# A Layered Video Object Coding System Using Sprite and Affine Motion Model

Ming-Chieh Lee, *Member, IEEE*, Wei-ge Chen, *Member, IEEE*, Chih-lung Bruce Lin, *Member, IEEE*, Chuang Gu, Tomislav Markoc, Steven I. Zabinsky, and Richard Szeliski, *Member, IEEE*

***Abstract*— A layered video object coding system is presented in this paper. The goal is to improve video coding efficiency by exploiting the layering of video and to support content-based functionality. These two objectives are accomplished using a *sprite* technique and an affine motion model on a per-object basis. Several novel algorithms have been developed for mask processing and coding, trajectory coding, sprite accretion and coding, locally affine motion compensation, error signal suppression, and image padding. Compared with conventional frame-based coding methods, better experimental results on both hybrid and natural scenes have been obtained using our coding scheme. We also demonstrate content-based functionality which can be easily achieved in our system.**

***Index Terms*— Affine motion model, image padding, layered video object coding, MPEG-4, scalability, shape coding, sprite coding.**

## I. Introduction

OBJECT-BASED and related layered coding have been active topics of research in the coding field [18], [19], [4]. By coding each object/layer separately, improved coding efficiency and many content-based functionalities can be realized. Recently, the MPEG community chose a layered coding architecture for the prototype, called Verification Model, of the new international video compression standard MPEG4 [3], [20]. This standard emphasizes multiple functionalities such as the following.

- *Object scalability*: Each object can be coded and decoded independently of others, so that decoder-end video editing (without decompression) or selected access of multimedia databases can easily be accomplished.
- *Spatial scalability*: Each object can be decoded at several specified spatial resolutions. Different objects can be encoded at different spatial resolution.
- *Temporal scalability*: Each object can be decoded at several temporal resolutions. Different objects can be encoded at different temporal resolutions.
- *Coding of graphics and hybrid data*: Synthetic video from computer graphics and synthetic-natural hybrid video are becoming increasingly important sources of digital video at such a pace so that coding of these data requires special attention.

Manuscript received March 10, 1996; revised July 1, 1996. This paper was recommended by Guest Editors Y.-Q. Zhang, F. Pereira, T. Sikora, and C. Reader.

The authors are with the Microsoft Corporation, Redmond, WA 98052 USA.

Publisher Item Identifier S 1051-8215(97)00880-X.

This paper describes a video coding system which exploits a layered representation and coding of video frames. In fact, it constitutes Microsoft Corporation's proposal to the MPEG4 standardization process. Because of its improved coding efficiency and the support for diverse functionalities sought by MPEG4, many parts of this layered coding scheme have been selected as fundamental components of the first MPEG4 Verification Model.

A layered representation [4] considers a video frame as the superposition of a number of layers. For example, the video frames of the sequence "Fish" shown in Fig. 14 can be decomposed into three layers, namely the fish, the caption and the background (fourth row of Fig. 14). The whole video frame becomes a sorted (according to the depth of each layer) linear combination of the layers. The weighting factor of each layer can be loosely defined as the alpha channel (a collection of alpha values) of that layer, which essentially represents the transparency of the pixels in that layer. For example, if we overlay a pixel $A$ with alpha value $\alpha_A$ on top of $B$, the resulted pixel will have an intensity value of $\alpha_A I_A + (1 - \alpha_A) I_B$ where the alpha value $\alpha_A$ represents the fraction of intensity of $A$ which will be seen and the rest of the fraction comes from $B$. Such compositions have been proven indispensable in computer graphics since they allow selective rendering of only part (layer) of the image that needs to be updated. Likewise, the exact reason should be at effect for video coding as far as the compression efficiency is concerned. This is particularly the case for computer generated synthetic images where the layered decomposition is readily available. Our layered coding scheme takes exactly such a viewpoint. For example, if the background of the video sequence is stationary, it will be coded only once and never be updated. As a result, our video coding system takes advantage of the fact that the properties or parameters of graphics objects are readily available and do not rely on image analysis, and thus codes graphics objects at a higher efficiency. On the other hand, we have noticed that while the readily available layer decomposition is very convenient for synthetic video, for natural video sequences, it can be obtained by using moving image segmentation [23], [21]. Usually, the segmentation procedure has to be considered as a significant overhead for any layered coding schemes. Such topics are beyond the scope of this paper.

The organization of this paper is as follows. Section II gives an overview of the proposed video coding system. Section III describes the main algorithms used in our coding scheme. Section IV illustrates the encoding process which involves the coding of all the layers, namely, their shapes (masks or gray-level alpha-channels), trajectories and textures. Section V shows the corresponding decoding process. Section VI demonstrates the experimental results. Finally, Section VII provides the conclusions.

## II. Overview of the Video Coding System

Besides its layered object-based video coding architecture, our video coding system is also noteworthy for several major innovations. First, in addition to the coding of image textures in existing schemes, the associated alpha channel of each layer may be coded. Since the alpha channel is similar to a color plane, its coding can be expensive at times. In order to maximize coding efficiency, we adopted an adaptive approach. If a layer consists mostly of opaque pixels, its alpha channel can be adequately described by a binary mask, in which case we use contour coding to represent the shape of the object. Otherwise, the gray-level alpha channel is coded as another color plane. Sometimes, we refer to these two approaches together as shape coding.

Second, we classify the layers of video frames into two categories: sprite layers and ordinary layers. A sprite is a large object built up from pixels in potentially many frames which is transmitted first and then warped and cropped (masked) to form a part of an object in subsequent frames. The portions of objects created from sprites do not usually have an associated error signal. For example, the sprite for a stationary background is a union of its visible part in all of the frames in the sequence. During decoding, the sprite is warped into the current frame with a small amount of additional information, i.e., the trajectories of the sprite. Note that a sprite is not necessarily a background. The use of a sprite provides two important advantages to our coding scheme. First, for some objects that undergo rigid two-dimensional (2-D) motion, sprite composting is good enough to generate estimated objects so that motion estimation and error signal encoding are avoided. Second, we use sprites to identify parts of an object in a particular frame that cannot be predicted from the previous frame. Once identified, these parts can be encoded more efficiently than traditional motion-compensated coding methods. An ordinary layer will be coded with conventional motion-compensated transform coding, e.g., the H.263 standard [14].

Third, we have developed several new algorithms to solve problems that arise specifically from coding of arbitrarily shaped objects in a layered-coding scheme. In our layered object-based video coding scheme, compression, decompression, and processing are done on a *per-object* basis. There are six major tools in our video coding system.

- *Mask processing and coding*: The lossy simplified chain code, as its name suggests, is a lossy simplified version of the well-known chain code. This contour coding method improves contour coding efficiency with almost invisible loss of the boundary information.
- *Sprite accretion*: For natural objects whose motion can be modeled by sprite warping, the sprites need to be accreted if they are not provided. The accretion process uses a planar perspective transformations to describe the deformations between a sprite and the frames where the sprite appears.
- *Trajectory coding*: It was found that the perspective and affine motion parameters are not suitable for motion coding. Another representation, namely, "the coordinate representation," is developed for both global perspective and local affine motion compensation. In addition, the coordinate representation enables a scalable representation of motion.
- *Local affine motion estimation*: Arbitrarily shaped objects require special attention in terms of motion prediction and compensation. In our approach, first, "fast block (polygon) matching" is applied to each pixel inside the object to obtain the dense motion vector field. Next, an affine motion model is used to represent the dense motion within a block.
- *Error signal suppression*: This technique eliminates residual signals that hardly degrade visual quality when not coded. An effective nonlinear filtering of residual signals is utilized.
- *Image padding*: Padding is important for coding of arbitrarily-shaped regions. A repetitive padding method is developed, which can be useful not only for coding but also for object-based motion estimation and compensation that is frequently encountered in object-based coding.

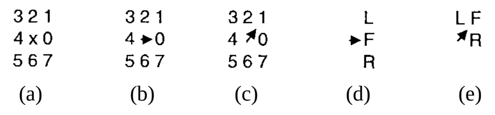


Fig. 1. Eight-connected chain code.

## III. Algorithms

### A. Mask Processing and Coding

*1) Simplified Chain Coding:* The mask, a basic data type enabling object-based processing, is defined as a 1-b alpha channel. If the pixel value of a mask is one (opaque), it means that the object is valid at this pixel location. If it is zero (transparent), the object is not defined there. In order to compress video efficiently, lossy contour coding technique to describe the shape of masks should be developed. Contours on a rectangular grid can be described by a traditional eight-connected chain code [10], [11] [see Fig. 1(a)]. Given the location of the starting point and the chain code for each subsequent point, the entire contour can be reconstructed. Any continuous contour can be described in this way. This method requires 3 b per contour pixel before entropy coding. Comparied to other contour coding techniques such as polygon approximation, chain code normally provides better results in terms of complexity, efficiency, and visual quality for nature images.

Lossless simplified chain code [22] is a modification of the chain code which only requires three symbols to represent the movements of a chain. Furthermore, lossy simplified chain code was developed in [12]. This lossy technique reduces the number of bits necessary to describe each link in the chain and, furthermore, does this in such a way that naturally leads to further gains when entropy coding is used on the simplified chain. A new lossy eight-connected simplified chain code is designed which only leads to minor loss of resolution for the original contour while improving the contour coding efficiency even more.

*2) Simplified Chain Encoding:* It was discovered experimentally that most of the links either preserve direction or change it by only one pixel left or right. There are two cases: when the previous point is along the grid (chain codes 0, 2, 4, 6) and when the previous point is diagonal (chain codes 1, 3, 5, 7). Consider the case of previous point 0 [Fig. 1(b)] as the representative of the "along the grid" case. Codes 0, 1, and 7 are the most common, with the other codes occurring less than 0.2% of the time in our tests. Similarly, in the diagonal case represented by previous code 1 [Fig. 1(c)], the codes 1, 0, and 2 are the most common with the other codes again occurring less than 0.2% of the time.

The simplification consists of using these three possibilities to define an eight-connected chain movement: forward, left, and right (F, L, and R, respectively). This requires that the original chains be modified, since sections of high curvature cannot be represented with this simplified code. These changes can be made to any contour encoded as an eight-connected chain code, or the simplification can be made as the chain is being encoded.

The simplification of an eight-connected encoded chain is best described by examining all possible cases. First, when the chain code can be described by the new code F, L, or R, simply substitute the new codes for the old. When this is not possible, there will be three modifications each for the "along the grid" [Fig. 1(d)] and the "diagonal" case [Fig. 1(e)], neglecting rotations and reflections. We will use the sequences 02x, 03x, and 04x as the representatives of "along the grid" and sequences 13x, 14x, and 15x as the representatives of "diagonal." Now, 02 becomes 1, 03 becomes 2. Both of these cases "cut off" the sharp corners. Case 04 is just removed from the chain, since it represents a contour that is one pixel left, then one pixel back right. Again cutting off the sharp corner, 13 becomes 22, 14 becomes 2. Case 15 is removed. These modifications are shown in Fig. 2. Note that all of these modifications involve reducing the curvature in regions where the curvature is very high, that is, 90 degrees or more over two pixels. The sharpest curve possible in the simplified code is a right angle over three pixels. The loss of these extremely fine details has not been apparent in the decoded sequence.

It sometimes happens that the modifications described above introduce sharp curvature in the contour. These cases can be handled either by iterating the simplification process until no changes are made, or by looking backward as the simplification takes place and handling these cases as they arise. The iterative method is much simpler to implement, and the "look backward" method may be faster for chains where several iterations would be necessary with the first method.

After the chains have been encoded using the simplified chain code (F, R, L), they are entropy coded. Experimental results show that the forward case F occurs about 50% of the time, with the right and left cases each at about 25%, which suggests that a first-order Huffman code will work very well, representing F by 0, R and L by 10 and 11, respectively. Indeed, bitrate of about 1.47 b per pixel (bpp) is typical for our contours derived from masks for video compression. Comparing to the polygon approximation, we conclude that the lossy eight-connected simplified chain code normally provides better results for nature images.

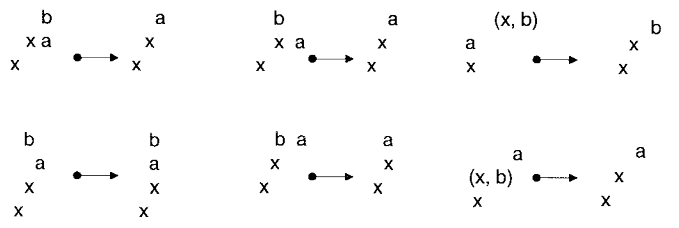


Fig. 2. Modifications of special cases: "x" represents the existing chain, "a" is the penultimate point to consider and "b" is the last point in the chain. "(x, b)" indicates that the last point "b" coincides with a point "x" which is already part of the chain.

## B. Trajectory Coding

A particular feature point of an object in a frame has a corresponding location in the next frame. A list of the coordinates of such a feature point through the sequence constitutes a *trajectory*. As will be seen in the next sections, a number of feature points are used to globally warp the *sprite* into the current frame or locally motion-compensate an object. In these two cases, trajectories define the global perspective and local affine transforms, respectively. The computed perspective or affine coefficients are nonintegers. Hence, we have to quantize the coefficients before transmission or storage. Obviously, we have to be careful in quantizing these coefficients since the quality of motion compensation will be greatly affected.

Instead of directly quantizing the eight (perspective model) or six (affine model) coefficients, we quantize the coordinates of four pairs (perspective) or three pairs (affine) of $(x, y)$ and $(x', y')$ for a sprite or a block. This coordinate representation turns out to be much less sensitive to quantization noise and yields better quality than directly quantizing the coefficients. Furthermore, coding point coordinates provides a convenient way to achieve *motion scalability*. Motion with increasing degree of complexity can be easily represented and coded by using an increasing number of point pairs. For example, one pair of $(x, y)$ and $(x', y')$ is sufficient to describe translational motion which can be modeled as

$$\begin{aligned} x' &= x + c \\ y' &= y + f \end{aligned}$$

because the decoder has enough information to solve for the unknowns $c$ and $f$ with information of one pair of points. In fact, this mode coincides with MPEG-2's motion estimation/compensation. If we add one more pair, rotation, magnification, and translation can be described. For example, anisotropic magnification and translation can be modeled as

$$x' = ax + c$$
$$y' = ey + f.$$

Rotation, isotropic magnification, and translation can described simultaneously by

$$x' = a\cos\,\theta x - a\sin\,\theta y + c$$
$$y' = a\sin\,\theta x + a\cos\,\theta y + f.$$

Both models have four unknowns and thus the decoder needs four equations, i.e., two point pairs, to solve for the unknowns. Shear can be described if the third pair is added, which constitutes the complete affine model

$$x' = ax + by + c$$
$$y' = dx + ey + f.$$

If one more pair is added, we can obtain perspective transformation

$$x' = (ax + by + c)/(gx + hy + 1)$$
$$y' = (dx + ey + f)/(gx + hy + 1).$$

However, to reduce the degradation of motion prediction introduced by the coordinate quantization, the configuration of the these points has to be chosen with care. For example, good performance can be obtained when they form an equilateral triangle for an affine model or a rectangle for the perspective model. For simplicity, in the case of affine model, we choose the three points to be the center, left-top corner, and right-top corner of the block. In the case of perspective model, four corners of a rectangle are used.

In the following, we describe the quantization of the coordinates of the point pairs in detail. The location of the feature point is first inserted to the head of each trajectory. Each component ($x$ or $y$) of an augmented trajectory is differentially coded and, then, concatenated before coded by the QM-coder [17]. The bitstream contains the coordinates of the points and the coordinate difference of the points in the subsequent frames. In other terms, if $(x_i^{(j)}, y_i^{(j)}), i = 1, \cdots, T, j = 1, \cdots, N$ are feature points and $N$ trajectories for total of $T$ frames, respectively, $x_1^{(1)}, \Delta x_2^{(1)}, \cdots, \Delta x_T^{(1)}, y_1^{(1)}, \Delta y_2^{(1)}, \cdots, \Delta y_T^{(1)}; x_1^{(2)}, \Delta x_2^{(2)}, \cdots, \Delta x_T^{(2)}, y_1^{(2)}, \Delta y_2^{(2)}, \cdots, \Delta y_T^{(2)}, \cdots, x_1^{(N)}, \Delta x_2^{(N)}, \cdots, \Delta x_T^{(N)}, y_1^{(N)}, \Delta y_2^{(N)}, \cdots, \Delta y_T^{(N)}$ will be QM-coded.

### *C. Sprite Accretion and Coding*

*1) Definition of Sprite:* A *sprite* is a large static image composed from the pixels in an object visible through the entire scene. Consider the example of a background object in a sequence. Portions of this background may not be visible in certain frames because of foreground objects or camera motion (panning). However, if we collect all of the relevant pixels throughout the entire sequence, we can obtain a complete background *(sprite)*, which can be transmitted or stored once and then used to recreate portions of many different frames.

*2) Sprite Accretion:* The process of registering object pixels from different frames and then blending them into a single sprite is called *accretion*. This is similar to creating a photographic mosaic using traditional means [5], [6], [24]. Accreted sprites can be used to fill in portions of each frame which become exposed because of foreground or camera motion. Because of camera (or object) motion, the sprite may have to be warped differently to each frame in the sequence.

A sprite is represented and encoded in the same way as a regular object, i.e., as a collection of pixels with an alpha mask. To transmit the information necessary to warp the sprite into its correct position at each frame, we use a collection of *feature points*, or *trajectories*. These feature points do not have to correspond to any semantically meaningful features in the scene—they just have to be points whose positions in the sprite and frame are in correspondence. For example, for the planar perspective warps described below, the feature points will simply be the locations in the sprite corresponding to the four points in the object.

We use planar perspective transformations to describe the deformations between a sprite and the frames where it appears. This transformation is appropriate when the sprite corresponds to a single planar surface in the world, or when the sprite is a static background and the camera motion is a pure rotation around its optical center. In both cases, the geometrical relationship between pixels in different frames (and hence between the frames and the sprite) is represented as a planar perspective transformation [9]

$$x' = (ax + by + c)/(gx + hy + 1)$$
$$y' = (dx + ey + f)/(gx + hy + 1)$$

where $(x, y)$ and $(x', y')$ are the coordinates of the pixels in the current frame and the sprite, respectively, and $a$ through $h$ are perspective coefficients. The values for these coefficients are derived from the locations of the corners of each frame in the sprite (note that these locations may be outside the sprite and/or current object).

To compute the values for $a$ through $h$ for each frame, we use a global intensity-based registration algorithm proposed in [5]. This algorithm operates in a hierarchical (coarse-to-fine) way. At each level, the displaced frame differences are computed between the warped frame and the previous sprite. The motion parameters $a$ through $h$ are then updated using gradient descent on the total displaced frame difference (DFD) error. After several iterations, the algorithm converges, provided the initial motion was not too large at that level.

At all levels, a mask for that frame is used to indicate the locations of pixels for registration.

Once a given frame has been registered with the previous sprite (the sprite is originally initialized to be the object appearing the first time), it is then blended *(accreted)* with the previous sprite to construct the current sprite. A weighting function which decreases near the edges in order to mask out differences in overall intensity may be used.

There are two major advantages of using sprites. First, if an object only undergoes simple 2-D motion, sprite composition and warping is accurate enough to describe the movement of the object without running the local dense motion estimation/compensation as described below. This saves a large amount of encoding time and bits for coding of motion vectors and error signals. The most obvious example is the background layer in a video conferencing scene. In fact, obtaining the background sprite can be a very easy and fast operation, since often a complete background is given at the beginning. Second, information from a sprite can be used to represent newly exposed regions in each frame. Instead of attempting to do motion estimation in those areas, we use pixels from the background sprite because these newly exposed areas cannot be predicted from the previous frame. Furthermore, intraframe compressing these newly exposed areas for the whole image sequence (i.e., the master sprite) together is more efficient than processing them in each frame as error signals, i.e., MPEG-1 or MPEG-2 because spatial correlation can be exploited.

*3) Encoding and Decoding:* Sprite encoding is done using an intraframe (as opposed to error signal) still image compressor. In our scheme, a lattice wavelet coding algorithm [16] is employed for the sprite compression. Wavelet coding is preferred because it can avoid blocking artifacts. The warping parameters of the sprite to each frame are encoded using the trajectory coding technique described in Section III-B. In this case, four pairs of feature points are used for the frames where the sprite appears.

### *D. Local Affine Motion Estimation*

Motion estimation and compensation play very important roles in a video compression scheme [7], [8]. Since strong correlations and similarities exist between consecutive frames, the data rate can be dramatically reduced if temporal correlation is appropriately exploited. The motivation of our work is to provide a method that can use more complex motions than translation for motion compensation. These complex motions include rotation, magnification, and shear. In all of the video compression standards, such as MPEG-1, MPEG-2, and H.26X, only translation is used in motion estimation and compensation. This approximation is good enough for slow-motion video such as video conferencing, or translation-only frames. It is, however, not accurate enough for most of the video signals that contain more complex motions. High error signals, and therefore high data rates, are expected if only translation is used to model these complex motions.

Local affine transformation, as a simplified version of perspective transformation, can describe translation, rotation, magnification, and shear. Two processes are involved in our implementation for the computation of affine transform, namely, dense motion vector estimation and clustering to obtain affine coefficients. We use a modified block match algorithm to compute the dense motion vectors, and then get the affine coefficients by clustering from computed motion vectors.

*1) Fast Modified Block (Polygon) Matching:* Since we are dealing with object-based coding, boundary information should be taken into account in the motion estimation. Two particular issues are addressed: modified block (or polygon) matching algorithm and its fast implementation.

*a) Modified block (polygon) matching:* A mask which defines the region of the object is first input to the process for each object, assuming that this mask has been obtained. To compute the motion vector for one pixel, a block centered at this pixel is formed. The user specifies the block size. The closest block is then computed using a distortion criterion in the neighborhood of the next frame. For example, if sum of absolute difference (SAD) is used, we compute

$$\mathrm{SAD}_k = \Sigma\, ij(|r - r_{ij}| + |g - g_{ij}| + |b - b_{ij}|)* \\ \cdot (\mathrm{Alpha}_{\mathrm{original}}\,! =\ 0), \qquad 0 \le i < m, 0 \le j < n$$

for each $i$ and $j$. Here $m \times n$ is the block size specified by the user. The last term $(\mathrm{Alpha}_{\mathrm{original}}\,! =\ 0)$ indicates that only nontransparent original pixels are used in the computation, thus avoiding the problem of including the pixels belonging to a different object with different motion. This is equivalent to forming a polygon to do the matching. The motion vector is then defined as the difference between the location of the current pixel and the corresponding one in the reference frame with the smallest distortion $\mathrm{SAD}_k$.

*b) Fast block (polygon) matching:* Dense motion estimation (i.e., a motion vector for each pixel) is very slow to compute if exhaustive block (polygon) matching is applied. However, we can dramatically speed the whole process up by taking advantage of the memory of the data of the adjacent pixels. The blocks for the two horizontally adjacent pixels only differ by two columns of pixels, one incoming for the new pixel and one outgoing for the old pixel. Therefore, if we store the computation results of the previous pixel and only update the changes, we can greatly reduce the amount of complexity.

The procedure is as follows. Assume the block size is $n \times n$, the search range is $m \times m$, and the rectangle size of the bounding box in an object is $p \times q$.

1) To fully take advantage of the spatial, namely, horizontal and vertical, memory, we need to store ($m^2*$ (block + bound).width) error numbers, where (block + bound) is the union size of the block and bounding rectangle. Denote (block + bound).width as biWidth. We create an array for these numbers.
   - For the first row of the image, we generate $m^2$ difference numbers for each column.
   - Starting from the second row, we update the array by deleting the $m^2$ difference of the outgoing pixel and adding the $m^2$ difference of the incoming pixel in each column.
2) We can generate the $m^2$ difference numbers (assign an array for this) for each block to decide the motion vector of the corresponding pixel.
   - For the left-most pixel of each row, we sum up the difference of the $n$ columns for each one in the $m \times m$. So we generate $m^2$ difference numbers for this pixel and store this in the array. The motion vector is the difference of the location from the pixel to the one in the $m \times m$ pixels with the smallest value.

- From the second pixel of each row, we update each of the $m \times m$ numbers by adding the incoming column and deleting the outgoing column. Then we find the motion vector using the same criterion.

The complexity is proportional to the image size $p \times q$ times the search range $m \times m$. It is independent of the block size. So we reduce the arithmetic complexity by a factor of $n^2$, ignoring the bigger memory buffer required to do the computation and the memory swapping time. Compared with the brute force approach, which can be huge if $n$ is large, the accuracy remains the same. In other words, the results are exactly the same as exhaustive block (polygon) matching. With the fast block matching algorithm, dense motion estimation becomes feasible.

*2) Affine Transform Clustering:* Once the dense motion vectors are obtained for an object, we can generate affine transforms for that object. The computed affine transform is the best approximation of the dense motion vectors in the least-square error sense. The affine transformation, as a simplified version of perspective transformation with $g = h = 0$, is defined as follows:

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \end{aligned}$$

where $(x, y)$ and $(x', y')$ are the coordinates of the pixels in the current and reference objects, respectively, and $a$ through $f$ are the affine coefficients. Since we need six parameters to describe an affine transform, we need at least three pairs of $(x, y)$ and $(x', y')$ to compute an affine transform.

It is usually not sufficient to use one affine transform to describe the motion of a whole object. We use locally affine transformations. We divide an object into squares of the same shape, e.g., 32 $\times$ 32 blocks. Given the motion vector of each pixel (e.g., computed from polygon matching), except for some pixels with low confidence, we have $n$ pairs of corresponding coordinates, where $n$ is the number of pixels with a high-confidence motion vector. So we have $(x_i, y_i) \rightarrow (x'_i, y'_i), i = 0, \cdots, n-1$. Therefore, we define $n$ sets of the equations for affine transformations

$$\begin{aligned} x'_i &= ax_i + by_i + c \\ y'_i &= dx_i + ey_i + f. \end{aligned}$$

We then compute the six affine coefficients from these $n$ equations. This is an over-determined system, meaning that the number of equations is greater than the number of unknowns. For fast computation, we apply *singular value decomposition* (SVD) [13] to obtain these six coefficients. The computed affine transformation is the best approximation of the dense motion vectors in the least-square error sense.

Since we need six coefficients of overhead to describe an affine transform for each square, which is three times more than the translation-only motion approaches, we need to be careful about choosing the block size. If the block size is too small, the amount of data for encoding these six coefficients will add significantly to the entire bitstream. This loses the advantage of applying affine motion compensation. On the other hand, if the block size is too large, one affine transform may not be accurate enough to describe the motion of the entire block, which will result in high error signals, and thus high data rate, for this block.

In our preferred mode, we use the size of 32 $\times$ 32 for each block, instead of 16 $\times$ 16 used in MPEG-1 and MPEG-2. Since the overhead here is three times as much as in MPEG-1 or MPEG-2, we enlarge the block size to be four times of MPEG-1 so that the average overhead is actually smaller. However, since more accurate motion estimation is performed, the error signal is not expected to be larger than that of MPEG-2 even when the block size is four times larger.

*3) Pixel Interpolation:* Since the warped pixel coordinates $(x', y')$ are usually not integers, some interpolation technique has to be applied to yield the pixel values. In the current implementation, the simple bilinear interpolation is used because of its low computational complexity and reasonable quality.

*4) I, B, and P Objects:* I (intra), B (bi-direction), and P (prediction) object modes are used in our video coding system. Similar to I, B, and P frames in MPEG1 [1] and MPEG2 [2], the I-object mode is to code the original signals, P-object is to do motion estimation and compensation using one object as the reference in some previous frame, and the B-object mode is to do motion estimation and compensation bidirectionally with two reference objects, one in some previous frame and one in some later frame. Both the two reference objects in the B-object mode have to be either I- or P-objects. In our design, the user can specify the number of P-objects between two I-objects, and the number of B-objects between two I-objects, an I- and a P-objects, or two P-objects.

There are several advantages of B-objects. Since the motion estimation for B-objects is done bidirectionally, the estimated objects are expected to be more accurate than P-objects, especially for the newly exposed regions since one following frame is referenced. Also, temporal scalability can be easily achieved using B-object since they can be dropped at the decoder side without affecting the following frames. Furthermore, since B-objects are not used as reference objects, bit allocation to the error signals can be flexible depending on the bit rate and the desired image quality. Usually, the amount of bits allocated to B-objects is much smaller than that of I- and P-objects to achieve high compression.

One drawback of using B-objects is that the amount of bits allocated to P-objects is high. This is because that frame distance of the current and the reference objects is large, thus the motion estimation is more difficult. This results in higher error signals and hence more bits for coding the error signals.

*5) Unrestricted Motion Estimation:* An unrestricted motion estimation mode is designed in our video coding system. The technique is to improve the motion estimation techniques, especially for object-based coding schemes. It is known that it is difficult to do motion estimation for the newly exposed regions, especially in P-objects, since the reference object does not contain any information for estimating the motion of these parts. Therefore, the error signals in these regions are high, thus causing high bitrate.

One way to alleviate this problem is to provide the option of coding the original signals instead of the error signals if the variance of the original is smaller than that of the error signals. It is likely that the variance of the error signals is larger than that of the original if the motion estimation is inaccurate.

In this technique, the error signal is generated by padding the reference signals, applying motion compensation, and taking the difference between the original and the estimated signals. This takes advantage of the spatial, in addition to temporal, correlation when motion compensation is applied. Therefore, for the newly exposed regions, even if the temporal estimation cannot be used, the error signals might still be low because of the spatial resemblance between pixels.

Also, because of the limitations in restricted motion model, the motion estimation for the block that is partly newly exposed is forced to locate at a wrong position because the motion vector cannot be out of bound in the reference object. The error signals in these regions are hence high. The original signals therefore tend to be coded. With this unrestricted motion model, it is guaranteed that part of the motion estimation is done accurately, even though the error signals for the padded region may be high. So the error signals for the whole region are likely to be small.

The procedure for this unrestricted motion estimation is described as follows.

- Expand the bounding rectangle of the reference object by some amount of pixels in all four directions. The amount of expansion should be large enough, but not too big, to save memory size and computations. Currently, we expand the image in each direction by the motion-compensation block size
- Pad the transparent pixels of the reference object using the repetitive padding.[1] Use the padded image as the new reference object.
- Apply local motion estimation and compensation described in Section III-D.

The computational complexity for this scheme is low. Only expansion and padding need to be done in addition to motion estimation and compensation.

## E. Error Signal Suppression

Error signals are generated by subtracting the motion-compensated signals from the original signals. The error signals usually cost the largest amount of bits in the entire bitstream, typically from 80 to 95%. Therefore, it is important to suppress the error signals in order to reduce the bitrate without sacrificing the visual quality.

A simple, fast, and efficient error signal suppression method is designed. This method is composed of two steps: a thresholding and a nonlinear filtering of the thresholded signal. The goal of the thresholding mechanism is to get rid of the random noise. Because of the imperfection of the original image sources, random noise will still be left over in error signals even if the motion estimation is accurate. Thresholding can reduce this effect. Afterwards, a 3-by-3 cross-shaped median filter (see Fig. 3) is applied. This step is to eliminate the salt-and-pepper like signals that do not affect the visual quality. It is well known that motion estimation is much more

[1] See Section III-F for detailed descriptions of repetitive padding.

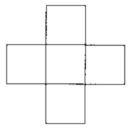


Fig. 3. Cross-shape median filter.

difficult at the high contrast regions. A small inaccuracy of motion estimation will result in large impulsive error signals. These impulsive signals are very difficult to code using an error signal coder since this creates a lot of high frequency components in the interframe compressor. Getting rid of these signals actually does not influence the visual quality too much, while reducing the bitrate significantly.

The advantage of such a technique is that the impulsive noise can be removed while the other original signals are faithfully preserved. Other nonlinear filtering techniques such as normal separable or nonseparable median filters and morphological filters may destroy more meaningful signals. The simulations show that the bitrate can be reduced by 10 to 20% without loss of image visual quality.

## F. Image Padding Technique

The supports for image regions of each object have to be made rectangular in order for subband or transform coding to be applied, except shape adaptive approaches of transform coding such as shape adaptive discrete cosine transform (DCT). An appropriate padding scheme should be employed to the transparent pixels of an object to reduce the data rate required for this rectangular support. In other words, we should replace the color values of the transparent pixels by some numbers so that it benefits the texture coding the most. Since contour information, i.e., the mask, is already sent, we can change the values of the transparent pixels arbitrarily without affecting the original data. In the following, we discuss a novel image padding technique, i.e., repetitive padding, followed by a summary of the performances of repetitive padding and comparisons with another two well-known padding techniques.

The repetitive padding process consists of five steps.

1) Consider each undefined pixel outside the object boundary a zero pixel.
2) Scan each horizontal line of the original image region. Each scan line is possibly composed of two kinds of line segments: zero segments that have all zero pixels within each segment and nonzero segments that have all nonzero pixels within each segment. If there are no nonzero segments, do nothing. Otherwise, there are two situations for a particular zero segment: it can be positioned either between an end point of the scan line and the end point of a nonzero segment, or, between the end points of two different nonzero segments. In the first case, fill all of the pixels in the zero segments with the pixel value of the end point of the nonzero segment. In the second case, fill all of the pixels in the zero segments with the averaged pixel value of the two end points.
3) Scan each vertical line of the original image and perform the identical procedure as described in 1) to each vertical

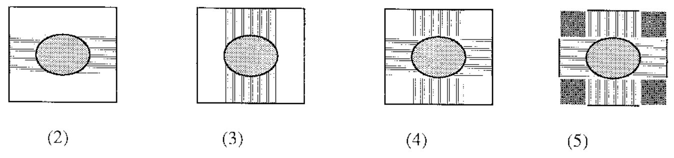


Fig. 4. Illustration of some steps of repetitive padding.

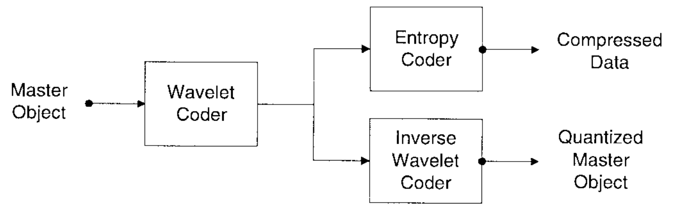


Fig. 5. Sprite encoder.

scan line.

4) If a zero pixel can be filled in by both 2) and 3), the final value takes the average of the two possible values.
5) Consider the rest of zero pixels. For any one of them, scan horizontally to find the first horizontal nonzero pixel, and scan vertically to find the first vertical nonzero pixel. Replace the zero pixel by the average of the first horizontal and vertical nonzero pixels.

Fig. 4 illustrates the outcome of each of the steps described above.

Many padding techniques have been proposed, e.g., constant-value extension (a special case is called "zero-stuffing") and mirroring [15]. Based on our experience, for transform coding techniques such as DCT, repetitive padding technique provides better results than constant-valued extension and mirroring methods for intratype image coding. For the displaced object difference (DOD) coding, zero-stuffing technique often outperforms mirroring and repetitive padding, not to mention its simplicity. Therefore, the zero-stuffing technique has been applied in our DOD compression while the repetitive padding method is used when the object appears at the first time.

## IV. Encoder

This section will describe the complete encoding process, which in essence is a combination of the tools described in Section III.

### *A. Pre-Generated Data*

In our current implementation, the following data have to be generated before the error feedback process if they exist or the option is turned on:

- masks;
- trajectories;
- quantized sprites (master objects), shown in Fig. 5.

In Fig. 5, a lattice wavelet coder [16] represents a wavelet transform followed by a quantizer. The inverse wavelet coder is an inverse quantizer followed by an inverse wavelet transform. Although the processes of generating these data are off-line, they can be changed to on-line without too much complication. For example, for sprite accretion and generation, instead of quantizing and transmitting the whole sprite together, we can determine the newly exposed regions in one frame and code these regions as they are determined. However, this approach leads to less efficient compression of the sprite. Therefore, the real-time processing ability is only obtained at a cost of compression efficiency.

### *B. Error Feedback Loop*

The block diagram of the error feedback loop per object is shown in Fig. 6. Note that in Fig. 6, the coordinate transform block actually includes two parts: 1) the perspective transform defined by the trajectories for sprite warping and 2) the affine transform defined by the dense motion estimation for estimating the object of the current frame from some reference object. For sprite warping, the coordinate transform is a global perspective transform. For object warping (dense motion compensation), the coordinate transform is a local affine transform. For the clarity of the figure, this distinction is not shown in the block diagram. Also, this diagram only shows a pure object-based coding approach. Several other modes are also supported by the scheme.

- Frame-based coding scheme, such as MPEG-1, MPEG-2, and H.26X. This is actually a special case of this object-based scheme since we can set the number of objects to be one and define the object to be the whole frame. This mode is to support backward compatibility to the existing standards and handle scenes that are not suited for object-based coding.
- Per-frame error signal coding and compression. In this mode, the error signals for all of the objects in the frame are composited and then coded together. The whole quantized previous frame, instead of an individual object, will be the reference for the current object. This mode is supported mainly for enhanced compression efficiency when the functionalities of pure object-based coding are not required. Although it is not easy to verify the above statement rigorously, most obviously per-frame coding does not have to code the padded pixels that exist in per-object coding and thus at least it has fewer pixels to code. However, the advantage of independent per-object coding is that we can intelligently allocate various amounts of bits to different objects depending on the resolution requirement for each object. Many bits could be saved by using this method.

The detailed encoding process is described in the following two sections:

*1) The Frame When the Object First Appears (Refer to Fig. 7):* Note that in general, an object can appear at any time.

- Load and quantize the masks. Put the coded data to the bitstream.
- For each object, check the existence of the sprite.
- If YES, load the quantized sprite. Warp the quantized sprite perspectively to the first frame based on the transform defined by trajectories of four feature points. Apply the quantized mask to the warped quantized sprite. Fig. 7

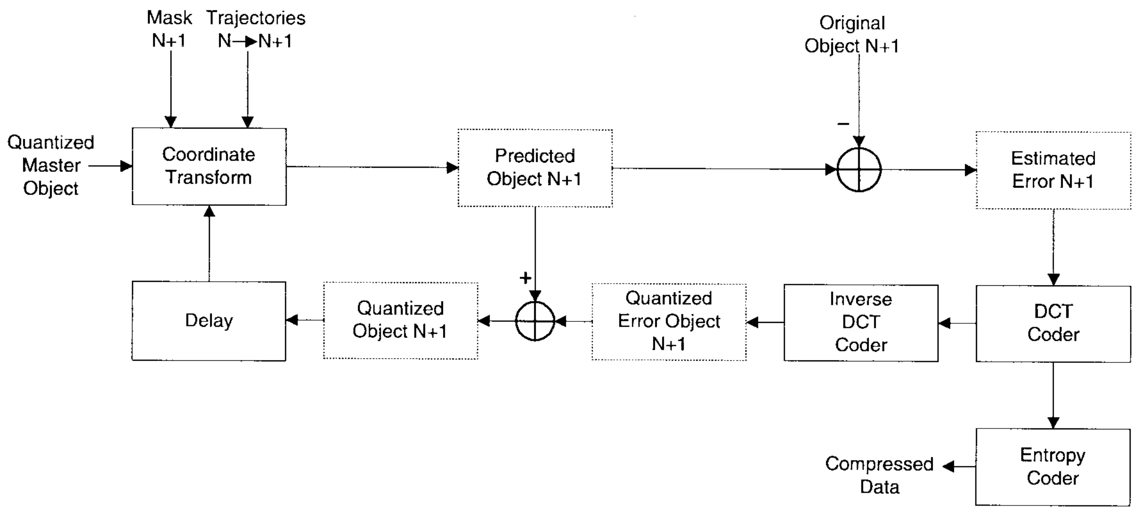


Fig. 6. Encoder error feedback loop.

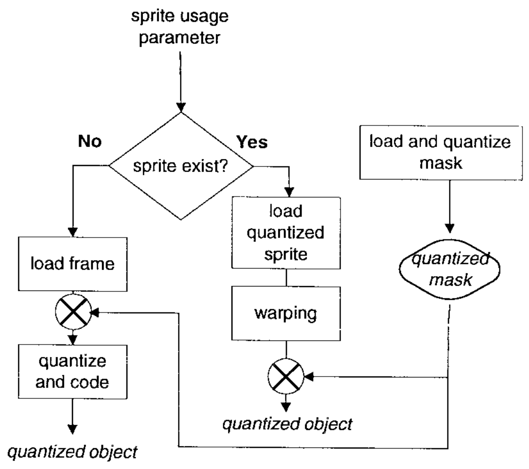


Fig. 7. Encoding for the first frame.

represents the procedure to apply a mask to an object. This is the quantized object for the first frame.

- If NO, load the original object if it is available, or load the original frame, then apply the quantized mask to this frame. Quantize and code this object. Put this coded data to the bitstream.

*2) Subsequent Frames (Refer to Fig. 8):* Here the quantized object warping includes motion estimation and motion compensation. For each object we have the following.

- Load and quantize the mask. Put the coded data to the bitstream.
- Check the *sprite usage* encoding parameter.
    - If it is zero, this means that no sprite is used for this object. Check the codeword for affine quantization. If it is three, only pure block match is applied to generate the estimated object. Otherwise, perform dense motion estimation and affine clustering for each block in the object. Quantize and code the affine coefficients using the trajectories of three feature points. Put the coded data to the bitstream. The estimated object is obtained by warping the region in the reference object to the current block using the quantized affine coefficients. Check the *per-object error signal compression* encoding parameter.
    - If it is ON, compute the error signal for this object by subtracting the estimated object from the original object. Quantize and code this error signal using DCT. Put this data to the bitstream.

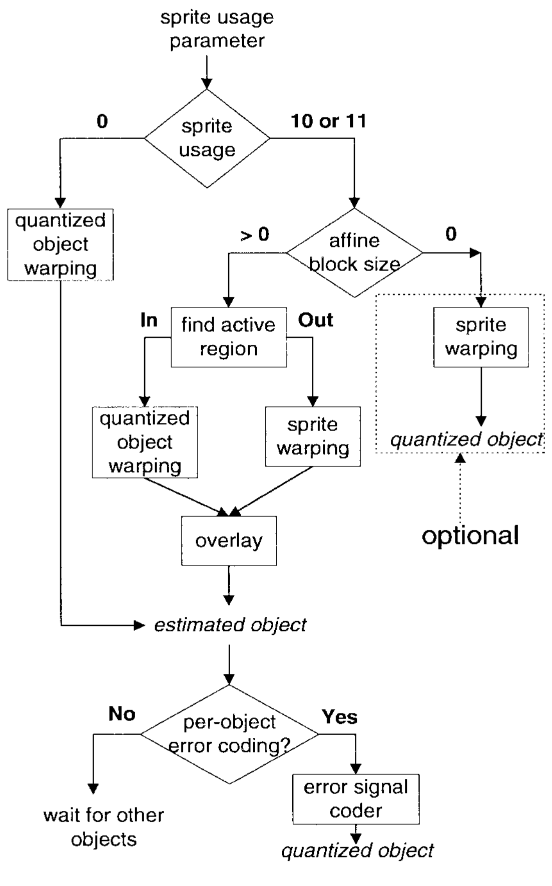


Fig. 8. Encoder for the subsequent frames.

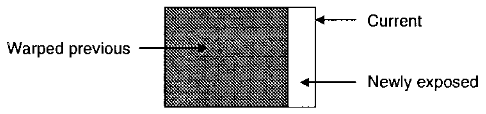


Fig. 9. Illustration of the mask operation for detecting newly exposed region.

- If it is OFF, continue to the next object.

Note that motion estimation and compensation should be done bidirectionally with two reference objects if it is a B-object.

- If it is 10 or 11, a sprite is used. Ten means that a stationary sprite is used (no sprite warping is needed), and 11 means that a moving sprite is used. Check the affine block size for this object.
    - If it is zero, continue to the next object. However, the encoder may choose to warp the sprite to the current frame using the trajectory information to obtain the quantized object for the current frame. This operation is not necessary for bitstream generation.
    - If it is greater than zero, it implies that motion compensation also needs to be applied.
        - The mask of the previous frame is warped to the current frame first using the trajectories.
        - Subtract the warped previous mask from the current mask. This results in the newly exposed region in the current frame. This is illustrated in Fig. 9. The sprite is warped only to this newly exposed region using the trajectories.
        - The intersection of the current mask and the warped previous mask is the region where local motion estimation and compensation is applied. Use the same method as in the case of *sprite usage* being zero to generate the estimated object and/or quantized error signals as well as the quantized object.

- If the *per-object error signal compression* flag is FALSE,
    - Combine all of the estimated objects by compositing them together. This yields the estimated frame.
    - Subtract the estimated frame from the original frame. This gives the error signals for the current frame.
    - Quantize and code the error signals using DCT. Put the compressed data to the bitstream.
    - Add the quantized error signals back to the estimated frame to get the quantized frame.

### C. Encoder Output Bitstream

The output bitstream of the encoder includes the following: overhead, sprites, alpha channel information, trajectories for global sprite warping and local affine motion compensation, and error signals. Not all of the data need to be present in the bitstream. This depends on the objects and the coding modes that the user specifies.

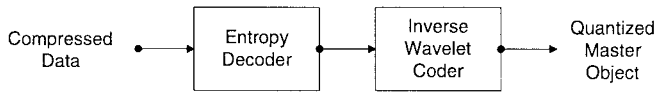


Fig. 10. Master sprite decoder.

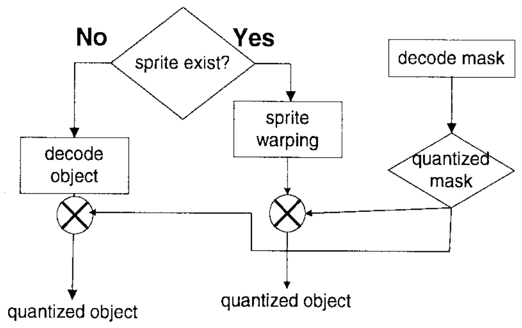


Fig. 12. Decoder for the first frame.

## V. Decoder

The decoder generates the reconstructed objects and/or frames from the bitstream. Although it is basically a subset of the encoding process since the encoder needs to generate the reconstructed objects for the reference of the next frame, we still describe this part separately because MPEG4 requires a complete video coding system which includes both encoding and decoding syntax.

### A. Overhead and Sprite Decoding

- Decode the overhead to obtain all of the modes specified in the encoder.
- Decode the sprites using a wavelet decoder, as shown in Fig. 10, if they are available. The QM-coder is used for the entropy decoder. Here the inverse wavelet coder includes an inverse quantizer followed by an inverse wavelet transform.
- Decode the trajectories if the *sprite usage* parameter is 11.

### B. Decoder Error Feedback Loop

The diagram of the decoder error feedback loop is shown in Fig. 11.

### C. The Frame When the Object First Appears (Refer to Fig. 12)

This can be detected from the bitstream by

1) if the *fixed number of objects* flag is TRUE, then this is always the first frame in the sequence;
2) if the flag is FALSE, then this has to be detected by the *object existence* flag in each frame. If this flag is zero in the previous frame and one in the current frame, this indicates that this object newly appears in this frame.

After detecting this, for each object, the following operations are performed.

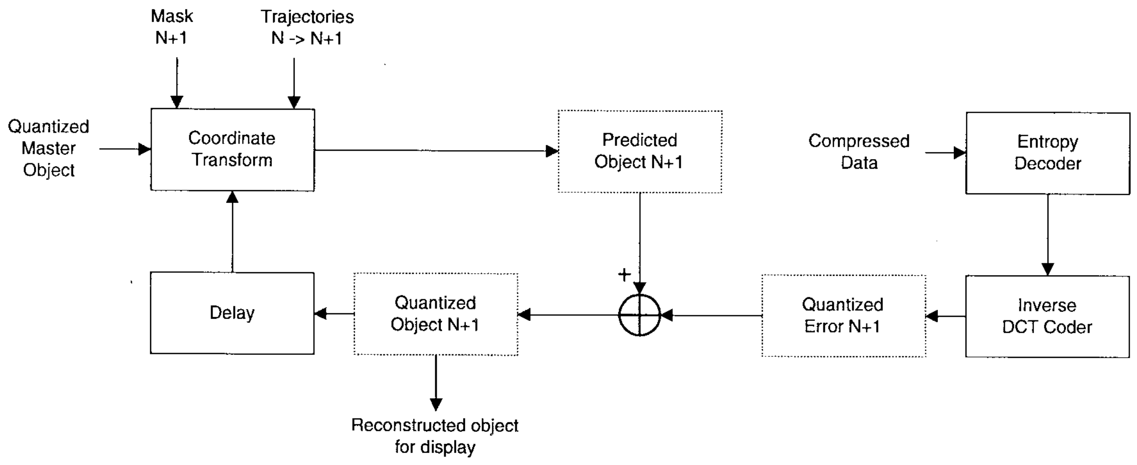


Fig. 11. Decoder error feedback loop.

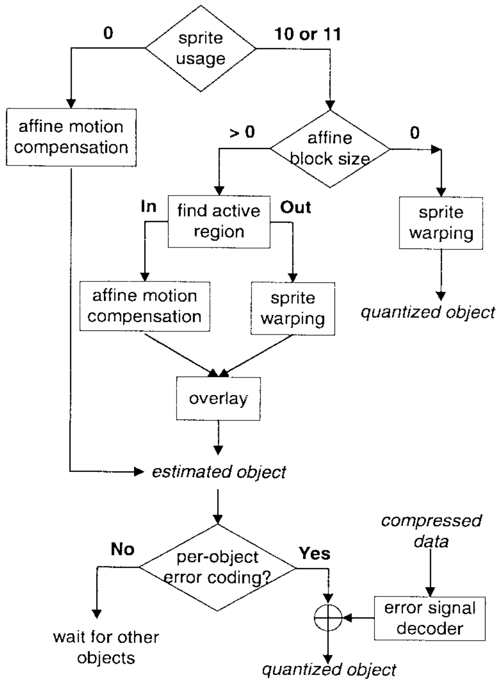


Fig. 13. Decoder for the subsequent frames.

- Decode the mask.
- If the sprite exists, check the *sprite usage* flag.
    - If it is ten, apply the mask to the quantized sprite. This is the quantized object.
    - If it is 11, warp the sprite to the first frame using the trajectory. Apply the mask to the warped sprite. This is the quantized object for the first frame.
- Otherwise, decode the intraframe coded data, then apply the mask to this decoded object. This yields the quantized object for the first frame.

*1) Subsequent Frames (Refer to Fig. 13):*

- Decode the mask.
- Check the *sprite usage* flag for this object.
    - If it is zero, this means that no sprite is used for this object. Decode the dense affine coefficients. Generate the estimated object using these affine coefficients and the reference object (may be the quantized object of the previous frame). Check the *per-object error signal compression* flag.
        - If it is one, decode the error signal for this object using a DCT decoder. Add this error signal to the estimated object. This gives the quantized object.
        - If it is zero, continue to the next object.
    - If it is 10 or 11, a sprite is used. Check the affine block size for this object.
        - If it is zero, warp the sprite to the current frame using the trajectories information. This is the quantized object.
        - If it is greater than zero, it implies that this sprite has to be combined with the dense motion compensation.
            - The mask of the previous frame is warped to the current frame first using the trajectories.
            - Subtract the warped previous mask from the current mask. The result is the newly exposed region in the current frame. Warp the sprite to this newly exposed region using the trajectories information.
            - The intersection of the current mask and the warped previous mask is the region where dense motion estimation/compensation has been applied at the encoder. The dense affine coefficients are decoded first. The estimated signal for this active region is generated by warping (backward) the reference object using the decoded affine information.

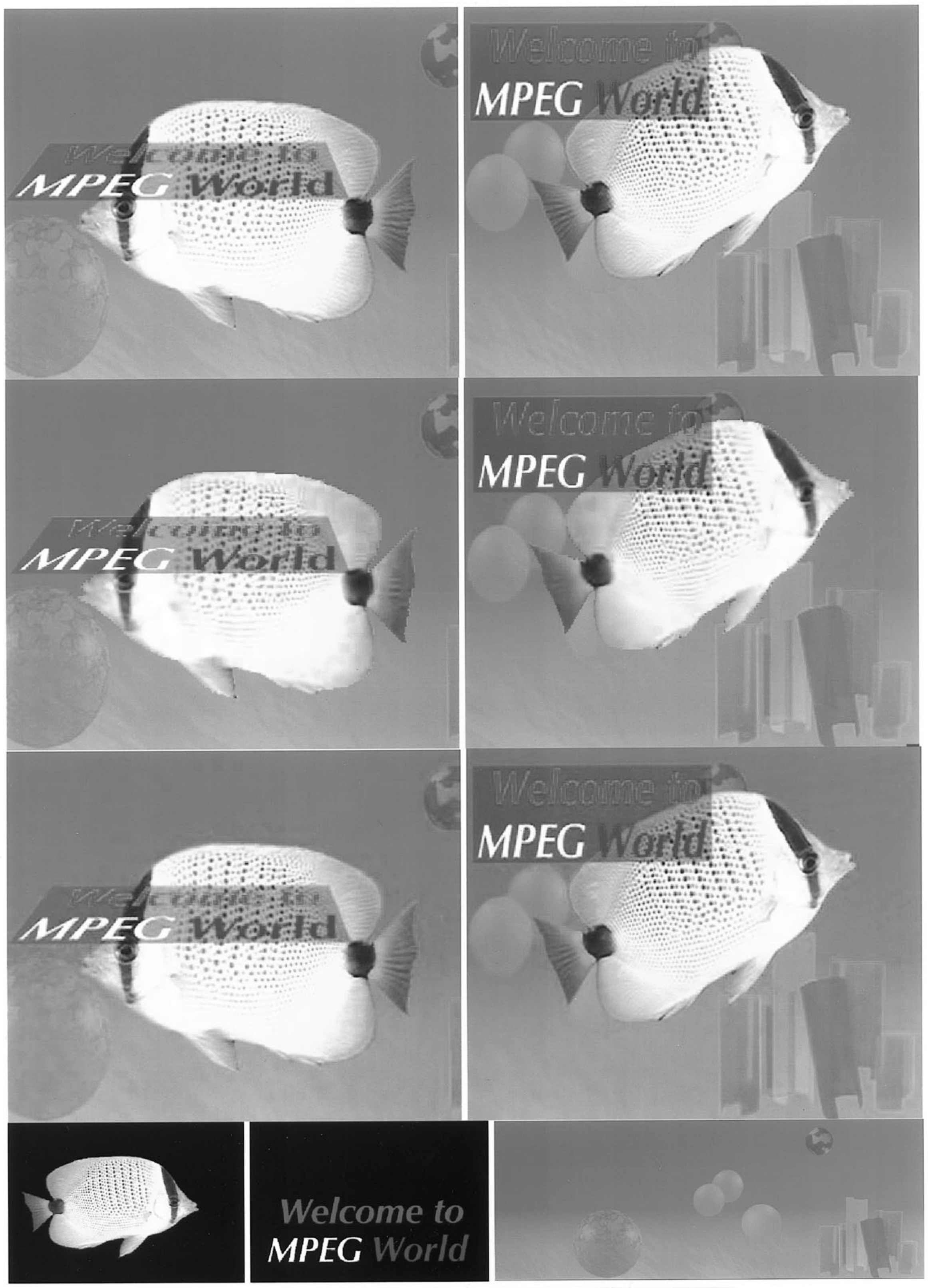


Fig. 14. Experimental results on “Fish” sequence.

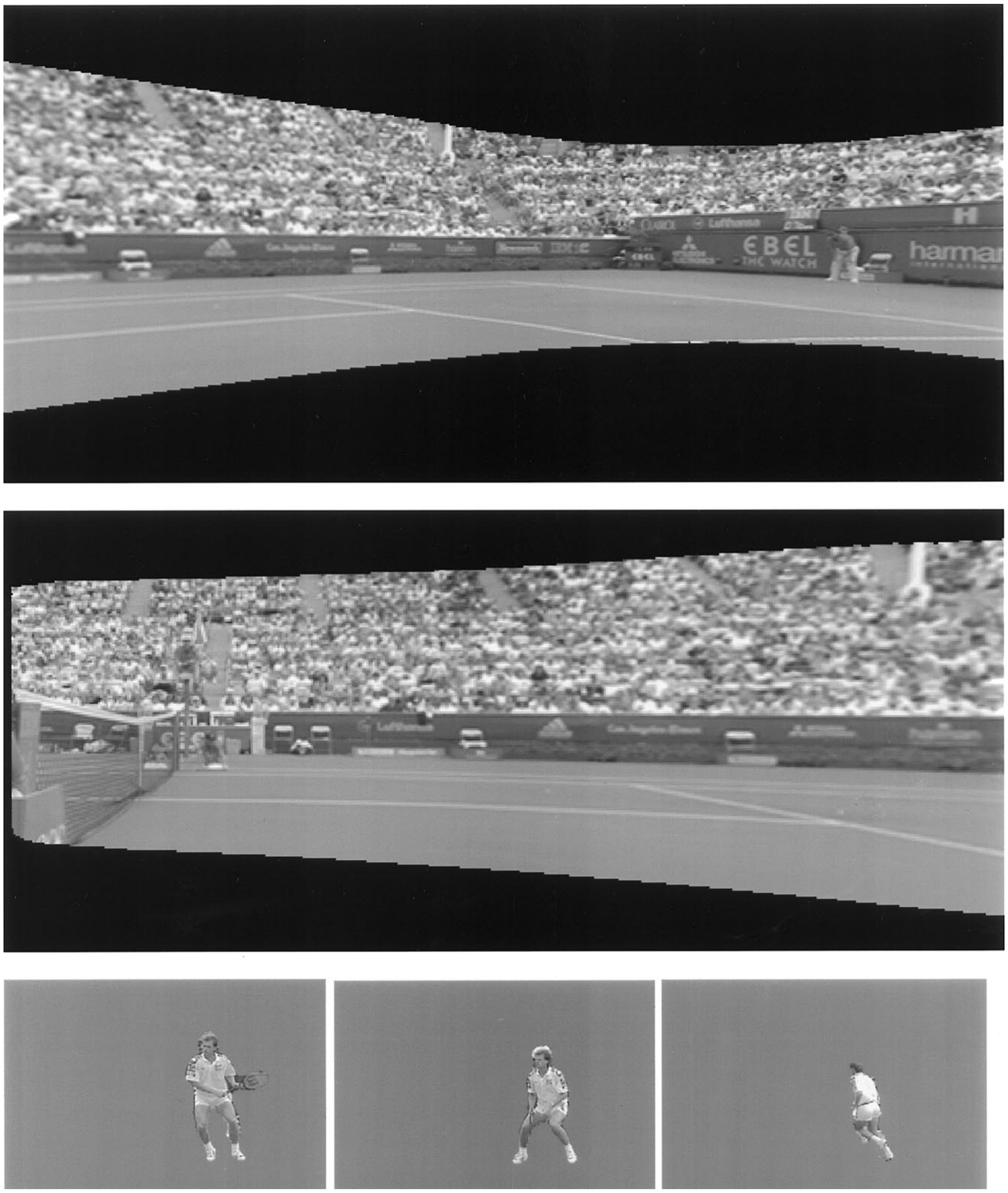


Fig. 15. Experimental results for "Stephan" sequence.

- Overlay the two regions together to yield the estimated object. Check the per-object error signal compression flag.
    - If the per-object error signal compression flag is TRUE, decode the error signal for this object. Add this error signal to the estimated object. This yields the quantized object for the current frame.
    - If it is FALSE, continue to the next object.
- If the *per-object error signal compression* flag is FALSE
    - combine all of the estimated objects by overlaying the objects together. This yields the estimated frame;
    - decode the error signal for this frame using a DCT decoder;
    - add the quantized error signals back to the estimated frame to get the quantized frame.

## VI. Experimental Results

In this section, we will show some experimental results using our video coding system. Two sequences, namely "Fish"

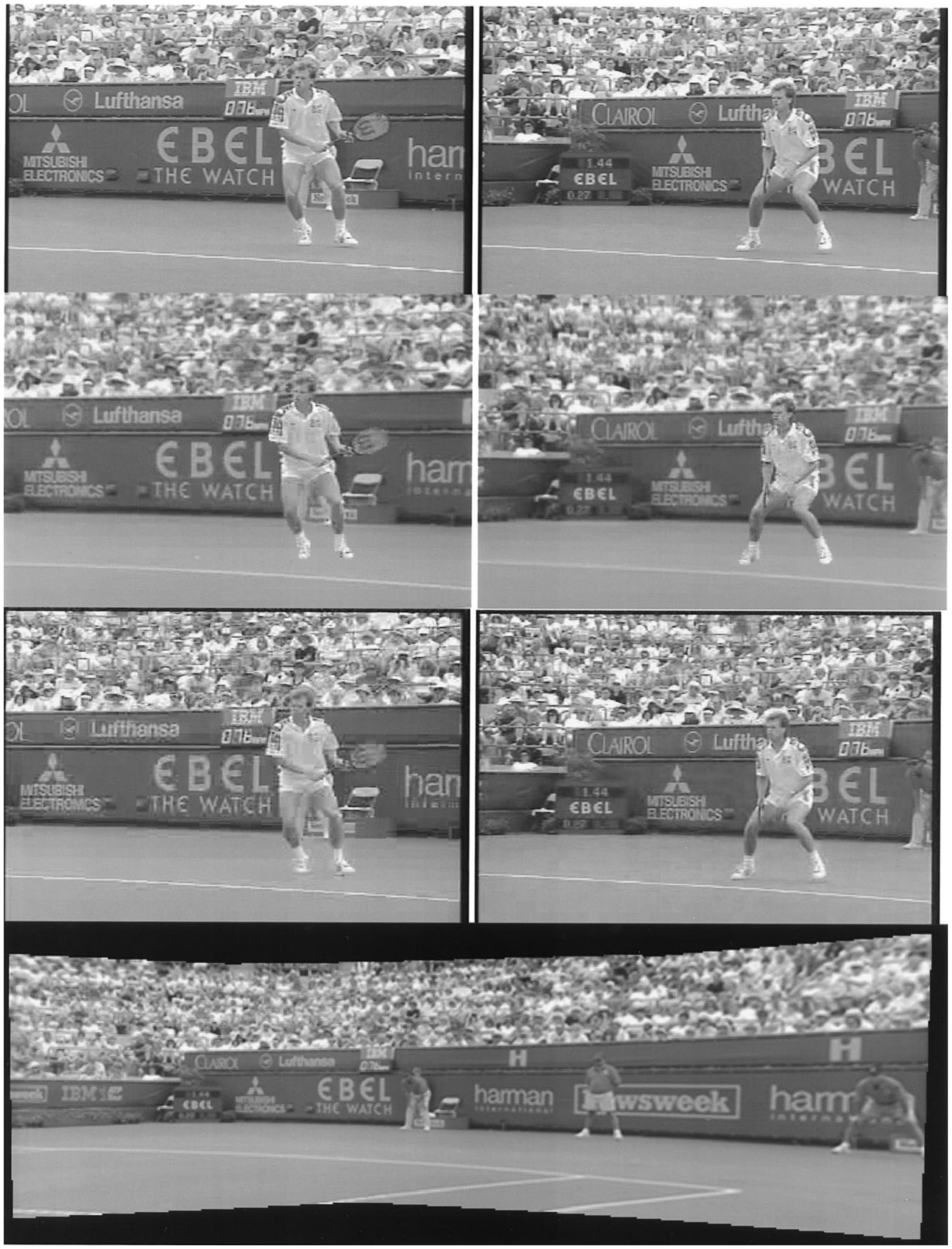


Fig. 15. (*Continued.*) Experimental results for "Stephan" sequence.

(hybrid sequence) with CIF format (352 × 288) and "Stefan" (nature sequence) with SIF format (360 × 240), are tested. In both cases, 10 s of video clips are processed. We will highlight our improvement of coding efficiency as well as new functionalities. The comparison is carried out with a traditional frame-based video codec like H.263. The common testing conditions are the following.

- Frame rate: 15 frame/s.
- DCT quantization step: 20.

In Fig. 14, the first row shows two original frames in "Fish" sequence. The second row presents the corresponding decoded frames using our codec at 99 Kb/s. The third row provides the decoded result by H.263 in the same visual quality but at a much higher bitrate: 320 Kb/s. The fourth row illustrates each layering object involved in the sequence.

In Fig. 15, a nature sequence “Stefan” is tested. The first row shows two original frames. The second row demonstrates the decoded frames at 87 Kb/s. The third row provides the decoded result by H.263 in the same visual quality but at a much higher bitrate: 520 Kb/s. The following three images represent the extracted sprites which are used to represent the background without update. The reason to have three separate sprites for background is purely caused by buffer limitation. Part of the masks are provided by MPEG4 testing group and some of them are obtained using a semi-automatic segmentation tool developed in Microsoft Corporation. The last row shows that we can achieve object scalability very easily using our codec.

## VII. Conclusions

A video object coding system using layered representation has been presented in this paper. It achieves improved coding efficiency compared to the traditional nonobject-based video coding system. Moreover, it also supports additional functionalities such as object scalability. Due to its promising performance, many parts of this layered video object coding scheme have already been selected as elementary components of the MPEG4 Verification Model (VM), e.g., the overall object-based architecture, the polygon matching, the repetitive padding, and object-based motion estimation and compensation. Moreover, sprite warping and affine motion compensation are anticipated to be adopted in the future version of VM. We believe that this new technology could be integrated into additional multimedia storage and real time video applications.

## Acknowledgment

The authors want to thank the detailed comments provided by two anonymous reviewers. These comments were very useful to the authors in improving the quality of this paper.

## References

[1] ISO/IEC 11172-2 MPEG1, “Information technology—Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s-video,” Geneva, 1993.

[2] ISO/IEC 13818-2 MPEG2, “Generic coding of moving pictures and associated audio,” Nov 1993.

[3] ISO/IEC JTC1/SC29/WG11/N998, “MPEG4 proposal package description,” Tokyo, July 1995.

[4] J. Y. A. Wang and E. H. Adelson, “Representing moving images with layers,” *IEEE Trans. Image Processing*, vol. 3, pp. 625-638, Sept. 1994.

[5] R. Szeliski, “Image mosaicing for tele-reality applications,” in *IEEE Workshop on Applications of Computer Vision (WACV’94)*, IEEE Computer Society, Dec. 1994, pp. 44–53.

[6] M. Irani, S. Hsu, and P. Ananda, “Video compression using mosaic representations,” *EUROSIP Signal Processing: Image Commun.*, vol. 7, nos. 4–6, pp. 529–552, Nov. 1995.

[7] H. G. Musmann, P. Pirsch, and H. J. Grallert, “Advances in picture coding,” *Proc. IEEE*, vol. 73, pp. 523–548, Apr. 1985.

[8] F. Dufaux and F. Moscheni, “Motion estimation techniques for digital TV: a review and a new contribution,” *Proc. IEEE*, vol. 83, pp. 858–879, June 1995.

[9] R. Y. Tsai and T. S. Huang, “Estimation three-dimensional motion parameters of a rigid planar patch,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-29, pp. 1147–1152, Dec. 1981.

[10] H. Freeman, “Computer processing of line drawing images,” *Comput. Survey 6*, pp. 57–98, Mar. 1974.

[11] M. Eden and M. Kocher, “On the performance of a contour coding algorithm in the context of image coding part I: contour segment coding,” *EUROSIP Signal Processing*, vol. 8, pp. 381–386, 1985.

[12] C. Gu and M. Kunt, “Contour simplification and motion compensated coding,” *EUROSIP Signal Processing: Image Commun.*, vol. 7, nos. 4–6, pp. 279–296, Nov. 1995.

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*. Cambridge, MA: Cambridge, 1992.

[14] International Telecommunication Union Recommendation H.263: Video Coding for Narrow Telecommunication Channels.

[15] S. F. Chang and D. G. Messerschmitt, “Transform coding of arbitrarily-shaped image segments,” *ACM Multimedia*, p. 8388, June 1993.

[16] P. T. R. Vaidyanathan, *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1993.

[17] W. B. Pennebaker, J. L. Mitchell, G. L. Langdon, and R. B. Arps, “An overview of the basic principles of the Q-coder adaptive binary arithmetical coder,” *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 717–726, Nov. 1988.

[18] M. Kunt, A. Ikonomopoulos, and M. Kocher, “Second generation image coding techniques,” *Proc. IEEE*, vol. 73, pp. 549–575, Apr. 1985.

[19] H. G. Musmann, M. Hotter, and J. Ostermann, “Object-based analysis-synthesis coding of moving images,” *EUROSIP Signal Processing: Image Commun.*, vol. 1, no. 2, pp. 117–138, Oct. 1989.

[20] ISO/IEC JTC1/SC29/WG11, “MPEG-4 Video Verification Model Version 1.1,” Feb. 12, 1996.

[21] C. Gu, T. Ebrahimi, and M. Kunt, “Morphological moving object segmentation and tracking for content-based video coding,” in *Int. Symp. Multimedia Communications and Video Coding*, NY, Oct, 1995. Plenum Press, pp. 233–240.

[22] T. Kaneko and M. Okudaira, “Encoding of arbitrary curves based on the chain code representation,” *IEEE Trans. Commun.*, vol. COM-33, no. 7, July 1985.

[23] C. Gu, “Multivalued morphology and segmentation-based coding,” Ph.D. dissertation, Swiss Federal Institute of Technology at Lausanne, Dec. 1995, Available: http://ltswww.epfl.ch/Staff/gu.html.

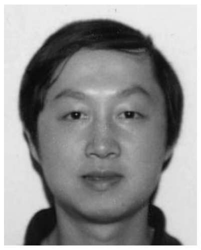
[24] H. S. Sawhney, S. Ayer, and M. Gorkani, “Model-based 2D&3D dominant motion estimation for mosaicing and video representation,” presented at *ICCV’95*, Cambridge MA, June 1995.

**Ming-Chieh Lee** (S’90–M’95) was born in Taiwan. He received the B.S. degree in electrical engineering from the National Taiwan University, Taiwan, in 1988 and M.S. and Ph.D. degrees in electrical engineering from California Institute of Technology, Pasadena, in 1991 and 1993, respectively. His Ph.D. research topic was on still and moving image compression using multiscale techniques.

During January 1993 to December 1993, he was with the Jet Propulsion Laboratory as a Member of Technical Staff and was working on multiresolution image transmission and enhancement. In December 1993, he joined the Advanced Video Compression group of Microsoft Corporation, Redmond, WA, as a Software Design Engineer. Since then, he has been working on object-based video compression. He is currently the development lead of this project.

**Wei-ge Chen** (S’92–M’95) received the B.S. degree from Beijing University, Beijing, China, in 1989 and the M.S. degree from the University of Virginia, Charlottesville, in 1992, both in biophysics. He received the Ph.D. degree from the University of Virginia, Charlottesville, in 1995 in electrical engineering.

Since 1995, he has been with Microsoft Corporation, Redmond, WA, where he works on the development of advanced video compression technology. His research interests include image/video processing, analysis, and compression.



**Chih-lung Bruce Lin** (S'93–M'95) received the B.S. degree from National Taiwan University in 1988 and the M.S. and Ph.D. degrees from the University of Maryland, College Park, in 1994 and 1996, respectively, all in computer science.

He was a Research Assistant at the Center for Automatic Research at University of Maryland from 1992 to 1995. Currently, he is with Microsoft, Redmond, WA, working on video compression. His research interests also include image/video processing, object recognition, and computer graphics.

**Chuang Gu** received the B.S. and M.S. degrees in computer science from the Fudan University, Shanghai, China, in 1986 and 1989, respectively. He earned the Ph.D. degree in electrical engineering from Swiss Federal Institute of Technology at Lausanne (EPFL) in 1995.

From 1989 to 1990 he worked in the Computer Aided Design Center of Fudan University as an Assistant Director for the management of various projects in image processing and computer graphics. From 1991 to 1992 he was an Associate Research Fellow of European Organization for Nuclear Research (CERN). He was in charge of several real-time network control and communication systems. In 1992, he joined the Signal Processing Laboratory in Swiss Federal Institute of Technology at Lausanne (EPFL) as a Research Fellow. He supervised several European ERASMUS projects in the 3-D model-based coding and digital copyright protection field. He was actively involved in MPEG4 and several European RACE projects, namely, MORPHECO (Morphological codec for still and moving images), MAVT (Mobile Audio Visual Terminal), and ACCOPI (Access control and copyright protection for images). He is the author of more than 20 research publications. He holds four patents in digital video coding field and consulted several industry companies. His main research interests include nonlinear image/video processing, analysis, compression and mathematical morphology. Since 1995, he has been with Microsoft Corporation, Redmond, WA, where he works on nonlinear filtering, segmentation, and motion analysis for advanced video compression.

Dr. Gu was awarded the title of China National Best Student.

**Tomislav Markoc** received the bachelors degree in electrical engineering from University of Zagreb, Croatia, in 1990.

During 1990 and 1991 worked at University of British Columbia, Canada, in the position of electrical engineer. In 1993 he joined EyeTel Communications, Inc., where he worked on telecommunication software. In 1995 he joined Microsoft Corp., Redmond, WA, where he works on advanced video compression technology and MPEG 4 standard. Current interests include audio and video processing.

**Steven I. Zabinsky** received the Ph.D. degree in physics from the University of Washington, Seattle, in 1993.

His work on X-ray absorption fine structure (XAFS) calculations has been published in *Physical Review B* and in *Physical Review Letters*. The codes implementing this theory are being used by experimental XAFS groups worldwide. Since graduating, he has been employed at the Microsoft Corporation, Redmond, WA, where he is helping to develop video compression algorithms and working on multimedia drivers.

**Richard Szeliski** (S'78–M'79) received the B.Eng. degree in electrical engineering from McGill University, Montreal, in 1979, the M.Eng. degree in electrical engineering from the University of British Columbia, Vancouver, in 1981, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1988.

He is a Senior Researcher in the Advanced Technology and Research division of Microsoft Corporation, Redmond, WA. His research interests include three-dimensional computer vision, motion estimation, geometric modeling, and virtual environments. He is the author of *Bayesian Modeling of Uncertainty in Low-Level Vision* (Kluwer).

Dr. Szeliski is a member of ACM and Sigma Xi, and an Associate Editor of the IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE.