SOFTWARE IMPLEMENTATION OF ASYNCHRONOUS VIDEO CODING FOR WIRELESS COMMUNICATION

by

Junjing Yan

Memorandum No. UCB/ERL M96/6

12 February 1996

SOFTWARE IMPLEMENTATION OF ASYNCHRONOUS VIDEO CODING FOR WIRELESS COMMUNICATION

by

Junjing Yan

Memorandum No. UCB/ERL M96/6

12 February 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Software Implementation of Asynchronous Video Coding for Wireless Communication

Abstract

Wireless video transport is becoming a popular topic in current multimedia networking. Video services such as video conferencing, video broadcasting and video-on-demand in an wireless environment challenge the existing video compression standard designed for wired, circuit-switch services.

Asynchronous Video Coding (ASV) was proposed as an efficient video compression strategy for applications in wireless communication.[6] ASV uses joint source channel coding in conjunction with variable-QoS[16], using a multiple-substream abstraction for the transport. ASV tries to simultaneously achieve high spectral efficiency, good subjective quality, and low perceptual delay. These features are especially important for a wireless channel. By identifying packets to the transport with relaxing reliability and delay requirements, the wireless transport can hopefully achieve higher channel capacity than existing video coding techniques.

We describe our software implementation of ASV algorithms in this report. Techniques involved include substream abstraction, vector quantization, fuzzy control, and multiple codebooks design. The implementation strategy is described in full detail. Simulation results are presented and compared. The direction of future ASV development is also proposed.

The implementation is written in C programming language. Simulation on a SunSparc 20 workstation gives an encouraging result.

Acknowledgments

I would like to acknowledge those who have helped make my path a challenging one and those who have supported me with their time, effort, and encouragement.

First and foremost, I would like to thank my advisor Professor David G. Messerschmitt for his guidance, encouragement and support. His continues help both from a technical and personal standpoint has help me tremendously throughout my stay in Berkeley. His insight in communication systems and his comprehensive knowledge of signal processing have encouraged me to work in this area and ultimately resulted in this study. His commitment to find solutions to problems as well as his technical experience have made my work interesting and challenging.

I would also like to thank the other member of my thesis committee, Professor Robert W. Brodersen. His support and advice throughout this study has been extremely valuable.

Thank Mr. Jeff Gilbert for helping me getting start with InfoPad video codec software implementation. Thank Mr. Johnathan Reason for helping me getting start with ASV technology. I owe special gratitude to Mr. Yuan-Chi Chang who made a great help in this study. I would like to thank him for his great mind and skills in the implementation of Fuzzy Logic Engine, Texture Estimation and Codebook designing as well as all the other help he offered me in this work. Thank Dr. Huen Joo Lee for contributing valuable knowledge in vector quantization and fuzzy logic theory. Thank Mr. Weiyi Li for proof reading my thesis as well as letting me learn a lot from him.

A special thank goes to the stuff at the Department of Electrical Engineering and Computer Science, Heather Levien, Mary Byrnes, Mary Stewart, Cindy Manly-Fields and Joyce McDougal for putting up with all my requests and for creating this pleasant atmosphere that makes it at home to study in the Department.

Last but by no means the least my family, who with their love, compassion and encouragement, helped me get through all these years. To them I dedicate this work.

Contents

List of Figures

List of Tables

Chapter One

Introduction

1.1 Overview of ASV

The performance of conventional video compression algorithms, which typically generate a single bit stream with high reliability requirements, is severely compromised by the high error-rate typical of an interference-dominated wireless access link. At the expense of increasing end-to-end delay and channel bandwidth, sophisticated error correction/data retransmission schemes are often used to achieve high reliability. In contrast to these conventional algorithms, asynchronous video coding allows the receiving end to reconstruct video asynchronously, thereby reducing an application's perceptual delay. This project is motivated by the needs of wireless video applications with low delay requirements, such as video conferencing, multimedia editing and interactive TV.

ASV coding scheme is described in figure 1.1. ASV coding scheme includes four major parts: Video Frame Partition, Fuzzy Control, Vector Quantization and Transmission

1) Frame Partition segments each video frame into small blocks;

2) Fuzzy Control takes each block's fuzzified values of texture estimation, motion estimation, as well as the feedback of rate monitor as its inputs. Based on user-defined fuzzification rules, it outputs the codebook-selector and substream-selector to specify the codebook and substream for the block;

3) Vector Quantization codes the block using a specific pre-designed codebook according to the codebook-selector. It can achieve compression ratio around 16:1;

4) Transmission packetises the codec data and transports the codec output along a

specific substream determined by substream-selector.

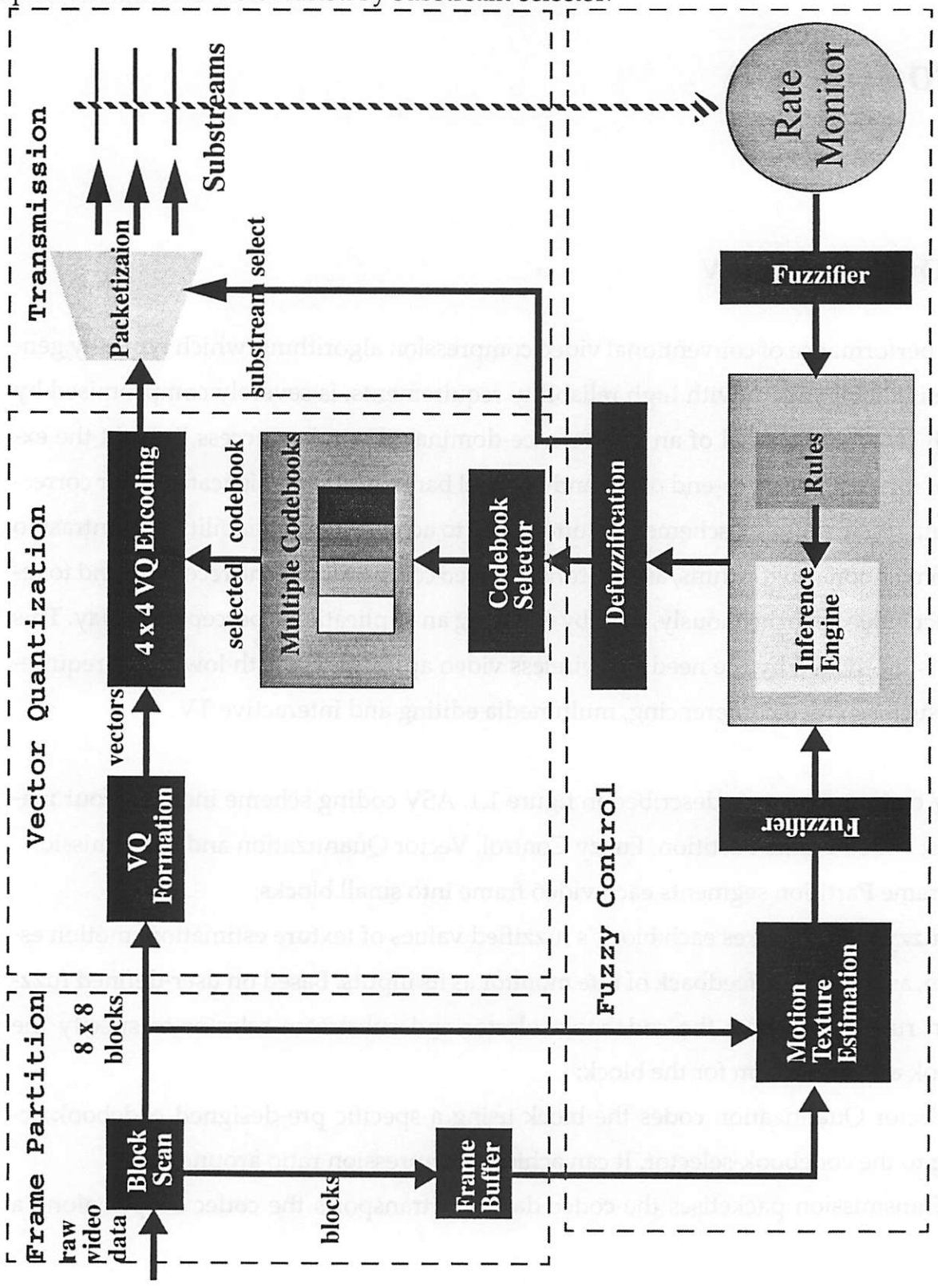

Figure 1.1 Overview of ASV Coding Scheme

Th asynchronous video coder segments its output data into multiple bit streams (called substreams) based on motion and texture estimates. These substreams are transported with different quality-of-service (QoS) requirements such as reliability and delay. Blocks of the video presentation in low-motion areas are transported with relaxed delay objectives, which can be exploited to achieve higher traffic capacity in the statistical multiplexing. Data in high motion areas is transported with released reliability requirements than data in low motion areas, taking advantage of the masking effect that the motion affords. Overall, this approach affords the traffic-increasing benefits of joint source-channel coding, while maintaining good system modularity through the substream transport abstraction.

The bit rate of each substream, as a part of the QoS contract with the network transport service, is used by the transport layer with other QoS parameters to allocate network resources. Rate control functions as a traffic-regulator in the video coder to make sure the generated traffic does not violate its rate contract. The advantage of building the rate controller within the video coder rather than leaving the traffic shaping to the transport is that the coder can adjust the bit rate of each substream with the objective of minimizing the subjective impairment, while the transport must indiscriminately drop data resulting in greater subjective impairment.

Using fuzzy descriptions for rate feedback and fuzzy descriptions for video characteristics provides a structured unified approach to build a knowledge-based coding system. The motion and texture contents of video, which can be perceived by human visual system but are difficult to quantify, are fuzzified and expressed as linguistic variables. They are jointly considered with rate feedback, based on the fuzzy rule base, to select quantization levels and outgoing substreams of video blocks.

This report presents the software implementation of such an ASV coding system.

1.2 Overview of thesis

In this thesis we describe the algorithm and implementation of a Asynchronous Video Application software in C programming language.

Chapter 2 is an introduction to the main techniques used in ASV. These techniques includes Substream Abstraction, Vector Quantization(VQ), Fuzzy Logic Control Engine. Chapter 3 introduces the detailed ASV coding and decoding schemes. We will discuss the implementation of spacial partition, motion and texture estimation, codebook design, fuzzy logic control and etc. Results of the simulation running on a SunSparc20 workstation are shown in Chapter 4. Possible future directions of ASV implementing are proposed in Chapter 5.

Chapter Two

Techniques Involved in ASV

2.1 Substream Abstraction

Substream partition is the key feature of ASV. Motivated by the scant bandwidth and high error rate of the wireless channel, joint source coding is desirable for the optimal wireless transportation. A multiple substream abstraction for source traffic with different delay, loss and reliability characteristics is proposed in ASV video application (See figure 2.1).

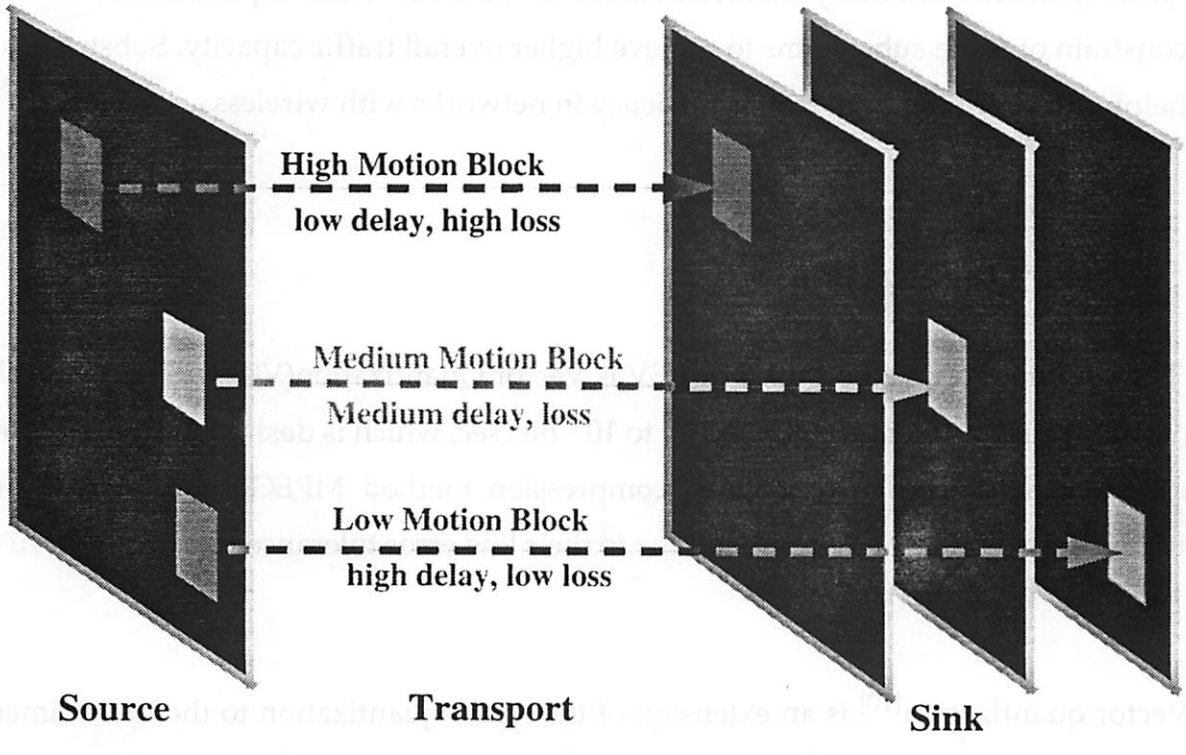

Figure 2.1 Joint video source coding with substreams and asynchronous reconstruction

For a particular video stream, the transport provisions substreams (through a link-layer protocol). Each substream has a quality of service (QoS) specification of loss, corruption, and delay characteristics. The QoS of the substreams are different, as negotiated at session establishment. Video data is segmented into multiple substreams based on motion and texture estimations. Blocks of the video with different motion and texture characteristics are transported through different substreams with various delay and reliability requirement. Video data arrives at the receiver asynchronously due to the different delay of substreams. We reconstruct the data asynchronously by updating the most recently received block and throwing away the stale data which violates ordering of arrival constrains.

In the absence of substreams, the delay characteristics of all the data in a continuous-media stream is the same. With substreams, on the other hand, data can be segregated into parts with different delay characteristics, and the network can exploit the relaxed delay constrain of some substreams to achieve higher overall traffic capacity. Substreams are helpful in obtaining high traffic efficiency in networks with wireless access links.

2.2 Vector Quantization

The compression method used in ASV is Vector Quantization(VQ). ASV has the objective of high error tolerance rate to 10^{-2} to 10^{-3} bit/sec, which is desired in a wireless access environment. The current video compression method MPEG1 and MPEG2 are unsuitable for a wireless environment due to their low error tolerance rate of 10^{-8} to 10^{-12} bit/sec.

Vector quantization[10] is an extension of the scalar quantization to the multidimensional form. It is a mapping of an input vector X, onto a representative vector Y in a codebook according to some distortion criterion, $d(X,Y)$. For images, X is generally a two-

dimensional(2-D) nonoverlapping block of pixels of dimension *K*. In ASV, VQ vector size is 4x4 pixels. The encoder generates the index *j*, such that

$$d(X, Y_j) < d(X, Y_i), \qquad i = 1, 2, ..., L, \quad i \neq j \qquad (1)$$

where Y_i is the *i*th codevector of the codebook of size *L*. The distortion measure used in ASV is the square error distortion given by

$$d(X, Y) = \|x - y\|^2 = \sum_{j=0}^{K-1} (X_j - Y_j)^2 \qquad (2)$$

Figure 2.2 shows the general VQ applied in image/video compression.

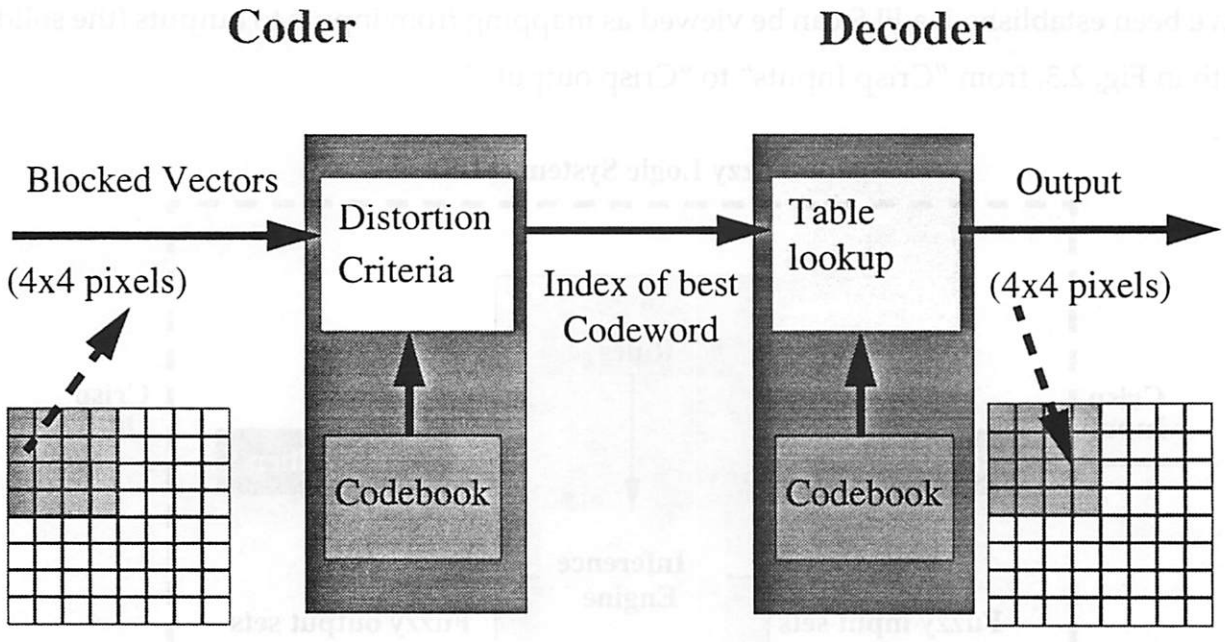

Figure 2.2 Vector Quantization for video compression

To make rate control feasible, multiple codebooks are designed in ASV to achieve scalable resolution level as well as bit rate of the transport. Codebooks of different sizes are

designed in this implementation. The detail of such codebook design will be covered in section 3.2.7. The compression ratio of vector quantization is about 16:1.

2.3 Fuzzy Logic Control System

A fuzzy logic system (FLS)[2] is unique in that it is able to simultaneously handle numerical data and linguistic knowledge. It is a nonlinear mapping of an input data (feature) vector into a scalar output, i.e., it maps numbers into numbers. Fuzzy set theory and fuzzy logic establish the specifics of the nonlinear mapping.

Figure 2.3 depicts a FLS that is widely used in fuzzy logic controllers, signal processing and communication applications. A FLS maps crisp inputs into crisp outputs. It contains four components: rules, fuzzifier, inference engine, and defuzzifier. Once the rules have been established, a FLS can be viewed as mapping from inputs to outputs (the solid path in Fig. 2.3, from "Crisp Inputs" to "Crisp outputs").

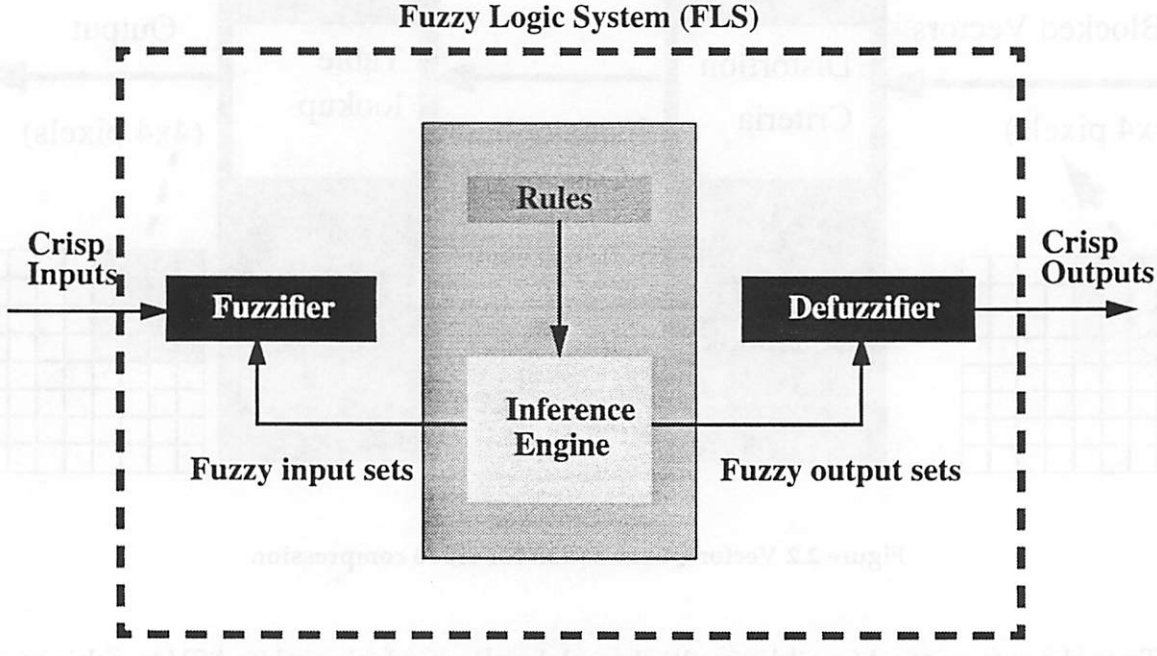

Figure 2.3 Fuzzy logic systems (FLS)

In ASV the rate control, substream abstraction and codebook selection are controlled by the fuzzy logic system. The crisp inputs of this FLS are the motion estimation, texture estimations of video blocks, as well as the transport bit rate on each substream. The crisp outputs are the substream selector and codebook selector. The inference engine fuzzy rules are user defined principles which define the mapping of the fuzzy input set and fuzzy output set. The detailed implementation of this fuzzy logic control engine will be described in section 3.2.5.

Chapter 3

ASV Coding

3.1 Overview of ASV Codec

The scheme works in the following way: First, we block scan each video frame raw data. Each video frame is partitioned into small blocks of 8x8 pixels. Then we apply motion estimation and texture estimation to these blocks. After the fuzzifier, the fuzzified results from the two estimations are fed into Fuzzy Control Engine along with the rate information from the rate monitor (which traces the traffic on each transport substream). The fuzzy control engine then outputs two control-signals, codebook selector and substream selector, to select the codebook and substream for each video block. Each step will be discussed in detail in the following sections.

For the convenience of the reader, the diagram of overview of ASV coding scheme is quoted again in this section in Figure 3.1. (Note this figure is the same as Figure 1.1).

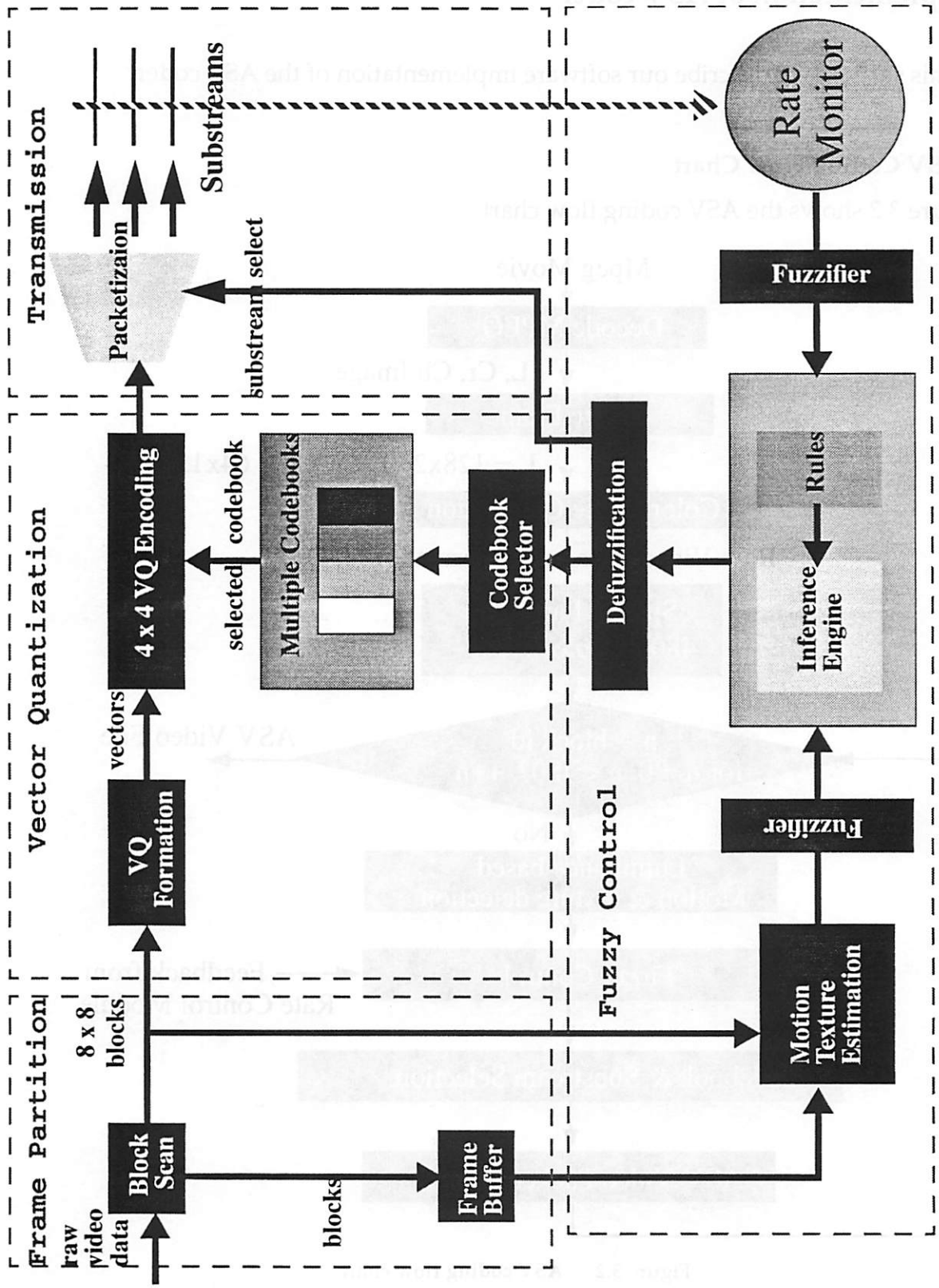

Figure 3.1 Overview of ASV Coding Scheme

3.2 Implementation of ASV coder

In this section, we describe our software implementation of the ASV coder.

3.2.1 ASV Coding Flow Chart

Figure 3.2 shows the ASV coding flow chart.

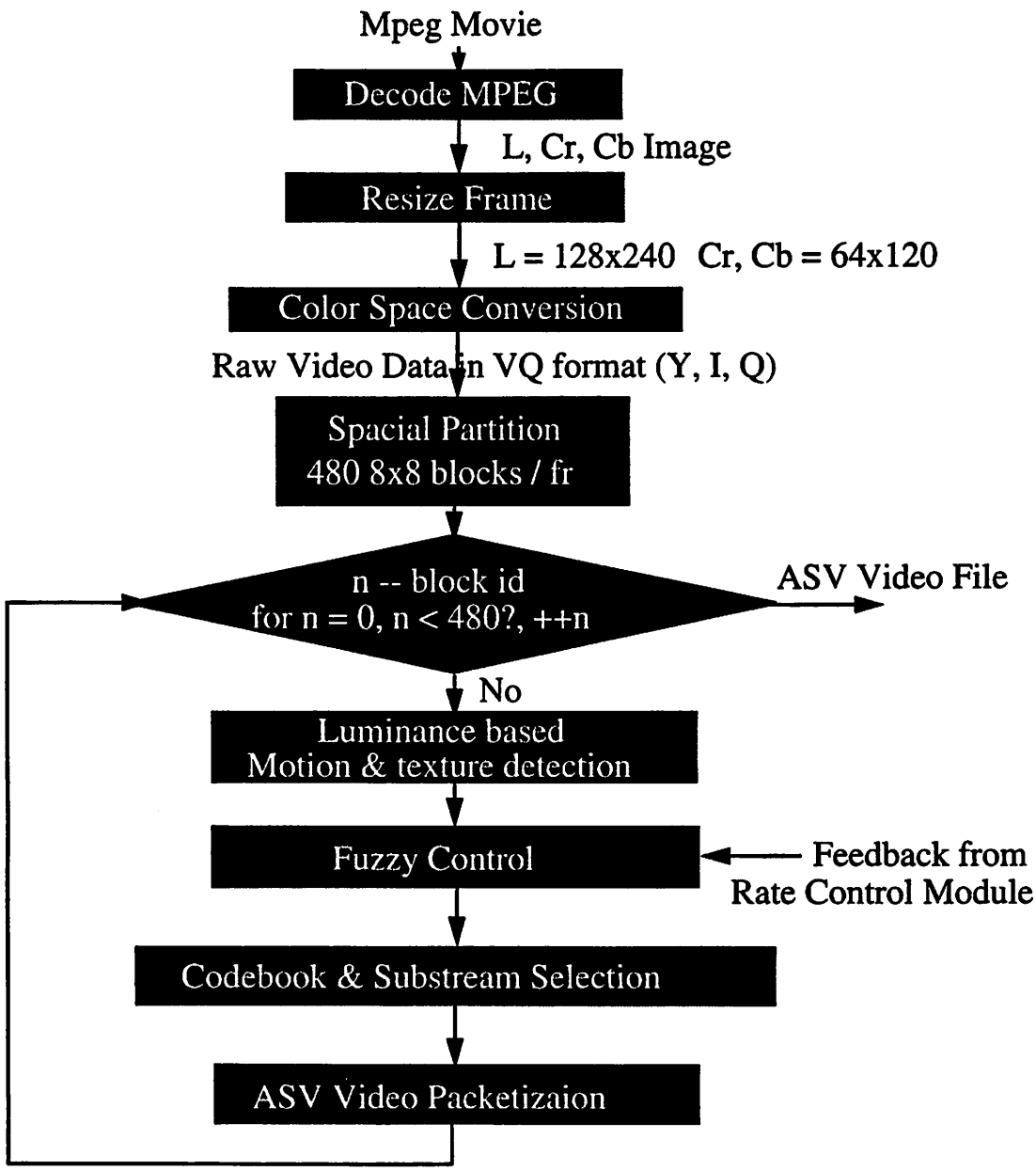

Figure 3.2 ASV coding flow chart

3.2.2 Mpeg to VQ Transcoding

Mpeg to VQ transcoding includes color space transformation and frame resizing.

Color space transformation

The source stores video in the MPEG format. Therefor, before we can apply ASV coding scheme, we need to transcode the video from MPEG format to VQ format.

There are many representations of color images. According to trichromatic theory, ideally, three arrays of samples should be sufficient to represent a color image[12]. RGB is one example of a color representation requiring three independent values to describe the colors. Each of the values can be varied independently, and we can therefor create a three-dimensional space with the three components, R, G, and B, as shown in Figure 3.3. Note that shades of gray from black to white are found on the diagonal line in this plot. In general pixels in a color image have information from the samples of each component, and the color image is compromised of the two-dimensional arrays of pixels.

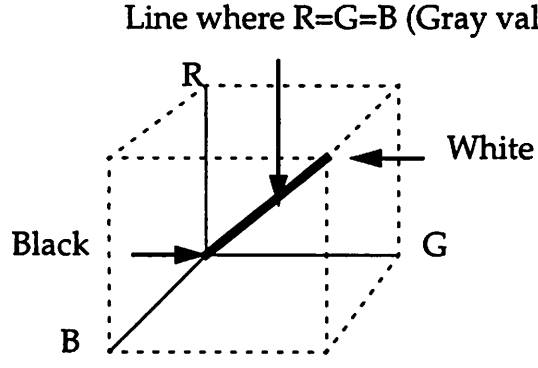

Figure 3.3 RGB (red-green-blue) color coordinate system.

Color representation such as RGB is not always the most convenient. A color space or color coordinate system in which one component is the luminance and the other two components are related to hue and saturation is called luminance-chrominance representations. The luminance provides a grayscale version of the image, and the

chrominance components provide the extra information that converts the grayscale image to a color image. Luminance-chrominance representations are particularly important for good image compression. Both MPEG and VQ use this luminance-chrominance representation except for different coefficients.

In MPEG, the color space representation is LCrCb. LCrCb was used extensively in the development of the JPEG and MPEG standard. It defines:

$$R = L + 1.402 \times (C_b - 128)$$
$$G = L - 0.344 \times (C_r - 128) - 0.174 \times (C_b - 128)$$
$$B = L + 1.772 \times (C_r - 128)$$

Vector Quantization uses a different color space known as YI Q. YIQ is used in the North American television systems. It defines (here using Info-pad as an example):

$$R = Y + 1.188 \times (I - 128) + 0.719 \times (Q - 128)$$
$$G = Y - 0.328 \times (I - 128) - 0.750 \times (Q - 128)$$
$$B = Y - 1.375 \times (I - 128) + 1.969 \times (Q - 128)$$

Thus, overall we have

$$Y = L$$
$$I = -0.384 \times C_r + 0.831 \times C_b + 70.784$$
$$Q = 0.630 \times C_r + 0.583 \times C_b - 27.264$$

as the color space transformation.

Frame Resizing

The frame size of MPEG video varies. In Infopad, the display terminal can only accept video fixed to an image size of 128x240. Consequently, we need to resize the source video frame before applying ASV coding.

The C program function "rescale_image()" is called to implement this image resizing.

```
================================================================
/*
 * rescale_imageis
 * Called to rescale an image from one size to another
 * using precomputed image rescale aux buffers
 * Results:
 *      Destination image is computed.
 * Side effects:
 *      None.
 *----------------------------------------------------
 */
void rescale_image(src_image, dst_image, ri, src_x_offset,
src_y_offset, dst_x_offset, dst_y_offset)
     unsigned char   *src_image;
     unsigned char   *dst_image;
     RescaleInfo     *ri;
     int             src_x_offset, src_y_offset;
     int             dst_x_offset, dst_y_offset;
{}
================================================================
```

For data structure of "RescaleInfo", see appendix A.

After transcoding, the raw video data of each frame in YIQ color space is stored in three buffers which store Y, I, Q respectively:

```
static u_char *L_Y_scaled_buf;
static u_char *Cr_I_scaled_buf;
static u_char *Cb_Q_scaled_buf;
```

In general, the chrominance data are not as important as the luminance data. Thus, the resolution of I and Q is half of Y, which means that the buffer size of I and Q are one quarter of the size of the Y buffer.

3.2.3 Spacial Partition

One of the main procedures in ASV coding is the spacial partition. After transcoding from MPEG, each frame is partitioned into small 8x8 pixels blocks on which motion and texture estimation will be applied. These blocks are called a "Video Block". They are described by data structure as follows:

```
================================================================
struct Video_Block {
  int substream_id;    /* substream id */
  int temp_loc;        /* temporal locator */
  int cs;              /* codebook selector */
  int x_loc;           /* horizontal locator */
  int y_loc;           /* vertical locator */
  unsigned char Y[BLOCKSIZE][BLOCKSIZE];     /* Y data */
  unsigned char I[BLOCKSIZE][BLOCKSIZE];     /* I data */
  unsigned char Q[BLOCKSIZE][BLOCKSIZE];     /* Q data */
};
================================================================
```

The substream id identifies the substream along which the block will be transported. The temporal locator indicates the time stamp of the motion block. Codebook selector specifies the codebook to be used in vector quantization. Also, horizontal and vertical locator are the ordinate of the video block.

After spacial segmentation, the results are ready for motion and texture estimation.

3.2.4 Motion Estimation and Texture Estimation

Human eye has the characteristics of a lowpass filter in the temporal domain, and a bandpass filter in the spatial domain[5]. Theses visual characteristics are applied in our ASV coding scheme.

Perception of temporally changing stimuli is extremely important in interframe coding. The sensitivity of human eye depends on the degree of detail in image blocks. Human eye has the characteristic of low pass filter on the temporal domain, and the filtering effects are stronger in the lower spatial frequency region than in the higher spatial frequency region. That is, human eye hardly senses the image quality in the higher spatial frequency region at low frame rate.

Meanwhile, edges are very important in the perception of image quality. Human eye is very susceptible to the degradation along edges, e.g., edge jaggedness. The human eye, which can be viewed as a bandpass filter on the spatial frequency axis, has different peak frequency and passband depending on the moving speed of the image blocks. As the velocity of an object becomes greater, the peak frequency approaches zero and the relative sensitivity decreases. In other words, coarse resolution is permitted in the blocks where movements are relatively large.

Consideration of these human visual characteristics enables us to use the mixture information of motion estimation and texture estimation as factors of tuning the video block resolution level.

1) Motion Estimation

Motions are classified into three motion levels: No Motion, Low Motion and High Motion, which correspond to specific substream and codebook. The result of motion level classification is then fed into the fuzzifier to obtain the motion-fuzzified value of each video block, which will be the input to the Fuzzy Control Engine. Due to human eye characteristics, high motion blocks can use coarse resolution coding, while lower motion blocks require finer resolution coding.

Another consideration is that if a video block is detected as no motion, we can use the block of the same spatial location in the previous frame to display it. Thus we can save coding time as well as the transmission bandwidth by skipping the no motion blocks of each frame. In ASV we update a no motion block every 10 frames if motion is consistently detected.

Motion classification is performed similarly to interframe motion compensation (See figure 3.4).

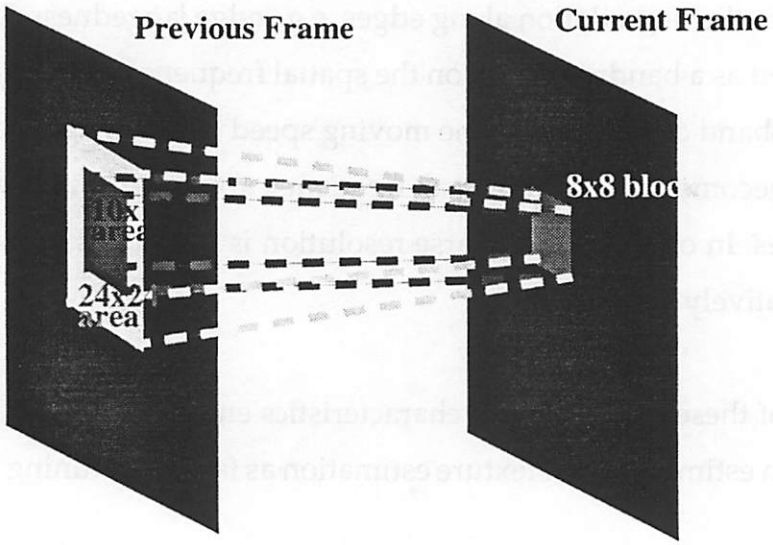

Figure 3.4 Interframe Motion Estimation

For each video block in the current frame, we compute the Local Minimum Absolute Error of this block and the position-shifted blocks in the previous frame within 24x24 pixels area. The motion level is detected based on the location where the error is below certain threshold, as well as the value of this error. If the error is detected within 10x10 pixels area in the previous frame, the block is then classified as No Motion block. If it is detected within 24x24 pixels area, then it is a Low Motion block. If the detection fails within 24x24 pixels area, then it is a High Motion block.

Figure 3.5(a) and 3.5(b) show the distribution of absolute error and distance while run-

ning frames of test video sequence Michael Jackson.

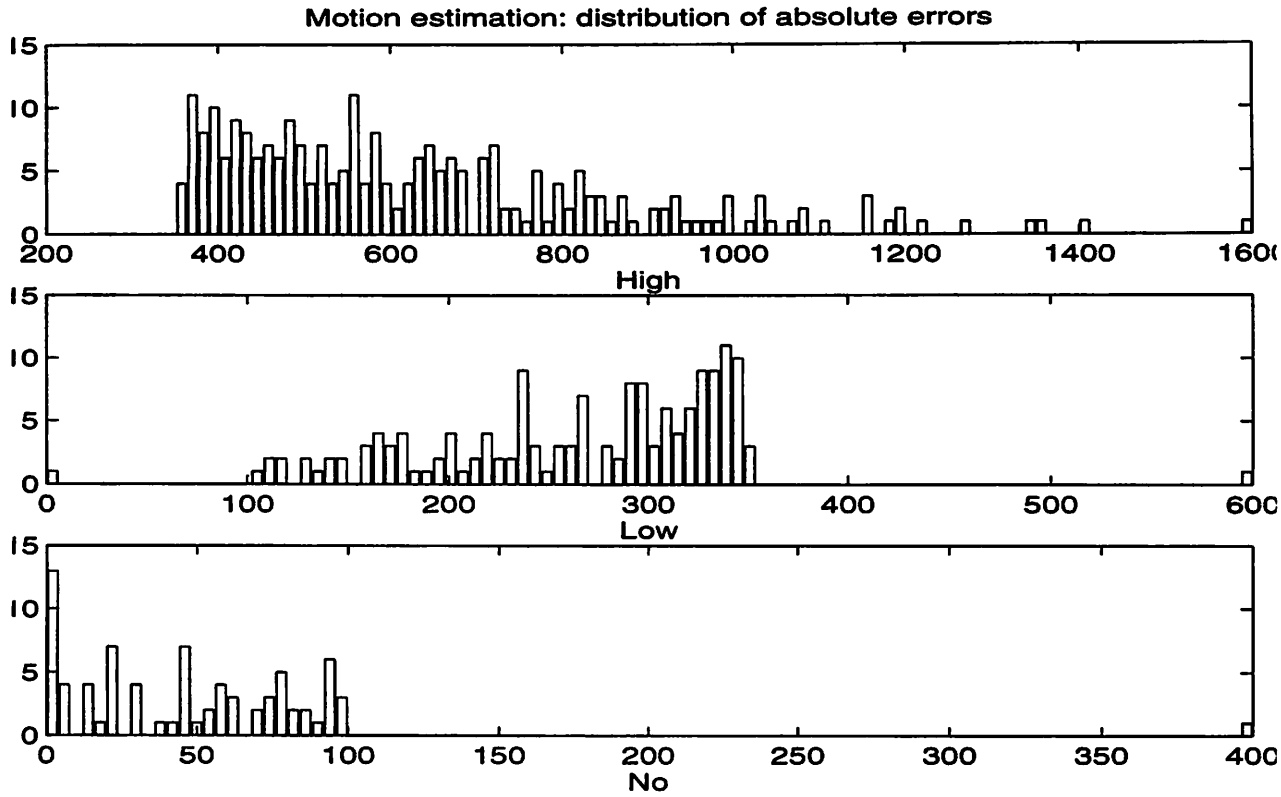

Figure 3.5(a) Distribution of absolute errors while running MJ video sequence

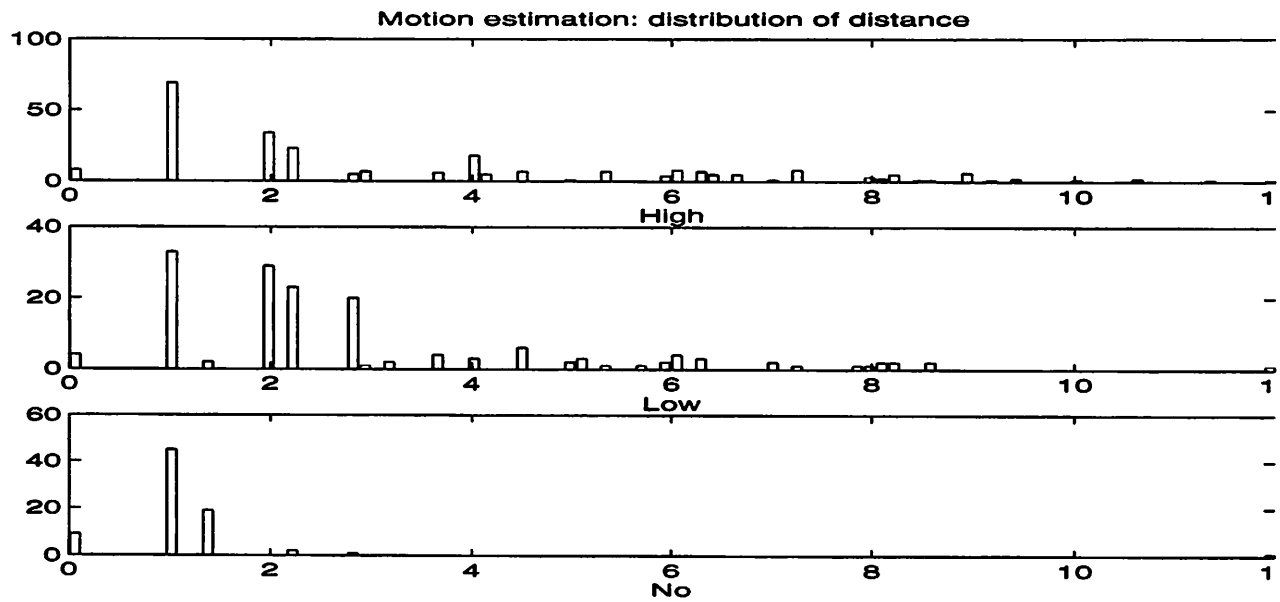

Figure 3.5 (b) Distribution of distance while running MJ video sequence

The **Motion Fuzzifier** is applied after motion classification to compute the fuzzy value of each video block. The fuzzified values of each block will be fed into the Fuzzy Control Engine to determine the substream ID and codebook selector. We fuzzify the motion information into fuzzy sets based on the distance of which the absolute value is below the threshold as well as the error itself. In order to reduce time needed to perform computations, a precalculated square value of distance is used rather than the distance $(X_{Cur}-X_{Prev})^2+(Y_{Cur}-Y_{Prev})^2$ itself to avoid the square root computation. The fuzzy sets of all the blocks in a video frame are stored in an array *fuzzified_motion[row][column][3]* where:

*[0] -- stores the fuzzy value of no_motion;

*[1] -- stores the fuzzy value of low_motion;

*[2] -- stores the fuzzy value of hi_motion;

Each element of fuzzified_motion value is calculated in the following way: for the motion level of each video block, if the error close to the upper bound of threshold, we compare the error with the 80% threshold. If the error is close to the lower bound, we compare it with 20% threshold. Then use the interpolation to calculate the percentage of each fuzzified value. Figure 3.6 shows

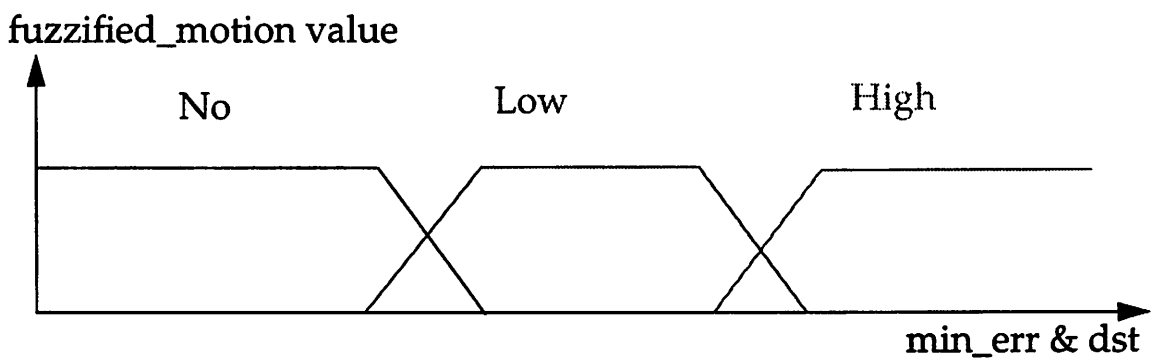

Figure 3.6 Motion fuzzifier

See appendix B motion_estimaition() for commented source code.

2) Texture Estimation

Three texture levels are defined in ASV: High Texture, Medium Texture, and Low Texture. We use the edge detection schemes to estimate texture information. For each pixel, we calculate the standard deviation of Luminance difference between this pixel and its neighborhood in the same frame as a criteria. The more edges are detected, the more texture information the block contains. Blocks with high and low texture information may be coded in lower resolution without significantly affecting subjective video quality due to human eye characteristics.

After edge detection, we fuzzify the texture information into fuzzy sets and store the value of in an array fuzzified_texture [row][column][3] where:

*[0] -- stores the fuzzy value of low_texture;

*[1] -- stores the fuzzy value of medium_texture;

*[2] -- stores the fuzzy value of high_texture;

Figure 3.7 shows the texture fuzzifier.

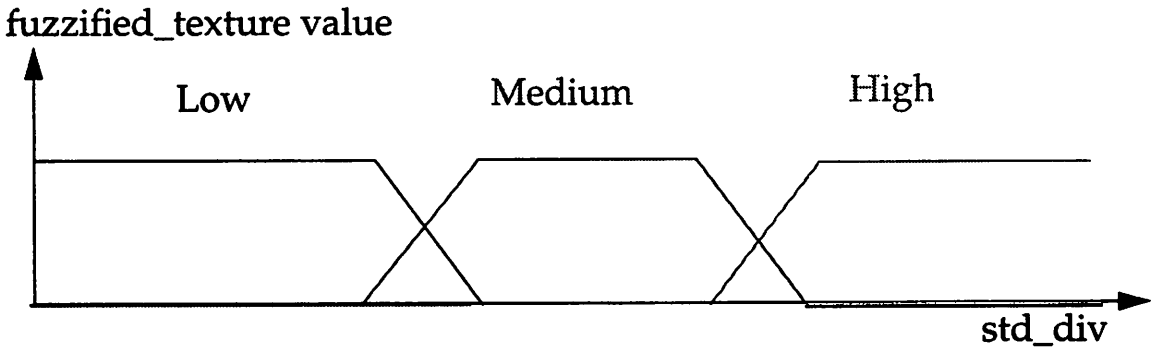

Figure 3.7 Texture Fuzzifier

3.2.5 Fuzzy Logic Control Engine

Fuzzy logic Control Engine of ASV takes the output of motion/texture estimation and rate module as its input fuzzy sets, applies the user defined fuzzy rules, and determines

the fuzzy set outputs. After defuzzification, it gives the crisp output of codebook selector and substream selector.

Table 3.1 describes the rules for determining the codebook and substream selectors of the fuzzy logic engine.

Table 3.1: Rules for the Fuzzy Logic Engine

Texture \ Motion	High	Low	No (when refreshing)
High	coarse resolution, low-delay substream	normal resolution, medium-delay substream	normal resolution, high-delay substream
Medium	medium resolution, low-delay substream	fine resolution, medium-delay substream	fine resolution, high-delay substream
Low	coarse resolution, low-delay substream	normal resolution, medium-delay substream	normal resolution, high-delay substream

The fuzzy rules in ASV implementation are also described in figure 3.8.

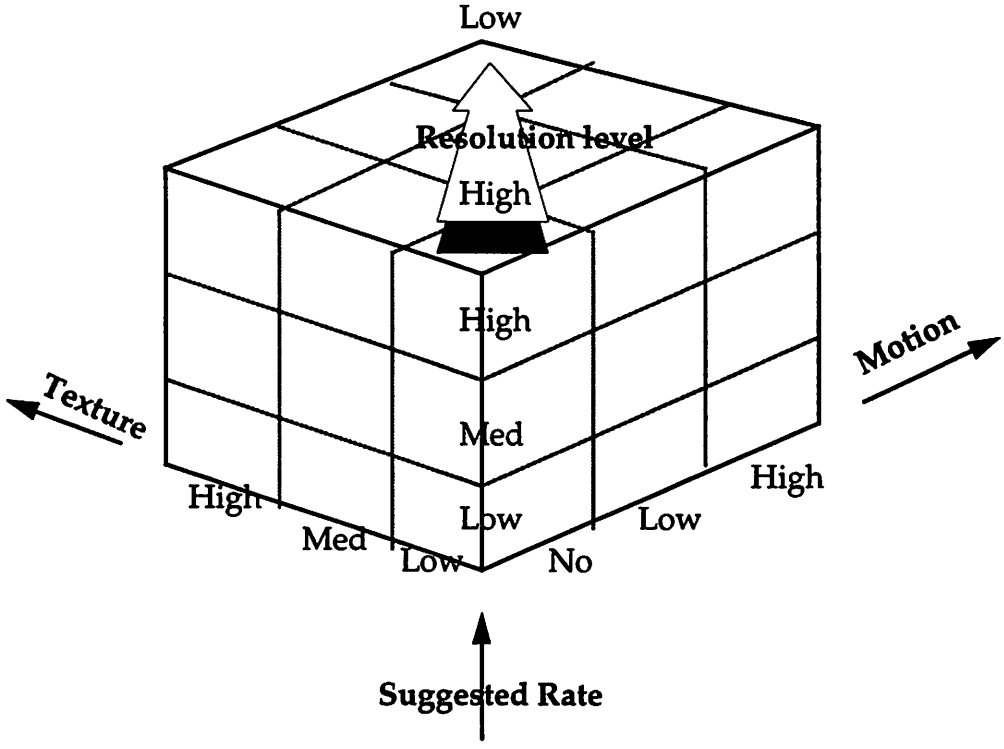

Figure 3.8 Fuzzy rules for fuzzy logic inference engine

We use VQ codebook of size 128 for coarse resolution, size 256 for normal resolution and size 512 for fine resolution. The design of these VQ codebooks are described in the next section.

3.2.6 Multiple Codebook Design

In ASV, vector quantization is used to perform video compression. To achieve adjustable resolution level and scalable transmission rate, multiple codebooks are designed using the Predictive Classified Vector Quantization (PCVQ) method[11]. We use predesigned codebook to avoid codebook transmission and to save valuable bit rate. In addition, the code vectors in the codebooks are arranged in an intelligent way to allow fast searching. Each 8x8 pixels video block uses the same codebook determined by codebook-selector from the Fuzzy Logic Engine. The size of VQ vector is 4x4. Thus, each video block contains 4 codewords.

In PCVQ, the classification information is predicted depending on a classification process called the Hadamard transform. Such a design of codebook not only classifies vector into shade and edge class, but also determines the orientation of the edge.

The Hadamard Transform Classifier (HTC) uses the Hadamard matrix H. A Hadamard matrix is characterized by an orthogonal matrix consisting of elements with values of 1 and -1 only. A Hadamard matrix of order 4 is used in the design of codebooks:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \qquad (1)$$

The Hadamard transform of an image vector X (i.e., 4x4 pixels) is given by

$$F = HXH = \begin{bmatrix} f_{00} & f_{01} & f_{02} & f_{03} \\ f_{10} & f_{11} & f_{12} & f_{13} \\ f_{20} & f_{21} & f_{22} & f_{23} \\ f_{30} & f_{31} & f_{32} & f_{33} \end{bmatrix} \quad (2)$$

The first coefficient f_{00} gives the mean intensity of the vector while the rest of the coefficients indicate the strengths of the different transitions occurring in the vector X. In general, the histograms of the coefficients other than those along the first row and first column are insignificant and can be ignored. The remaining coefficients indicate transition horizontally, via, f_{01}, f_{02} *and* f_{03}, and vertically, via, f_{01}, f_{02}, f_{03}.

In designing HTC, two arrays $F_h = [f_{01}, f_{02}, f_{03}]$ and $F_v = [f_{10}, f_{20}, f_{30}]$ are formed. Define Vector E is as:

$$E = \sqrt{e_v^2 + e_h^2} \quad (3)$$

where

$$e_h = \begin{cases} MAX[|F_h(j)|] & if \quad F_h(j) > 0 \\ -MAX[|F_h(j)|] & if \quad F_h(j) < 0 \end{cases} \quad (4)$$

$$e_v = \begin{cases} MAX[|F_v(j)|] & if \quad F_v(j) > 0 \\ -MAX[|F_v(j)|] & if \quad F_v(j) < 0 \end{cases} \quad (5)$$

E is plotted on the vector space diagram in figure 3.9. It is used to determine whether a 4x4 pixels block is a shade or an edge vector. In figure 3.8, the eight edge orientations are defined. Note that the arrow points toward the direction of transition from low to high intensity. The decision rules for shade and edge vectors are:

a) X is classified as a shade vector if $|E| < Th$, where Th is a threshold.

b) X is classified as an edge vector if $|E| > Th$, and its orientation is determined by the decision region, Rn, n = 1, 2,...,8, where it is located as shown in figure 3.9.

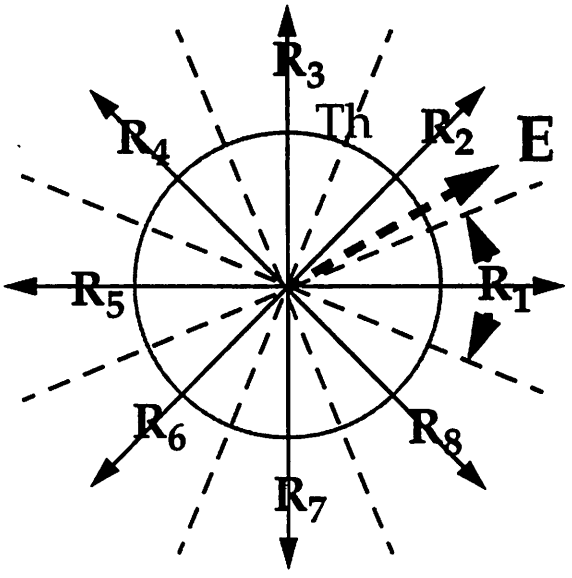

Figure 3.9 Decision regions of HTC

Mean Classification is applied to further classify the shade and each edge orientation class. The shade class is subdivided into 32 mean classes. And for each of the 8 edge orientation classes, 8 mean classes are allocated. Less number of mean classes are allocated to the edge classes because of the smaller population of edge vectors. The mean classification of the shade and edge vectors is shown in table 3.2. Note that the distribution of the mean classes for the edge class is nonuniform.

Table 3.2
Mean Classification

Mean Intensity Range		
Edge Class	Shade Class	Mean Class
0-47	0-7	0
48-79	8-15	1
80-103	16-23	2
104-127	24-31	3
128-151	32-39	4
152-175	40-47	5
176-205	48-55	6
206-255	56-63	7
-	64-71	8
-	72-79	9
-	etc.	etc.

Three codebooks of size 128, 256 and 512 are designed using PVCQ and Mean Classification.Table 3.3 shows the construction of each codebook.

Table 3.3: Construction of codebooks

Codebook Size	# of shade codevector	# of edge codevector
128	32	96
256	64	192
512	64	448

Figure 3.10(a), (b) and (c) show the codebooks of size 128, 256 and 512 respectively (codebooks are displayed in Sun Raster format).

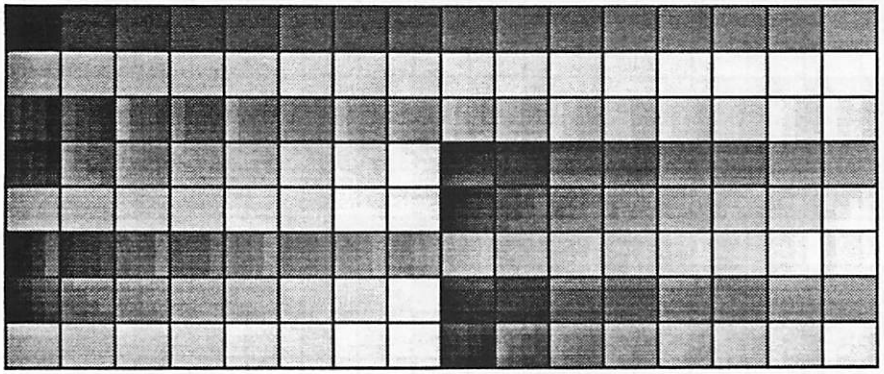

Figure 3.10(a) Codebook of size 128

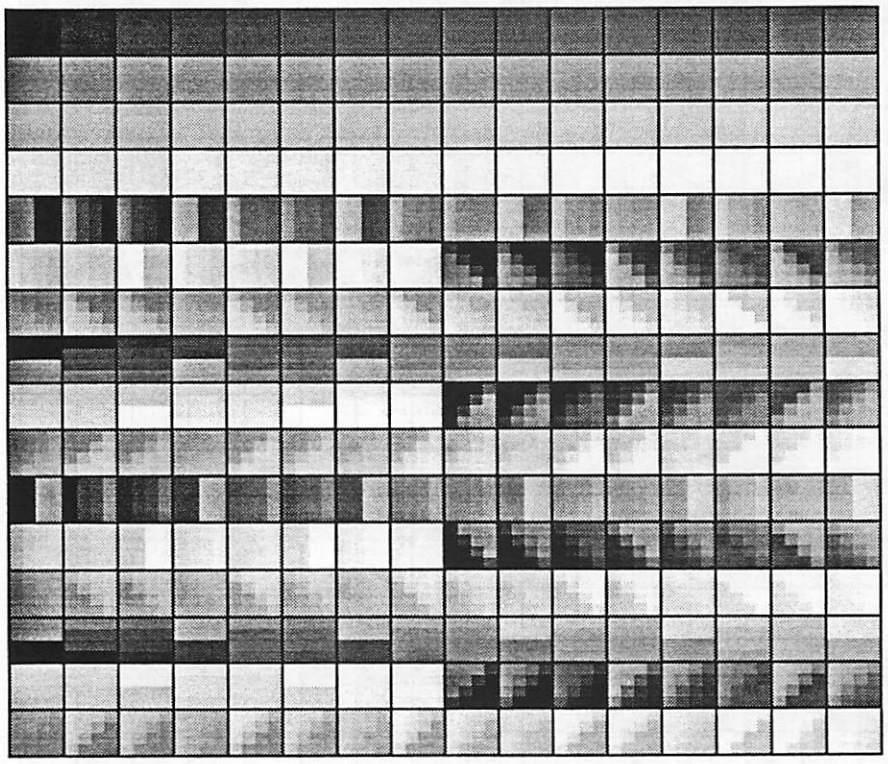

Figure 3.10(b) Codebook of size 256

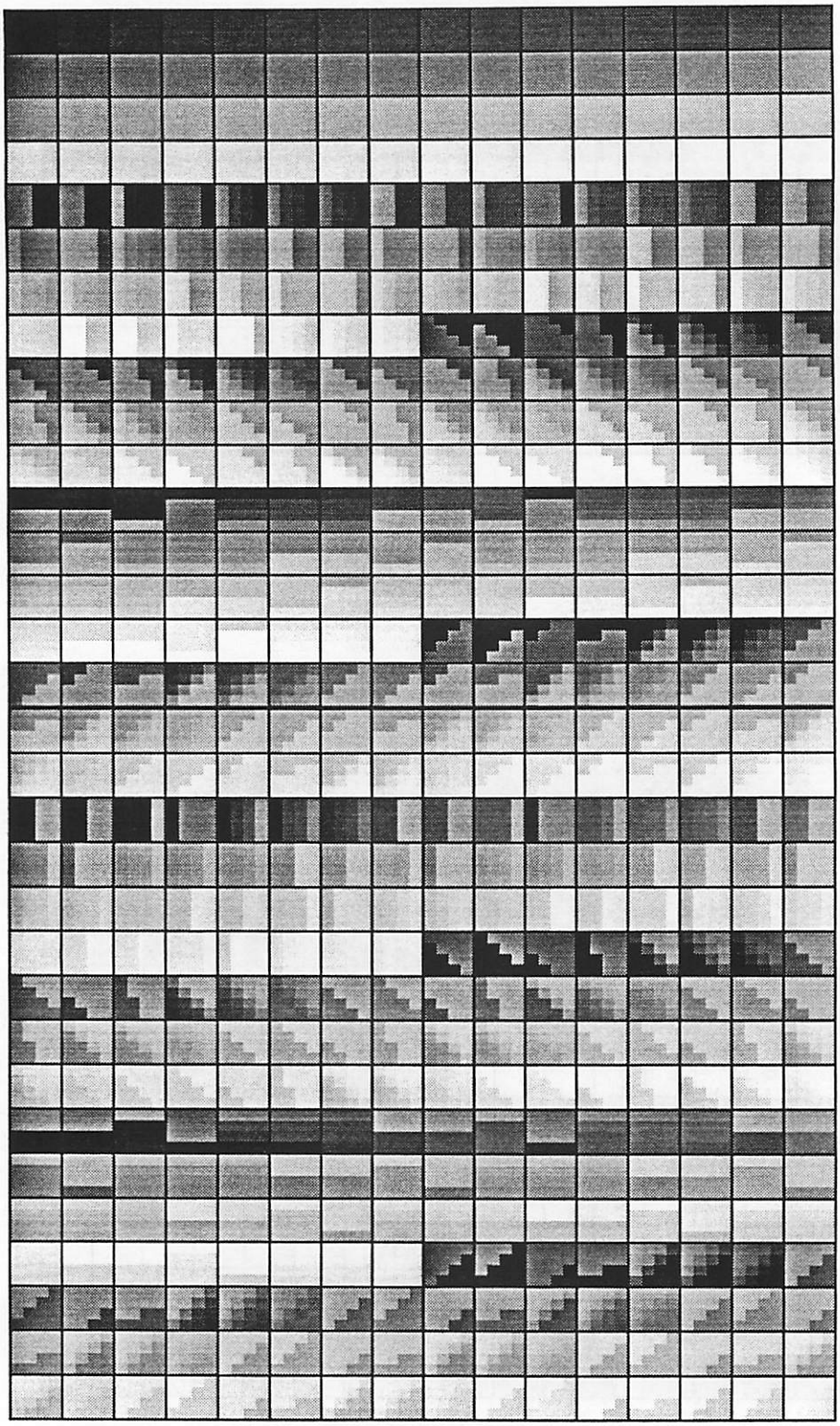

Figure 3.10(c) Codebook of size 512

3.3 ASV File Format

The ASV video file consists of codebooks and codewords of each frame in a video sequence. Three codebooks are sitting at the top of the ASV file with size of 128, 256 and 512 respectively. Each frame is separated by a frame breaker mark. Each frame contains video blocks with structure Video_Blocks.

```
================================================================
struct Video_Block {
  int substream_id;    /* substream id */
  int temp_loc;        /* temporal locator */
  int cs;              /*codebook selector */
  int x_loc;           /* horizontal locator */
  int y_loc;           /* vertical locator */
  unsigned char Y[BLOCKSIZE][BLOCKSIZE];     /* Y data */
  unsigned char I[BLOCKSIZE][BLOCKSIZE];     /* I data */
  unsigned char Q[BLOCKSIZE][BLOCKSIZE];     /* Q data */
};
================================================================
```

The header of the VQ file has the structure as follows:

```
================================================================
#define VQ_MAGIC_NUM     0x49505651
#define CODE_VECTORS           896
#define VECTOR_BYTES            16
/* All numbers should be in net order */
typedef struct _VQ_file_header {
   u_long   magic_number;
   u_short  major_version;
   u_short  minor_version;
   u_long   width;
   u_long   height;
   u_long   frame_size;
   u_long   codebook_entries;
   u_long   codebook_entry_size;
   u_long   frames_per_sec;
   u_long   extra_data_len;
} VQ_file_header;
================================================================
```

3.4 Implementation of ASV decoder

In this section we describe our software implementation of the ASV decoder.

3.4.1 ASV decoding flow chart

Figure 3.11 shows the ASV decoding flow chart.

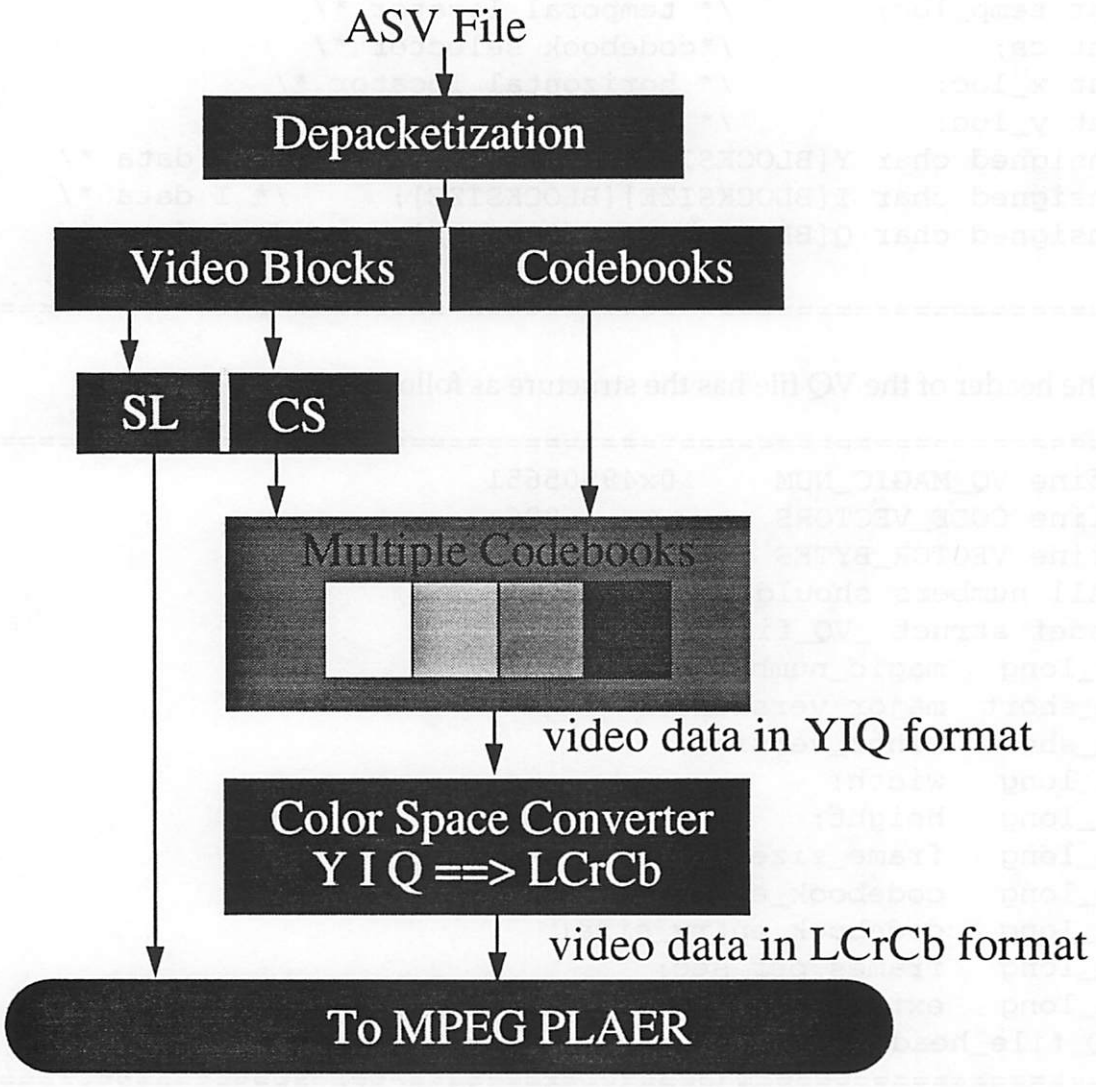

Figure 3.11 Decoding flow chart

3.4.2 ASV Decoding Scheme

At the receiver, the MPEG player is used as a facility to play back the video sequence. Whenever the VQ file arrives, the decoder depacketises the file, stores the codebook into memories for table look up, and decodes all video blocks in each frame. After that, color space transformation is applied again to transform the YIQ VQ format to LCrCb MPEG format to enable the MPEG play back.

Overall ASV is a front end transparent application. It provides good subjective video quality over a wireless access environment with perceptual low delay and high traffic capacity.

Chapter 4

Simulation Result

We have tested ASV with different video sequences. Currently, the real-time coding and decoding speed is about 7 frames/sec on a SunSparc 20 workstation. While the speed of playing back an ASV file is over 30 frames/sec.

The results of two testing MPEG video sequences, Jets and Michael Jackson, coded with MPEG and ASV, are compared in figure 4.1. Due to the vector quantization algorithm used by ASV, the block effect is hard to overcome. We can see in JETS video sequence some of the edge information of the plane wings is lost in vector quantization.

Figure 4.2 shows the traffic along different substreams, low delay, medium delay and high delay respectively, of testing sequence Michael Jackson. The high motion portion of the frame is sent along the low delay substream to achieve perceptual good subjective video quality, while low motion portion of the same frame is transported along high delay substream to reduce the traffic load of the transport layer. These results are consistent with the goal of ASV.

The simulation results of this first software implementation of ASV are encouraging.

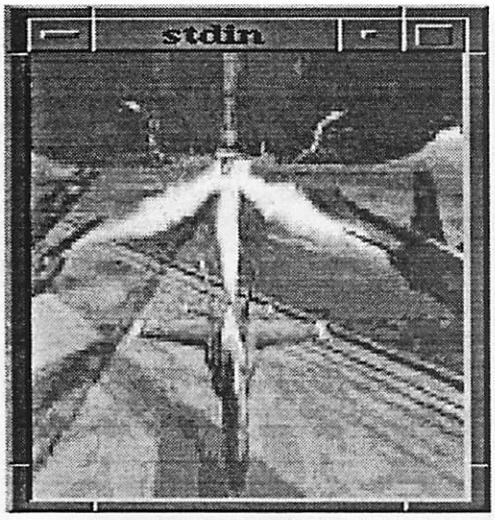

Fig. 4.1(a) JETS in MPEG

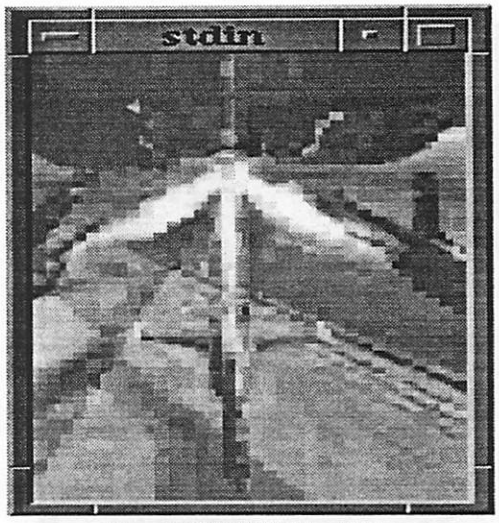

Fig. 4.1(b) JETS in ASV

Fig. 4.1(c) MJ in MPEG

Fig. 4.1(d) MJ in ASV

Figure 4.1 Comparison of MPEG Video and ASV Video

Fig 4.2(a) MJ in ASV display

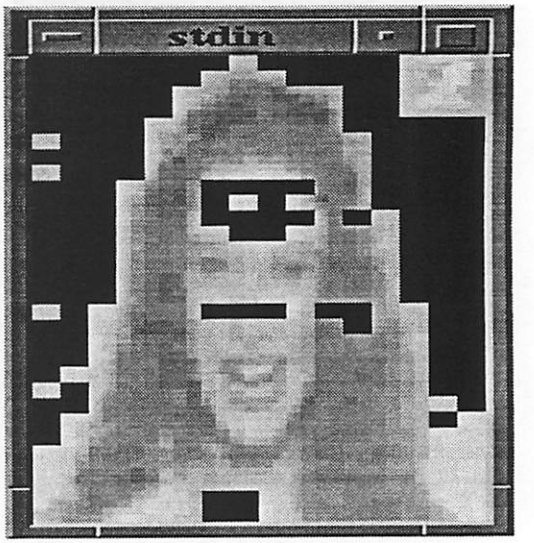

Fig4.2(b) MJ Low Delay Substream

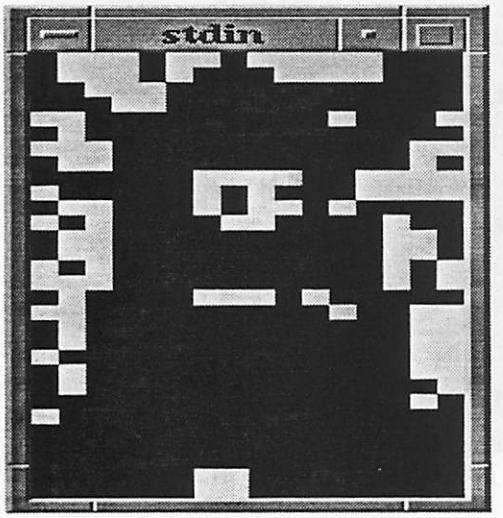

Fig. 4.2(c) MJ Medium Delay Substream

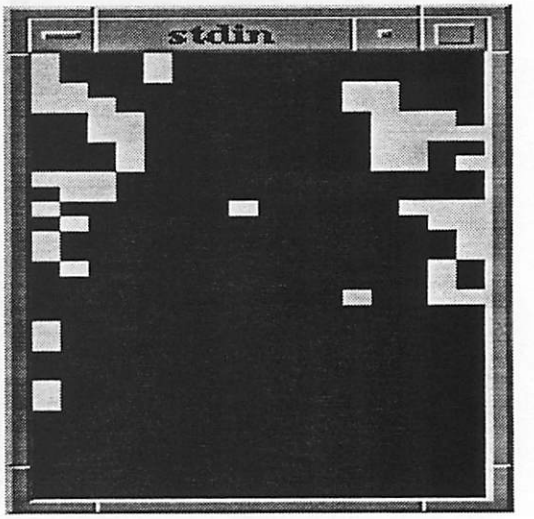

Fig 4.2(d) MJ High Delay Substream

Figure 4.2 Substreams of MJ Video Sequence

Chapter 5

Conclusion

We have done a implementation of the ASV concept. The simulation results show that the ASV has potential for wireless video transportation.

The current real time coding speed is about 7 frames per second.The bottleneck of the processing speed is the motion estimation part. We have put much effort into reducing the computation time. A possible solution (not tried yet) is to take advantage of the motion vector information in MPEG source code to avoid major part of motion estimation computation.

The block effect of the video display is subjective impairment in the vector quantization compression method. Though we could have use another sophisticated VQ method to train the codebook, yet such VQ requires codebook transmission which will consume large amount of bit rate, thus it is considered not suitable for wireless communication environment. Possible solution for improving subjective video quality is to perform subband coding[16] vector quantization which uses DCT transformation information. Video quality may be improved with the expense of computing overhead.

Further working can be focus on packetization to reduce transmission rate. Also merging the ASV application into network where rate control (or rate policing) and/or power control[17] are provided, is challenging but worth trying.

Our ASV implementation and simulation results shows that ASV is a good applica-

tion for wireless video communication. We look forward to the extension of the current ASV implementation with the hope of providing a valuable wireless video communication technique.

References

[1] Chan, C.K.and Ma, C.K., "A Fast Method of Designing Better Codebooks for Image Vector Quantization", IEEE transactions on communications, Vol. 42, No. 2/3/4, February/March/April 1994, pp237-242.

[2] Driankov, D., Hellendoorn, H. and Reinfrank, M., "An Introduction to Fuzzy Control", by Springer-Verleg publishing 1993.

[3] Haskell, P.and Messerschmitt, D. G., "A Signal Processing Perspective on Networking", invited paper, Department of EECS, UCB.

[4] Hirota, K., "Industrial Applications of Fuzzy Technology", by Springer-Verlag publishing 1993.

[5] Kim, J.T., Lee, H. J. and Choi, J. S., "Subband Coding Using Human Visual Characteristics for Image Signals", IEEE Journal on Selected Areas in Communications, Vol. 11, No. 1, January 1993, pp59-64.

[6] Lao, A., Reason, J and Messerschmitt, D. G., "Asynchronous video coding for wireless transport", IEEE Workshop of Mobile Computing Systems and Applications, Santa Cruz, CA., Dec. 1994.

[7] Lee, I., Kim, J. G., Halzan, S., and Ann, S.J. M., "Motion Dependent Vector Quantizer with Multiple Sub-codebooks for Image Coding", IEEE transactions on Consumer Electronics, Vol. 39, No. 4, November 1993, pp765-772.

[8] Lim, J. S., "Two-dimensional Signal and Image Processing", by P T R Prentice Hall, 1990.

[9] Meng, T. H., Gorden, B. M., Tan, W.C. Tsern, E.K., and Hung, A.C., "Portable Video-on-Demand in Wireless Communication", invited paper, Department of Electrical Engineering, Stanford University.

[10] Nasrabadi, N.M. and King, R. A., "Image Coding Using Vector Quantization: A Review", IEEE transactions on Communications, Vol. 36, No. 8, August 1988, pp957-969.

[11] Ngan, K. N. and Koh, H.C., "Predictive Classified Vector Quantization", IEEE transactions on Image Processing, Vol. 1, No. 3, July 1992. pp269-281.

[12] Pennebaker, W. B. and Mitchell, J. L., "JPEG Still Image Data Compression Standard", by Van Nostrand Peinhold Publishing, 1993.

[13] Ramamurthi, B. and Gersho, A., "Classified Vector Quantization of Images", IEEE transactions on Communications, Vol. COM-34, No. 11, November 1986, pp1105-1115.

[14] Sheng, S. Chandrakasan, A.P., Brodersen, R.W., "A Portable Multimedia Terminal", IEEE Transactions Magazine, pp.64-75, December 1992.

[15] Supangkat, S. H., and Murakami, K.,. "Quantity control for JPEG Image Data Compression Using Fuzzy Logic Algorithm", IEEE Transactions on Consumer Electronics, Vol. 41, No. 1, February 1995.

[16] Taubman, D. and Zakhor, A., "Multi-Rate 3-D Subband Coding of Video", First Submitted to IEEE transactions on Image Processing, April 1993.

[17] Yun, L.C. and Messerschmitt, D. G., "Power Control and coding for variable QOS on a CDMA channel" Proc. IEEE MILCOM Conf., Fort Monmouth, NJ, Oct 2-4, 1994.

Appendix A

Listing of Data Structures

```
% Video Block Data Structure
  struct Video_Block {
    int substream_id;    /* substream id */
    int temp_loc;        /* temperal locator */
    int cs;              /* codebook selector */
    int x_loc;           /* horizontal locator */
    int y_loc;           /* vertical locator */
    unsigned char Y[BLOCKSIZE][BLOCKSIZE];     /* Y data */
    unsigned char I[BLOCKSIZE][BLOCKSIZE];     /* I data */
    unsigned char Q[BLOCKSIZE][BLOCKSIZE];     /* Q data */
  };
```

% RescaleInfo -- used in resizing image.(3.2.1)

```
/* RescaleInfo is all of the aux info needed to do fast rescaling */
/* of an image from the original mpeg sized image to a reduced VQ */
/* image. There is one per transformation - i.e. one for the L-Y */
/* rescaling and one for the CrCb->IQ transformations. */

  typedef struct {
    /* orig_width, orig_height, scaled_width, scaled_height are
    * the dimensions of the original and scaled images */
   int        orig_width;
   int       orig_height;
   int       scaled_width;
   int       scaled_height;

   /* orig_bytes_per_line and scaled_bytes_per_line allow
   * selecting parts of the source and scaled image for
    * zooming and padding*/
   intorig_bytes_per_line;
   intscaled_bytes_per_line;

   /* x_orig2scaledlinebuf is used if the original width is
    * larger than the scaled width. It maps from the X coord
    * in the original image into the element in linebuf which
    * corresponds to the scaled image position. It will be
```

```
   * used to select into which linebuf bin each original
   * pixel goes */
   int        **x_orig2scaledlinebuf;

/* x_scaled2origoffset is used if the original width is smaller
 * than or the same as the scaled width.  It maps from the
 * scaled X coord to the original's x coord. It will be used
 * to select from which original pixel each linebuf pixel
 * comes */
   int        *x_scaled2origoffset;

/* linebuf is where the X size-scaled array is kept. It is
 * the sum of all of the pixels falling into it from all of
 * the unscaled rows corresponding to the current scaled pixel.
 * If the original height is smaller than this just collects
 * for one original row. */
   int     *linebuf;

/* x_scaled2val_scale_large and x_scaled2val_scale_small
 * are used to map from the shifted line_buf value which range
 * from 0 to 511 into pixel values ranging from 0 to 255.
 * For each scaled bin they point to one of the four xval_scale
 * arrays. x_scaled2val_scale_large is used if the original
 * height is smaller or equal to the scaled height or if
 * the original height is larger but this scaled y corresponds
 * to the larger of the y quantization steps.
 * x_scaled2val_scale_small is used if the original
 * height is larger and this scaled y corresponds
 * to the smaller of the y quantization steps.  */
   u_char    **x_scaled2val_scale_large;
   u_char**x_scaled2val_scale_small;

/* orig_rows_per_scaled_y counts how many source rows correspond
 * to the current scaled row. If the source height is larger
 * then this will determine how many rows to collect. If the
 * source height is smaller then it will say 1 to get new
 * values or 0 to skip. In this case the first entry will
 * always be 1. The sum of the values will always be
 * the source image height. */
   int        *orig_rows_per_scaled_y;

/* The value for the smaller rows just for quickly determining
 * whether to use small or large arrays */
   intsmall_rows_per_scaled_y;
```

```
   /* xval_scale holds the scale values pointed to by
    * x_scaled2val_scale_large and x_scaled2val_scale_small
    * There are 4 arrays of 512 values */
      u_char    xval_scale[512*4];

   /* shift_bits_large_y and shift_bits_small_y are the number
    * of bits to shift to get the value into 0..511 range before
    * indexing into the scale array.  They are indexed by
    * scaled X column */
    int*shift_bits_large_y;
    int*shift_bits_small_y;
   } RescaleInfo;


%  VQ File Header

   typedef struct _VQ_file_header {
      u_longmagic_number;
      u_shortmajor_version;
      u_shortminor_version;
      u_longwidth;
      u_longheight;
      u_longframe_size;
      u_longcodebook_entries;
      u_longcodebook_entry_size;
      u_longframes_per_sec;
      u_longextra_data_len;
   } VQ_file_header;
```