - StartFrame + 1) / FrameSkip * 1.5 *
      CFrame->Width[0] * CFrame->Height[0]),
     CFrame->ComponentFileName, FileSize);
      exit(ERROR_BOUNDS);
    }
  fclose(test);
}

/*END*/
```

A.2 File "ME.c"

```
/***
  File has been changed by:
  Brian Davison
  Latest change date: 24 Aug 95

  Changes:
    1. Added MotionEstimation, ThreeStep, & OneAtATime functions
         at the end along with extern DEFs
    2. Removed BruteMotionEstimation & FastBME along with DEFs
    3. Added a thresholz variable to check whether motion estimation
         should be performed or not.
    4. Added another variable for each MB = 16x16 sum of
         |intensity - u_intensity|.

    5. Uses original picture to determine VAR, VAROR, MWOR, and
        ASAD.
    6. Reduces the 0-SAD by 100 before as in TMN5.
***/

/************************************************************
Copyright (C) 1990, 1991, 1993 Andy C. Hung, all rights reserved.
PUBLIC DOMAIN LICENSE: Stanford University Portable Video Research
Group. If you use this software, you agree to the following: This
program package is purely experimental, and is licensed "as is".
Permission is granted to use, modify, and distribute this program
without charge for any purpose, provided this license/ disclaimer
notice appears in the copies.  No warranty or maintenance is given,
either expressed or implied.  In no event shall the author(s) be
liable to you or a third party for any special, incidental,
```

```
consequential, or other damages, arising out of the use or inability
to use the program for any purpose (or the loss of data), even if we
have been advised of such possibilities.  Any public reference or
advertisement of this source code should refer to it as the Portable
Video Research Group (PVRG) code, and not by any author(s) (or
Stanford University) name.
*************************************************************/
/*
************************************************************
me1e.c

This file does much of the motion estimation and compensation.

************************************************************
*/

/*LABEL me_1d.c */

#include "globals.h"

/*PUBLIC*/

extern void initmc();
extern void BMC();
extern MEM *MotionCompensation();
extern void MotionEstimation();
extern void ThreeStep();
extern void OneAtATime();

/*PRIVATE*/

static int VAR;
static int VAROR;
static int MWOR;
static int ASAD;
static int MEAN;
static int spike;
static int orspike;

int MeOSPK[1024];
int MeSPK[1024];
int MeVAR[1024];
int MeVAROR[1024];
int MeMWOR[1024];
int MX;
int MY;
int MV;
int OMV;
int MeX[1024];
int MeY[1024];
int MeVal[1024];
int MeOVal[1024];
int MeASAD[1024];
int MeN=0;
```

```
int SearchLimit = 15;
extern int no_motest_thresh;
extern int not_coded_thresh;
extern int ME_ThreeStep;

BLOCK nb,rb;

#define COMPARISON >=  /* This is to compare for short-circuit exit */

/*START*/

/*BFUNC

initmc() initializes the block structures used for copying areas
of memory.

EFUNC*/

void initmc()
{
  BEGIN("initmc");

  nb = MakeBlock();
  rb = MakeBlock();
}

/*BFUNC

BMC() does a motion compensated copy from one memory structure to
another memory structure with appropriate indexes.

EFUNC*/

void BMC(rx,ry,rm,cx,cy,cm)
     int rx;
     int ry;
     MEM *rm;
     int cx;
     int cy;
     MEM *cm;
{
  BEGIN("BMC");

  SetPointerBlock(rx,ry,cm,nb);
  SetPointerBlock(cx+MX,cy+MY,rm,rb);
  CopyBlock(rb,nb);
}

/*BFUNC

MotionCompensation() does a full motion compensation of all the blocks
based on the motion vectors created by BruteMotionEstimation. Not
actually used in the program. If (omem) is null, it creates a new
```

```
memory structure.

EFUNC*/

MEM *MotionCompensation(mem,omem)
     MEM *mem;
     MEM *omem;
{
  BEGIN("MotionCompensation");
  int x,y;
  int count;
  MEM *temp;

  if (omem) {temp=omem;}
  else {temp = MakeMem(mem->width,mem->height);}
  for(count=0,y=0;y<mem->height;y+=16)
    {
      for(x=0;x<mem->width;x+=16)
{
  SetPointerBlock(x,y,temp,nb);
  SetPointerBlock(x+MeX[count],y+MeY[count],mem,rb);
  count++;
  CopyBlock(rb,nb);
}
    }
  return(temp);
}


/* BFUNC

   Motion Estimation performs one of two motion estimation algorithms
if the zero displacement (D0) SAD is above its threshold value.  The
default algorithm is the One-At-A-Time method with an alternative option
of a Three-Step search method.  These are performed in the fcns OneAtA-
Time
and ThreeStep.

EFUNC */

void MotionEstimation(prevmem, currmem)
     MEM *prevmem;
     MEM *currmem;
{
  BEGIN("MotionEstimation");
  register unsigned char *pp, *cp;
  int x, y, lMB;
  register int m, n, datum;
  int Avg_SAD = 0;

  for (MeN=0, y=0; y<currmem->height; y+=16)
    {
      for (x=0; x<currmem->width; x+=16)
{
```

```
  MV = spike = orspike = 0;
  pp = prevmem->data + x + (y * prevmem->width);
  cp = currmem->data + x + (y * currmem->width);
  VAR = VAROR = MWOR = 0;
  for (m=0; m<16; m++)
    {
      for (n=0; n<16; n++)
{
  datum = (*(pp) - *(cp));
  if (abs(datum) > orspike)
    orspike = abs(datum);
  VAR += datum * datum;
  MWOR += *(cp);
  if (datum<0)
    {MV -= datum;}
  else {MV += datum;}
  pp++;
  cp++;
}
      pp += (prevmem->width - 16);
      cp += (currmem->width - 16);
    }

  OMV = MV;
  if (MeN==0)
    {lMB = 0;}
  else {lMB = MeN - 1;}
  if (OMV > no_motest_thresh)
    {
      if (ME_ThreeStep)
{
  ThreeStep(x, y, prevmem, x, y, currmem);
}
      else
{
  OneAtATime(x, y, prevmem, x, y, currmem, \
     MeX[lMB], MeY[lMB]);
}
    }
  else
    {
      MEAN = MWOR/256;
      cp = currmem->data + x + (y * currmem->width);
      for (ASAD=0, m=0; m<16; m++)
{
  for (n=0; n<16; n++)
    {
      VAROR += *(cp) * (*(cp));
      datum = *(cp++) - MEAN;
      if (datum<0)
{ASAD-=datum;}
      else {ASAD+=datum;}
    }
  cp += (currmem->width - 16);
```

```
}
      MX = MY = 0;
      VAR = VAR/256;
      VAROR = (VAROR/256) - (MWOR/256) * (MWOR/256);
      spike = orspike;
    }

  OMV -= 100;
  if (OMV <= MV)
    {
      MV = OMV;
      spike = orspike;
      MX = MY = 0;
    }

  MeSPK[MeN] = spike;
  MeOSPK[MeN] = orspike;
  MeVAR[MeN] = VAR;
  MeVAROR[MeN] = VAROR;
  MeMWOR[MeN] = MWOR;
  MeX[MeN] = MX;
  MeY[MeN] = MY;
  MeVal[MeN] = MV;
  MeOVal[MeN] = OMV;
  MeASAD[MeN] = ASAD;
  MeN++;
}
    }
}


/* BFUNC

   The ThreeStep algorithm searches within a +/- 7 pixel search range
using
three exponential steps.  The function first compares the zero-displace-
ment
SAD with all combinations of +/- 4 pixel displacements and uses the
MV with the least error as its new origin.  Then, it repeats the process
using a +/- 2 pixel displacement, and finally a +/- 1 pixel displacement.
This reduces the number of dispacement positions that must be used to
calculate errors from 241 to 25 (an order of magnitude decrease).  The
function calls FastTSE for each of its MBs and then stores the stats
for each in its variables' arrays.

EFUNC*/

void ThreeStep(rx, ry, rm, cx, cy, cm)
     int rx;
     int ry;
     MEM *rm;
     int cx;
     int cy;
     MEM *cm;
```

```
{
  BEGIN("ThreeStep");
  int px, py, dx, dy, step;
  register int i, j, data, val;
  register unsigned char *bptr, *cptr;
  unsigned char *baseptr;

  MX=MY=0;
  bptr=rm->data + rx + (ry * rm->width);
  baseptr=cm->data + cx + (cy * cm->width);
  cptr=baseptr;

  px=rx;
  py=ry;

  for (step=4; step>=1; step/=2)
    {
      for(dx=-1; dx<=1; dx++)
if ( ((px+dx*step) >= 0) && ((px+dx*step) <= rm->width-16) )
  for(dy=-1; dy<=1; dy++)
    if ( !((dx==0) && (dy==0)) && ((py+dy*step) >= 0) && ((py+dy*step) <=
rm->height-16))
      {
val=0;
bptr = rm->data + (px + dx * step) + ((py + dy*step) * rm->width);
cptr=baseptr;
for (i=0;i<16;i++)
  {
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
```

```
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      if (val COMPARISON MV) break;
      bptr += (rm->width - 16);
      cptr += (cm->width - 16);
    }
  if (val < MV)
    {
      MV = val;
      MX = px + (dx*step) - rx;
      MY = py + (dy*step) - ry;
    }
        }
        px = rx + MX;
        py = ry + MY;
      }
    bptr = rm->data + (rx + MX) + ((ry + MY) * rm->width);
    cptr = baseptr;
    for (VAR=0,VAROR=0,MWOR=0,i=0; i<16; i++)
      {
        for (j=0; j<16; j++)
  {
    data = *(bptr) - *(cptr);
    if (abs(data) > spike)
      spike = abs(data);
    VAR += data*data;
    VAROR += *(cptr) * (*(cptr));
    MWOR += *(cptr);
    bptr++;
    cptr++;
  }
        bptr += (rm->width - 16);
        cptr += (cm->width - 16);
      }
    MEAN = MWOR/256;
    cptr = baseptr;
    bptr = rm->data + (rx + MX) + ((ry + MY) * rm->width);
    for (ASAD=0,i=0; i<16; i++)
      {
        for (j=0; j<16; j++)
  {
    data = *(cptr++) - MEAN;
    if (data<0)
      {ASAD-=data;}
    else {ASAD+=data;}
  }
        cptr += (cm->width - 16);
      }
    VAR = VAR/256;
    VAROR = (VAROR/256) - (MWOR/256)*(MWOR/256);
```

```
}


/* BFUNC

   The default algorithm, the OneAtATime method, compares the D0 SAD
against the SAD computed using the MV of the MB to the left as a base.
Using the one with the least error as a new origin, the method then
computes the SAD using each adjacent (horizontal and vertical only on
first run) pixel as a base.  If it finds a lower SAD, it moves its origin
and begins to check its adjacent pixel bases again.  If not, it checks
its
diagonally adjacent pixel bases, and moves if it finds a lower SAD.  The
algorithm will continue to move until it reaches an equilibrium point or
reaches a threshold for the number of searches it can perform.

EFUNC */

void OneAtATime(rx, ry, rm, cx, cy, cm, lMX, lMY)
     int rx;
     int ry;
     MEM *rm;
     int cx;
     int cy;
     MEM *cm;
     int lMX, lMY;
{
  BEGIN("OneAtATime");
  int lMV, newMV, moves, x, y, px, py;
  register int i, j, data, val;
  register unsigned char *bptr, *cptr;
  unsigned char *baseptr;

/* Check whether MV of the GOB to the left provides a better
   origin than the D0 origin */

  MX=MY=lMV=0;
  baseptr=cm->data + cx + (cy * cm->width);
  cptr=baseptr;
  if (rx>0)
    {
      bptr=rm->data + (rx + lMX) + ((ry + lMY) * rm->width);

      for (i=0; i<16; i++)
{
  for (j=0; j<16; j++)
    {
      data = (*(bptr++) - *(cptr++));
      if (data<0)
{lMV -= data;}
      else lMV += data;
    }
  bptr += (rm ->width - 16);
  cptr += (cm ->width - 16);
```

```
}

      if (lMV < MV)
{
  px = rx + lMX;
  py = ry + lMY;
  MV = lMV;
}
      else
{
  px = rx;
  py = ry;
}
    }
  else
    {
      px = rx;
      py = ry;
    }

/* While moves are less than a threhold, continue comparing the SAD
of surrounding pixel origins */

  newMV = MV;
  moves = 0;

  while (moves < SearchLimit)
    {
      for (x=-1; x<=1; x+=2)
{
  bptr = rm->data + (px + x) + (py * rm->width);
  cptr = baseptr;
  for (val=0, i=0; i<16; i++)
    {
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
```

```
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        if (val >= newMV) break;
        bptr += (rm->width - 16);
        cptr += (cm->width - 16);
      }
    if (val < newMV)
      {
        newMV = val;
        MX = px + x - rx;
        MY = py - ry;
      }
}
        for (y=-1; y<=1; y+=2)
{
    bptr = rm->data + px + ((py + y) * rm->width);
    cptr = baseptr;
    for (val=0, i=0; i<16; i++)
      {
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
        if (data<0) {val-=data;} else {val+=data;}
        data=(*(bptr++)-*(cptr++));
```

```
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      data=(*(bptr++)-*(cptr++));
      if (data<0) {val-=data;} else {val+=data;}
      if (val >= newMV) break;
      bptr += (rm->width - 16);
      cptr += (cm->width - 16);
    }
  if (val < newMV)
    {
      newMV = val;
      MX = px - rx;
      MY = py + y - ry;
    }
}

/* If the horiz/vert pixel origins did not provide a lower SAD, then
   check the diagonal pixel origins */

      if (newMV == MV)
{
  for (x=-1; x<=1; x+=2)
    for (y=-1; y<=1; y+=2)
      {
bptr = rm->data + (px + x) + ((py + y) * rm->width);
cptr = baseptr;
for (val=0, i=0; i<16; i++)
  {
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
```

```
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    data=(*(bptr++)-*(cptr++));
    if (data<0) {val-=data;} else {val+=data;}
    if (val >= newMV) break;
    bptr += (rm->width - 16);
    cptr += (cm->width - 16);
  }
if (val < newMV)
  {
    newMV = val;
    MX = px + x - rx;
    MY = py + y - ry;
  }
      }
}

/* Set new origin & MV, increment # of moves, & continue checking
   only if the new SAD is lower than that of the previous origin */

      if (newMV < MV)
{
  px = rx + MX;
  py = ry + MY;
  MV = newMV;
  moves++;
}
      else {moves = SearchLimit;}
    }
  bptr = rm->data + (rx+MX) + ((ry+MY) * rm->width);
  cptr = baseptr;
  for (VAR=0, VAROR=0, MWOR=0, i=0; i<16; i++)
    {
      for (j=0; j<16; j++)
{
  data = *(bptr) - *(cptr);
  VAR += data * data;
  VAROR += *(cptr) * (*(cptr));
  MWOR += *(cptr);
  bptr++;
  cptr++;
}
      bptr += (rm->width - 16);
      cptr += (cm->width - 16);
    }
  MEAN = MWOR/256;
  cptr = baseptr;
  bptr = rm->data + (rx + MX) + ((ry + MY) * rm->width);
  for (ASAD=0,i=0; i<16; i++)
```

```
    {
      for (j=0; j<16; j++)
{
  data = *(cptr++) - MEAN;
  if (data<0)
    {ASAD-=data;}
  else {ASAD+=data;}
}
      cptr += (cm->width - 16);
    }
  VAR = VAR/256;
  VAROR = (VAROR/256) - (MWOR/256) *  (MWOR/256);
}

/*END*/
```