Model-Based Motion Estimation for Synthetic Animations

Maneesh Agrawala* Andrew C. Beers* Navin Chaddha†

* Computer Science Department, Stanford University
†Computer Systems Laboratory, Stanford University

ABSTRACT

One approach to performing motion estimation on synthetic animations is to treat them as video sequences and use standard image-based motion estimation methods. Alternatively, we can take advantage of information used in rendering the animation to guide the motion estimation algorithm. This information includes the 3D movements of the objects in the scene and the projection transformations from 3D world space into screen space. In this paper we examine how to use this high level object motion information to perform fast, accurate block-based motion estimation for synthetic animations.

The optical flow field is a 2D vector field describing the translational motion of each pixel from frame to frame. Our motion estimation algorithm first computes the optical flow field, based on the object motion information. We then combine the per-pixel motion information for a block of pixels to create a single 2D projective matrix that best encodes the motion of all the pixels in the block. The entries of the 2D matrix are determined using a least squares formulation. Our algorithms are more accurate and much faster in algorithmic complexity than many image-based motion estimation algorithms.

1 INTRODUCTION

A problem with synthetic animations is that, like video sequences, they contain a large amount of data. At a resolution of 352 by 240 pixels, five minutes of NTSC video requires about a gigabyte of data storage. To send such an animation across a 10 Mbit/sec network (e.g. Ethernet) would require approximately 15 minutes, assuming a sustained transfer rate of 10 Mbit/sec. In order to store long sequences or send them across a network, the data must be compressed.

Motion compensation is the first step in many video compression algorithms including MPEG [7], because it allows the compression algorithm to take advantage of frame to frame image coherency. The motion estimation algorithm and the associated motion model must be fast and accurate for this type of motion compensation to be practical.

Authors' Address: Center for Integrated Systems
Stanford University,
Stanford, CA 94305-4070
E-mail: maneesh@cs.stanford.edu, beers@cs.stanford.edu, navin@sleep.stanford.edu
World Wide Web: http://www-graphics.stanford.edu/

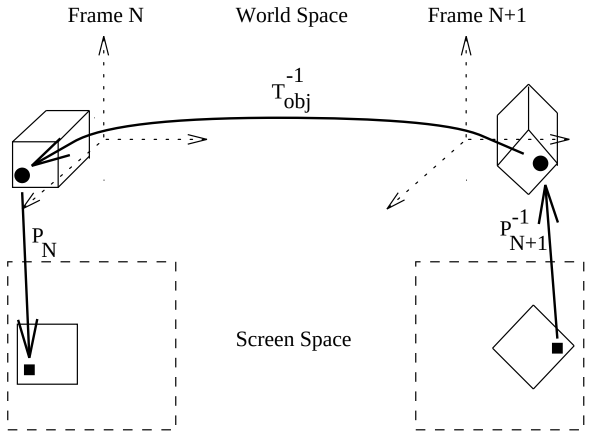

Figure 1: Backprojecting a pixel from frame $N+1$ into frame N is done by multiplying the frame $N+1$ pixel position through three matrixes. The inverse projection matrix P_{N+1}^{-1} transforms the pixel into frame $N+1$ world space. The matrix T_{obj}^{-1} transforms it back to frame N world space and the projection matrix P_N transforms it into the frame N screen space.

One approach to performing motion estimation on synthetic animations is to treat them as video sequences and use standard image-based motion estimation algorithms such as those used in MPEG[7]. Alternatively, we can take advantage of information used in rendering the animation to guide the motion estimation algorithm. Such information includes the 3D world space movements of the objects in the scene and the projection transformations from world space into screen space. In this paper we examine how to use this motion information to perform fast, accurate block-based motion estimation.

Many motion estimation schemes attempt to approximate the optical flow field between frames of a sequence. The optical flow field is a 2D vector field describing the motion of each pixel from frame to frame.

For synthetic animations there is a straightforward method for computing the optical flow field. When rendering a synthetic animation, we know the transformations from world space to screen space P_N and P_{N+1} for frames N and $N+1$, the transformations T_{obj} that each object undergoes between the frames, and the object that is visible in each pixel. Given this information, we can backproject each pixel in frame $N+1$ to its corresponding position in frame N, as shown in figure 1. This method for computing the optical flow field is used in the view interpolation schemes presented in [2]. It has also been used in previous work on motion compensation for synthetic animations and it is the basis for the model-based algorithms we present.

Guenter et al. [5] describe a lossless motion compensated compression algorithm for synthetic animations. In their scheme all the information necessary to compute the optical flow vector for each pixel is sent to the decoder. Thus, the encoder first sends the set of world space object transform matrices and the projective transform matrices to the decoder. It then encodes the (x, y, z) and an object ID for each pixel. Based on this information, the decoder can reconstruct the frame by backprojecting each pixel into the previous frame as described above. Because the decoder has access to the exact motion of each pixel, it can compensate for many types of motions, including translations, scalings, and rotations. However, compared to image-based motion estimation schemes, this approach incurs a large overhead in sending the z and object ID information for each pixel.

In general, the transmission overhead required to send motion information (such as optical flow vectors) for every pixel is large and can outweigh the advantage of motion compensation. To overcome this problem, motion compensation is often performed on blocks of pixels rather than individual pixels. Block based motion estimation proceeds as follows. For each square block of pixels in the current frame, a block that is similar to it (in terms of mean-squared or mean-absolute error) in the previous frame is found. The encoder determines the translational offset between two blocks and sends this translational vector to the decoder. The same translational vector is used for every pixel in the block.

In [9], Wallach et al. describe two methods for quickly calculating the optical flow and the per-block motion vectors of a synthetic animation using either the Gouraud interpolation or the texture mapping capabilities present in many modern hardware rendering systems. Once Wallach et. al. determine the per-pixel motion they calculate the mode vector (i.e. the most frequently occuring vector) within each 16×16 block, and use this vector to center a brute force search for the best block match. Like block-based image motion estimation schemes, this approach yields a 2D translational vector per block. The main drawback of these schemes is that they do not properly account for non-translational motion such as rotations, scales, and perspective warps of blocks.

Our model-based motion estimation schemes are aimed at accounting for translational and non-translational block motions. Like Guenter et al. and Wallach et al., our first step is to compute the optical flow field between frame N and frame $N+1$. We then consider each $B \times B$ block of pixels in frame $N+1$, as well as their corresponding optical flow vectors, and try to find a single 2D transformation matrix that best preserves these vectors. By using a two-dimensional matrix, we can encode more than just the translational motion of each block and thereby produce more accurate motion information. However, this increased accuracy comes at the cost of increasing the number of motion parameters that must be sent per block. Our results indicate that extra cost of sending more motion parameters is outweighed by the increase in motion accuracy. Furthermore, our algorithms are much faster in algorithmic complexity than many image-based methods.

This paper is organized as follows. Section 2 describes the algorithms for model-based motion estimation. Section 3 gives the computational complexity of different motion estimation methods. Section 4 describes our evaluation methodology. Section 5 gives the results. We describe future directions for this project and conclude in section 6.

2 ALGORITHM DESCRIPTION

While many techniques for block-based image motion estimation have been proposed, the simplest approach is to consider each $B \times B$ block in frame $N+1$ and perform a brute force block search for the the best matching block in frame N within a search window. The search window may be specified by the pair $[m, n]$, where m and n represent the the smallest and largest offsets respectively, to be checked along each axis. Given a block in frame $N+1$ centered at location (x, y), the brute-force algorithm (BRUTE) centers the search window at this location in frame N and examines every block within the search window to find the best match.

The strategy behind our motion estimation schemes is to make use of the object motion information that is available for synthetic animations. More specifically, we use the object movement information to quickly and accurately create a 2D transformation matrix per block in frame $N+1$ that best encodes the frame to frame motion of each pixel in the block. The basic algorithm consists of a loop over two steps and can be described as follows.

Loop over all the $B \times B$ blocks in frame $N+1$

1. For each pixel in the block, compute the corresponding pixel in frame N by backprojecting it into frame N. This creates an optical flow field for the pixels in this block.
2. Based on the optical flow field computed in step 1 use a least squares formulation, to compute a 2D transformation matrix that best preserves the pixel correspondences.

2.1 Computing Optical Flow

The first step in all our motion estimation algorithms is to compute the optical flow field from frame to frame. To do this we determine a correspondence between each pixel location in frame $N+1$ and locations in frame N by backprojecting each pixel in frame $N+1$ through a backtransform matrix. In order to determine the backtransform, we need to know for each pixel in frame $N+1$:

- which object is visible in the pixel,
- the z value of each pixel (immediately available with z-buffered rendering systems),
- the world space transformation matrix T_{obj} between frame N and $N+1$ for each visible object,

- the projection matrices P_N and P_{N+1} from world to screen space for each frame

We assume that objects move as rigid bodies, so the motion of the entire object is described by a single matrix T_{obj}. As shown in figure 1, under these conditions if q^{N+1} is the (x, y, z) position of a pixel in frame $N+1$, then its position q^N in frame N is given by the equation

$$q^N = P_N T_{obj}^{-1} P_{N+1}^{-1} q^{N+1} \quad (1)$$

We can determine the (x, y, z) location of each pixel in frame $N + 1$ by looking at its screen position and z-buffer value. By rendering an object ID map where the colors in the scene are replaced by object IDs, we can determine to which object each pixel belongs by looking at the corresponding location in the object ID map. Thus, each frame must be rendered twice, once to create the object ID map and once to create the real color rendering. On rendering systems that support four color channels, however, we could render the color channels and the object ID at the same time. The matrix $B_{obj} = P_N T_{obj}^{-1} P_{N+1}^{-1}$ is called the "backtransform matrix".

While this is the most straightforward method for computing the optical flow field based on information about world space object movements, other methods have been developed. Wallach et al. [9] propose two different schemes for computing the optical flow field using graphics hardware, one using Gouraud interpolation and the other using texture mapping. They backproject each vertex of the object in frame $N + 1$, and then interpolate the flow field across each polygon using the interpolation hardware. The complexity of our backprojection method is dependent on the number of pixels in a frame, while the complexity of Wallach et al.'s methods are dependent on the number of polygons in the scene. Thus, as the number of polygons in a scene increases, the performance of Wallach et al.'s methods degrades while the performance of the backprojection scheme remains constant.

2.2 Issues with Backprojection

There is one case in which the backtransformation scheme described in section 2.1 for generating the optical flow field does not work. Consider a pixel q^{N+1} on a surface that is visible in frame $N + 1$ and not visible in frame N. This can occur, for example, when one object occludes another object in frame N and moves so that it no longer occludes that object in frame $N+1$. In this case the frame N location q^N generated by backtransforming q^{N+1} will not correctly represent the optical flow. We reduce the severity of this problem by checking the object ID of q^N and making sure that it matches the object ID of q^{N+1}. In performing the second step of our algorithm we only consider optical flow vectors for which the frame $N+1$ and frame N object IDs match. It is also possible to reduce the severity of this problem by estimating frame $N+1$ motion from both frame N and frame $N+2$. Objects that are occluded in frame N may be visible in frame $N + 2$. Although such a scheme can easily be fit into the model-based framework we present, we do not discuss forward frame motion estimation, as we assume that the decoder has no future information.

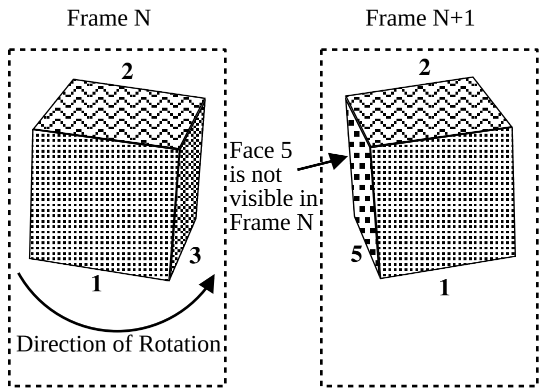

Figure 2: While face 5 of this object is visible in frame $N+1$, it is occluded by faces 1 and 2 in frame N. Backprojecting pixels that fall in face 5 will yield invalid optical flow vectors.

The object ID checking scheme presented above does not account for an object that occludes itself as it moves from frame to frame. See figure 2. One technique for finding these self occlusions is to make sure that the backtransformed z-value of the frame $N + 1$ pixel matches (or is within epsilon of) the z-value of the frame N pixel. While this technique is limited by the precision of the z-buffer, it works well for identifying self occlusions. We have found in the animations we tested that the model-based motion estimation algorithms perform better if we include self occluded vectors in our calculations. This is because the surface colors of objects tend to be similar over the entire object. Even if the vector does not represent the actual movement of the pixel on the object surface, the color contained in the incorrect frame N location is often close to the correct color. As we will show, the model-based motion estimation schemes require at least three vectors to produce a reasonable estimate. In cases where we have less than three good optical flow vectors, using these self-occlusion vectors allows our schemes to compute a reasonable motion estimate.

Even when the backprojection scheme produces exact pixel correspondences, the color of corresponding pixels may be slightly different due to shading changes. However, the change should not be very large for relatively small object movements and for white lights the color change will only reflect a change in the luminance of the pixel.

2.3 Building the Block Motion Matrix

Once we have computed the optical flow field we consider a $B \times B$ block of pixels in frame $N + 1$ and their optical flow vectors. In most cases this gives us a set of B^2 optical flow vectors, although it is possible to have fewer valid vectors per block. As described earlier, vectors that project pixels of one object into another object are considered invalid and are not included in our calculations. We want to build a single 2D transformation that best encodes all the valid optical flow vectors. If we

Least Squares Formulation

Consider a $B \times B$ block in frame $N+1$ containing n valid optical flow vectors. These vectors give us a set of n pixel locations $q_i^N = (x_i^N, y_i^N, 1)$ in frame N which correspond to the locations $q_i^{N+1} = (x_i^{N+1}, y_i^{N+1}, 1)$ in frame $N+1$. We desire a six or nine parameter 2D transformation A that best preserves the correspondences between points q^{N+1} and q^N. Using a six parameter transform results in the following equation:

$$\begin{aligned} Aq^{N+1} &= q^N \\ \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i^{N+1} \\ y_i^{N+1} \\ 1 \end{bmatrix} &= \begin{bmatrix} x_i^N \\ y_i^N \\ 1 \end{bmatrix} \end{aligned} \tag{2}$$

To solve for the unknowns, we need to solve the following system of linear equations:

$$M\vec{s} = \vec{t} \tag{3}$$

$$\begin{bmatrix} x_0^{N+1} & y_0^{N+1} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_0^{N+1} & y_0^{N+1} & 1 \\ \vdots & & & & & \vdots \\ x_{n-1}^{N+1} & y_{n-1}^{N+1} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_{n-1}^{N+1} & y_{n-1}^{N+1} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} x_0^N \\ y_0^N \\ \vdots \\ x_{n-1}^N \\ y_{n-1}^N \end{bmatrix} \tag{4}$$

When using a nine parameter transform, the entries of the last row of the matrix in equation 2 would be replaced by variables g, h, and i. Similarly, equation 4 would be extended to solve for all nine unknowns.

We optimize the least squares method by exploting the structure of matrix M, allowing us to solve equation 4 in linear time. Let $\vec{x}^N$ be a vector of the x components of the points q^N, and $\vec{x}^{N+1}$ be a vector of the x components of the points q^{N+1}. Similarly, let $\vec{y}^N$ be a vector of the y components of q^N and $\vec{y}^{N+1}$ a vector of the y components of q^{N+1}. We begin by splitting equation 4 into two parts:

$$\begin{aligned} M'\vec{z}_1 &= \vec{g}_1 \\ \begin{bmatrix} \vec{x}^{N+1} & \vec{y}^{N+1} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} &= \begin{bmatrix} \vec{x}^N \end{bmatrix} \end{aligned} \quad \text{and} \quad \begin{aligned} M'\vec{z}_2 &= \vec{g}_2 \\ \begin{bmatrix} \vec{x}^{N+1} & \vec{y}^{N+1} & 1 \end{bmatrix} \begin{bmatrix} d \\ e \\ f \end{bmatrix} &= \begin{bmatrix} \vec{y}^N \end{bmatrix} \end{aligned}$$

where M' is an $B^2 \times 3$ matrix in the worst case. Note that if we were solving for a nine parameter matrix, we would split equation 4 into three parts here.

We solve these systems using a standard least-squares approach:

$$z_i = (M'^T M')^{-1} M'^T \vec{g}_i \tag{5}$$

where

$$M'^T M' = \begin{bmatrix} \|\vec{x}^{N+1}\|^2 & \langle \vec{x}^{N+1}, \vec{y}^{N+1} \rangle & \sum_{i=0}^{B^2-1} x_i^{N+1} \\ \langle \vec{x}^{N+1}, \vec{y}^{N+1} \rangle & \|\vec{y}^{N+1}\|^2 & \sum_{i=0}^{B^2-1} y_i^{N+1} \\ \sum_{i=0}^{B^2-1} x_i^{N+1} & \sum_{i=0}^{B^2-1} y_i^{N+1} & B^2 \end{bmatrix} \tag{6}$$

have more than three valid optical flow vectors within a block we can set up an over-constrained linear system and use a least squares minimization to solve for the entries of this matrix. We can formulate the system as shown in the box entitled "Least Squares Formulation".

We have implemented two motion estimation schemes, LSQ6 and LSQ9, based on the six and nine parameter least squares formulations. With LSQ9 the resulting matrix A is a general 2D transform, while with LSQ6 it is an affine 2D transform.

In doing the least squares computation we can consider either all the valid optical flow vectors in the block (LSQ-ALL), or we can consider only the valid vectors of the object containing the most pixels in the block (LSQ-ONE). The first method tries to find the best fit across object edges, while the second method only considers vectors within one object.

2.4 Issues with Least Squares

Whenever a block contains less than three valid optical flow vectors, the least squares solvers LSQ6 and LSQ9 do not have enough information to uniquely solve the system in equation 4. Since they cannot determine a reasonable block motion matrix, we use identity as the motion matrix. Although it is unlikely that this produces a good block match, the least squares algorithms do not have enough model-based information to produce a better match. In these cases, since the BRUTE algorithm performs an image based search for the best matching block, it tends to find a better match than the least squares techniques.

Another case where brute force tends to do better than the least squares approach occurs for blocks containing object edges (i.e. when two or more objects appear in the same block). In these blocks, if the objects are moving in different directions, the LSQ-ALL least squares scheme tends not to preserve any of the motion vectors well in attempting to find a "best fit" for all of them. Although the LSQ-ONE scheme preserves the motion vectors of pixels within the object covering the largest percentage of the block, it does not perform well for pixels in the block belonging to other objects.

In some applications that involve synthetic animations, a single object is being manipulated against a solid color background. Whenever a frame $N+1$ block contains a majority of background color pixels we check whether the error frame variance would be lower if the entire block contained the background color, or if the block was reconstructed using motion parameters computed using a motion estimation technique. If a solid background colored block gives us a lower variance, the decoder is told to fill the block with background pixels, and no motion parameters are sent.

2.5 Hybrid Approaches

Based on the observations presented in the previous section, we have developed two hybrid motion estimation schemes that combine the least squares approach with the image-based brute force technique to obtain more accurate motion estimates. The simplest scheme, HYBRID-SMPL, performs the least squares estimation as well as the brute force estimation on each block. It then reconstructs the block using the motion information produced by these two motion estimation schemes and differences each reconstruction with the original block. The method that produces the smallest variance in the difference block is chosen as the motion estimation method for the block. The other hybrid scheme, HYBRID-EDGE, performs the basic least squares estimation on all the blocks and performs a brute force search only on blocks containing object edges and blocks which contain less than B^2 valid correspondences as these are the blocks in which new information is being introduced.

For comparison we have also implemented the predictive brute force (P-BRUTE) motion estimation scheme presented by Wallach et al. [9]. After generating the optical flow field they find the mode of the pixel motion vectors in each $B \times B$ block and use that vector to center a brute force search. They show that the P-BRUTE algorithm converges on the best block match with a smaller search window than standard BRUTE and therefore is often faster than BRUTE. We have experimented with both P-BRUTE or BRUTE within our hybrid techniques. Figure 3 shows how the various algorithms we have presented fit together.

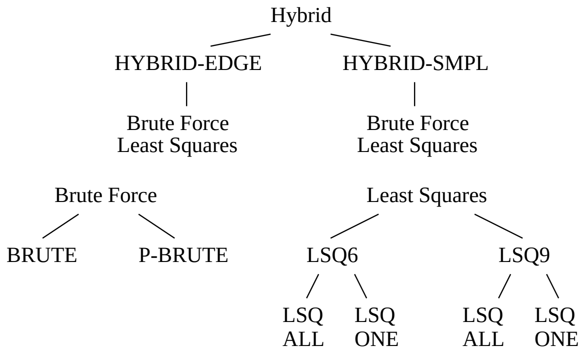

Figure 3: Overview of all the algorithms. Each branch in each tree represents a different motion estimation algorithm.

2.6 Block Reconstruction

Once we have generated a 2D transformation matrix for each block in frame $N+1$, we can reconstruct the frame by applying the matrix to each pixel in the block and using the color of the pixel at the transformed location in frame N. In most cases, however, the transformed location will not lie exactly at an integral pixel location. We use bilinear interpolation on the four pixels adjacent to the transformed location in order to determine the color at the transformed location.

If the original renderings are anti-aliased it is especially important to perform this interpolation in order to avoid introducing more sampling errors. By interpolating instead of point sampling, we obtain a better approximation of the color at the transformed pixel location.

2.7 Compressing the Motion Matrices

While most block based motion estimation algorithms for image compression generate two motion parameters Δx and Δy per block, our algorithms can generate six or nine parameters per block. Compressing these parameters is therefore essential to maintaining a good overall compression rate. In the model-based algorithms we propose, lossless coding would require many bits for encoding the motion information because the parameters generated using the least-square formulation have floating point precision. For example, Lempel-Ziv coding on a quantized version of the motion matrices resulted in a compression of only 2:1. Thus, we propose an algorithm for compressing this motion information based on vector quantization of the motion matrices.

Vector Quantization (VQ)[4] is a lossy compression technique. It is the extension of scalar quantization to higher dimensional spaces. In VQ, a vector of samples is quantized together to one of a number of predetermined reproduction vectors, called codewords. In full search vector quantization, the encoder consists of an exhaustive search for the minimum distortion codeword while the decoder consists of a table lookup. A major drawback of full search VQ is its high encoding complexity. Tree structured VQ[4] (TSVQ) is one scheme to reduce the encoding complexity by replacing the full search by a sequence of binary searches. Both full search and TSVQ produce a fixed rate-code. A variable rate code can be implemented with TSVQ by using an unbalanced tree. The advantage of doing this would be to allocate fewer bits to the commonly occuring motion matrices and more bits to matrices which occur very rarely. This is similar to the principle of Huffman coding.

To design an unbalanced tree we use a greedy growing algorithm[8]. In this method the tree is grown one node at a time. The node with the largest ratio of decrease in distortion to increase in rate or entropy is split. Hence, each split optimizes the rate-distortion tradeoff. We further prune this unbalanced tree using the Generalized BFOS algorithm[3]. This algorithm trades off the entropy of leaves for the average distortion. In this algorithm the average entropy is minimized instead of the average length.

Thus we obtain a TSVQ codebook using a greedy growing algorithm followed by pruning using a generalized BFOS algorithm. We choose a large training set representative of the different kinds of motion in synthetic animations. The motion matrices are used as the input vectors.

The encoder for compressing the motion matrices uses the motion matrix for each block obtained from the least squares algorithm as the input vector and outputs the index of the codeword (in the motion matrix codebook) closest to the input vector. The decoder uses the index to look up the codewords of motion matrices and outputs a set of quantized motion matrices. These quantized motion matrices are then used to reconstruct the frame.

3 ALGORITHMIC COMPLEXITY

There are several performance measures we can use to compare the model-based motion estimation algorithms with image-based algorithms. These include the algorithmic complexity of the methods, the quality of the reconstructed frame (i.e. how close it is to the original), and how well the error frame, that is the difference between the estimated and original frame, compresses. In this section we will consider the first of these measures; we will consider the other two in section 5.

Since all of the motion estimation algorithms we consider are block based we will analyze the complexity of performing each algorithm on a $B \times B$ block of pixels. The complexity of all the model-based algorithms is dependent on the number of valid optical flow vectors in a block. For this analysis we will assume all B^2 vectors are valid as this is the worst case. The complexity of the final block reconstruction from the computed motion parameters at the decoder is not included in the analysis of the algorithms. Table 4 summarizes the analyses discussed in this section.

3.1 Complexity of Brute Force

As described in [6], for a square, symmetric search window of size [-n,n] (that is, $\pm n$ pixels in both horizontal and vertical directions),the BRUTE algorithm requires $(2n+1)^2$ block compare operations. The block compare operation, based on a mean absolute error criterion, consists of summing the absolute differences between pixels in frame $N+1$ and the corresponding pixels in frame N on a block by block basis. For a $B \times B$ block, $2B^2$ operations are required, since B^2 absolute differences, $B^2 - 1$ additions and 1 compare operation must be performed. Thus, the total cost per block of the BRUTE algorithm is $(2n+1)^2 \cdot 2B^2$. Although faster image-based block motion estimation algorithms exist, as described in [6], they are less accurate than BRUTE. Thus, we compare our model-based schemes to BRUTE.

3.2 Complexity of Optical Flow

The first step in each of the model-based motion estimation techniques is calculating the optical flow field for each $B \times B$ block. As explained in section 2.1 we need a backtransform matrix for each object in the animation and an object ID map to compute this flow field. Since the cost of computing the backtransform matrices is only dependent on the number of objects in the animation, a constant factor must be added to the cost of backprojecting each block of pixels. In many cases however, the number of objects in the scene is much less than the number of blocks in each frame and this constant factor is negligible. As described in section 2.1 rendering the object ID map can be accelerated by using hardware rendering pipelines. We assume that such acceleration will be used, and do not include the cost of this rendering in our complexity analysis.

Once we have created the backtransformation matrices and the object ID map, we must multiply the (x, y, z) position of each pixel through the appropriate backtransform matrix. The multiplication requires 16 multiplies, 12 adds and because we are using homogeneous coordinates we must perform 2 divisions to find the (x, y) location of each pixel in frame N. Thus, for a $B \times B$ block the backtransform requires a total of $30B^2$ operations.

Based on the the cost of the optical flow field we can calculate the cost of the P-BRUTE algorithm. The P-BRUTE algorithm is similar to the BRUTE algorithm except that there is an extra cost for computing the optical flow field and then calculating the mode of the flow field vectors in each $B \times B$ block. Determining the mode of B^2 optical flow vectors requires approximately $2B^2$ operations. Thus, the total cost per block for the predictive brute force scheme is $(2n+1)^2 2B^2 + 30B^2 + 2B^2$ operations. Note that this n is often smaller than the one used for BRUTE.

3.3 Complexity of Least Squares

General approaches to solving a system via a least-squares method require time proportional to $8m^3$, where m is the largest dimension of the matrix. In our case, m is the number of optical flow vectors in a block which we assume is B^2. For our problem, we can exploit the simple structure of the matrix to solve the system in

Algorithm	Operation Count	with $n = 16$ and $B = 16$
BRUTE	$2(2n+1)^2 B^2$	557,568
Optical flow	$30B^2$	7,680
LSQ6	$55B^2 - 11$	14,069
LSQ9	$61B^2 - 14$	15,602
P-BRUTE	$2(2n+1)^2 B^2 + 32B^2$	565,760
HYBRID-SMPL(LSQ6)		
(with BRUTE)	$59B^2 + 2(2n+1)^2 B^2 + 2B + 10$	572,714
(with P-BRUTE)	$61B^2 + 2(2n+1)^2 B^2 + 2B + 10$	573,226
HYBRID-EDGE(LSQ6)		
(with BRUTE)	$55B^2 - 11 + 0.2(2(2n+1)^2 B^2 + 4B^2 + 8B + 12)$	125,816
(with P-BRUTE)	$55B^2 - 11 + 0.2(2(2n+1)^2 B^2 + 6B^2 + 8B + 12)$	125,918
Block Reconstruction		
(translational)	$2B$	32
(2D matrix, without interpolation)	$6B + 11$	107
(2D matrix, with interpolation)	$47B^2 + 6B + 11$	12,139

Table 4: Summary of the algorithmic complexities. The search windows for the brute force algorithms is [-n,n], and $B \times B$ is the block size.

asymptotically linear time, as shown in the box entitled "Least Squares Formulation".

The number of operations to compute matrix M' is $2B^2 - 1$ multiply/adds for each of the squared norms and the inner products. This yields a total of $6B^2 - 3$ multiply/adds if we take advantage of the symmetry of the matrix to avoid computing $\langle \vec{x}^{N+1}, \vec{y}^{N+1} \rangle$ twice. The summations over $\vec{x}$ and $\vec{y}$ each require $B^2 - 1$ operations. Again exploiting the symmetry of the matrix and only computing each sum once, the total number of operations to compute the above matrix is $8B^2 - 5$.

This matrix is always 3×3, so it can be inverted in constant time. Multiplication by M'^T requires $5B^2$ multiply/adds, and multiplication by the g_i vectors requires $6B^2-3$ multiply/adds each, for a total of $12B^2-6$ multiply/adds for the six parameter case and $18B^2 - 9$ multiply/adds for the nine parameter case. Including the $30B^2$ operations needed to compute the optical flow field, the total number of operations required by LSQ6 is $55B^2 - 11$, while LSQ9 requires a total of $61B^2 - 14$ operations.

3.4 Complexity of Block Reconstruction

Given a translational motion vector per block in frame $N + 1$, to reconstruct the block we add the translational vector to each pixel in the block and look up the transformed pixel location in frame N. Thus, for the brute force algorithm 2 additions per pixel are necessary which is a total of $2B^2$ additions per block. However, if we take advantage of spatial coherence within a pixel block 2 additions are only required for the first (i.e. lower left) pixel in each block. For the next pixel in this row the y coordinate does not change, so only 1 addition is required to transform its x coordinate. Similarly for each pixel in the first row and column only 1 addition is required to find its transformed location. For every other pixel in the block each coordinate has already been transformed to compute the position of a pixel in the first row or column, so we can simply look up the transformed position in a cache containing the transformed coordinates for the first row and column. This yields a total of $2(B-1)+2 = 2B$ additions per block.

Given a 3x3 2D transformation matrix per block, reconstructing the block requires that we multiply each pixel through the matrix and then perform a projective division on the x and y coordinates of the transformed pixel. This entails a total of 15 multiply/adds and 2 divides per pixel. However, once again we can use block coherence to only perform the matrix multiplies as necessary along the first row and column of the block. With this optimization a total of $6B + 11$ operations are required per block.

If we are interpolating during the reconstruction as described in section 2.6, we must add a constant cost of 2 floor operations per pixel, and 15 multiply/adds per pixel per color channel to perform the bilinear interpolation. Thus the total complexity of the interpolation operation for a block is $47B^2$ operations. Although interpolation increases the cost of the reconstruction algorithm, it considerably lowers the mean square error of the reconstructed image versus the original, especially for anti-aliased images.

3.5 Complexity of Hybrid Schemes

The HYBRID-SMPL algorithm performs both a least squares search and a brute force search on each block. Thus its complexity is the sum of the complexities for each scheme individually plus the complexity of deciding which method performs best. To perform this decision, we reconstruct the block using each of the two reconstruction methods described in the previous section, at a total cost of $8B + 11$ operations. We do not use interpolation with the least squares reconstruction. Computing the variance of the two difference blocks takes a total of $4B^2$ operations. Finally we choose the method producing the smaller variance, with 1 compare operation. Thus, the decision process costs a total

Sequence	# Frames	Description
BALLS	101	Two balls rolling on ramps
CUBE-SPIN	126	Four spinning cubes that overlap
FINDSPOT	196	Manipulating the the bunny to find a spot on it
ROTBUNNY	73	Bunny rotating about Y axis
SCALEBUNNY	100	Bunny scaling to different sizes
TOP-SPIN	126	A spinning top in a textured environment
TRANSBUNNY	133	Bunny translating in XY plane

Table 5: Summary of the animation sequences that comprise our animation database.

of $4B^2 + 8B + 12$ operations. The total cost of the HYBRID-SMPL scheme is therefore the complexity of least squares plus $2(2n+1)^2 B^2 + 4B^2 + 8B + 12$ operations. We assume that the cost of least squares includes the cost of computing the optical flow in this formulation.

The HYBRID-EDGE algorithm also performs a least squares motion estimation on every block and performs a brute force search only on blocks in which there are edges or new information has been introduced. The complexity of this algorithm is therefore dependent on the number of blocks for which the brute force search is performed and it varies between the complexity of only performing the least squares technique and the complexity of performing HYBRID-SMPL. With the synthetic animations we examined, a brute force algorithm was used on at most 20% of the blocks in each frame.

In each of these hybrid algorithms it is possible to replace the brute force search with any image-based block motion algorithm. We use BRUTE because it tends to be the most accurate image-based algorithm. Using a faster image-based algorithm would reduce the complexity of our hybrid algorithms. However it would reduce the accuracy of our hybrid algorithms as well.

4 EVALUATION METHODOLOGY

This section is divided into two parts. The first section describes the synthetic animation database on which the motion estimation algorithms are tested. The second section discusses the evaluation criterion for comparing different motion estimation algorithms.

4.1 Synthetic Animation Database

An important feature of any motion estimation or compression study is a database on which different algorithms can be tested and compared. There is no standard synthetic animation database available which spans the different types of applications in which synthetic animations are rendered. Furthermore, for most prerendered animations we do not have access to the object transformation matrices that were used during the rendering. Similarly we cannot use the video databases that are commonly used to test video compression algorithms because there is no rendering information for them.

Thus we have developed our own synthetic animation database containing several motion sequences corresponding to different kinds of motion commonly found in computer graphics applications. Each animation is stored as a script containing the transformations performed on each object from frame to frame. Table 5 is a list of each of the synthetic animations in the database with a short summary description of it.

Several of the animations in our database represent specific types of applications in which there are synthetic animations. The simplest animations consist of a single object moving on a black background. A set of three animations ROTBUNNY, TRANSBUNNY and SCALEBUNNY contain a single beige and brown bunny object that is either rotating about the vertical screen axis, translating in the plane of the screen or scaling up and down in size respectively. Using these animations we can examine how the model-based algorithms perform on each type of affine transformation (rotation, translation and scaling) individually.

In the BALLS animation, two textured balls roll up and down inclined ramps. One ball rolls in front of the other, making correct motion estimation difficult in blocks where the two balls overlap. During the animation, the camera also tracks toward the scene. This animation includes rotation, translation and scaling due to camera movement. The CUBE-SPIN animation shows four cubes spinning in various directions and at different speeds. The cubes overlap near the center of the frame, so new information is being introduced in each frame as new parts of each cube rotate into view. The FINDSPOT animation also contains the bunny, but one of its 69,451 polygons has been colored red. The animation shows the bunny being manipulated (rotated, translated and scaled) in order to find the spot. This FINDSPOT animation represents how someone might interact with an object to study its geometry or surface characteristics. In the TOP-SPIN sequence, a top is shown spinning in a simple environment. Both the camera and the top object move from frame to frame. In addition, all of the objects in the TOP-SPIN animation are texture mapped, to test how well reconstruction with resampling performs.

4.2 Evaluation Criteria

We are interested in three main quantitative performance measures for comparing the different motion estimation algorithms: algorithmic complexity, quality of reconstructed sequences, and compression achieved. We have already compared the complexity of the different algorithms in section 3. For comparing the quality of reconstructed sequences, we compare the Peak Signal-to-Noise Ratio (PSNR) of the DCT compressed error

Algorithm	Channel	BALLS	CUBES-SPIN	FINDSPOT	ROTBUNNY	SCALEBUNNY	TOP-SPIN	TRANSBUNNY
LSQ6 (LSQ-ONE)	Y	42.21	42.84	44.51	41.74	45.68	42.88	46.26
	U	45.06	44.63	54.51	48.89	52.45	43.69	57.34
	V	46.70	45.06	53.80	48.04	51.70	45.48	56.67
LSQ6 (LSQ-ALL)	Y	41.65	42.40	44.51	41.74	45.68	42.12	46.26
	U	44.13	43.49	54.51	48.89	52.45	44.15	57.34
	V	45.78	43.91	53.80	48.04	51.70	46.12	56.67
LSQ9 (LSQ-ONE)	Y	42.21	42.84	44.51	41.74	45.68	42.88	46.26
	U	45.06	44.63	54.51	48.89	52.45	43.69	57.34
	V	46.70	45.06	53.80	48.04	51.70	45.48	56.67
LSQ9 (LSQ-ALL)	Y	41.65	42.40	44.51	41.74	45.68	42.12	46.26
	U	44.13	43.49	54.51	48.89	52.45	44.15	57.34
	V	45.78	43.91	53.80	48.04	51.70	46.12	56.67

Table 6: Average PSNR per frame of the compressed error frame for each of the least squares algorithms. Lossless coding of the motion parameters is assumed.

frames.

To compare the compressed bitrate achieved using the different motion estimation schemes, we compress two things: (1) the error frame, which is the difference between a frame of the original animation and the reconstruction of that frame by one of the motion estimation algorithms; and (2) the motion information computed by the motion estimation algorithm. As described in the next two sections, we compress the error frame with DCT-based compression to determine the bit-rate associated with the error frame, and we approximate the bit-rate of the compressed motion information.

4.2.1 Compressing the Error Frame

The compression of the error frame is done in four stages: a transformation stage, a lossy quantization stage and two lossless coding stages. In the transformation stage, we use a two dimensional 8×8 DCT on an 8×8 block of pixels like JPEG. In the quantization stage the DCT coefficients are quantized to reduce their magnitude and to increase the number of zero value coefficients. We use the uniform mid-step quantizer with a different step size for each DCT coefficient. In the lossless coding stage we rearrange the quantized DCT coefficients into a zig-zag pattern. The zig-zag pattern is used to increase the run-length of zero coefficients found in the block. The DC coefficients are coded by taking the difference between the quantized DC coefficient of the current block and the quantized DC coefficient of the previous block. The quantized AC coefficients usually contain runs of consecutive zeroes. A coding advantage is obtained by using a run-length technique. The block codes from the DPCM and the run-length models are further reduced using Huffman coding with custom tables.

We work in the YUV space as we can decimate U and V horizontally and vertically by a factor of 2 in each dimension without much loss in visual quality. The same compression algorithm is used for the Y, U and V streams. The only difference is the quantization for Y and U,V data differ. Thus the different algorithms are compared for the bitrate and PSNR in the Y, U and V error streams.

4.2.2 Compressing Motion Information

To compare methods such as BRUTE and LSQ6 fairly, we also need to consider the overhead required to encode the motion parameters required by each scheme. The translational motion vectors in BRUTE and P-BRUTE are compressed via Huffman coding.

For the least-squares and hybrid methods, we code four different types of blocks: blocks containing background information, identity blocks (i.e. blocks that have not moved from their position in the previous frame), translational blocks with integral components, and six-parameter or nine-parameter motion blocks. A flag is used to distinguish between the four block types. To reduce the number of bits required by these flags, we Huffman code them.

The motion parameters for each of the four different types of blocks can be coded via a different scheme. The motion parameters for translational blocks can be coded like the translational vectors of BRUTE, using lossless Huffman coding. Six and nine parameter motion blocks are coded via the VQ method described in section 2.7. Background and identity blocks require no additional bits beyond the flag.

5 RESULTS

In this section the performance of the model-based motion estimation algorithms is compared with image-based brute force motion estimation in terms of error frame PSNR (i.e. quality) and bitrate. Summary results for all the animations in the animation database are presented in tables 6, 7, 8, 9 and 10 In section 5.1 we show that LSQ6(LSQ-ONE) is the best least squares method. In sections 5.2 5.3 and 5.4 we compare the P-BRUTE, LSQ6(LSQ-ONE) and hybrid algorithms to the BRUTE algorithm. Although there are several image-based block motion estimation algorithms that are faster than BRUTE, we do not compare the accuracy of our model-based algorithms with the accuracy of those algorithms because most of them are less accurate than BRUTE. All of the results described in this section were collected for animations rendered at 400×400 pixels, with a block size of 16×16 pixels.

Algorithm	BALLS	CUBES-SPIN	FINDSPOT	ROTBUNNY	SCALEBUNNY	TOP-SPIN	TRANSBUNNY
LSQ6 (LSQ-ONE)	24.375	18.772	13.788	27.080	13.137	20.629	11.201
LSQ6 (LSQ-ALL)	25.957	20.136	13.788	27.080	13.137	22.561	11.201
LSQ9 (LSQ-ONE)	24.378	18.782	13.788	27.080	13.137	20.629	11.201
LSQ9 (LSQ-ALL)	25.962	20.145	13.788	27.080	13.137	22.561	11.201

Table 7: Average bitrate per frame, measured in kilobits, for each of the least squares algorithms. Lossless coding of the motion parameters is assumed, and the overhead required by the motion parameters is not included in these numbers.

Algorithm	Channel	BALLS	CUBES-SPIN	FINDSPOT	ROTBUNNY	SCALEBUNNY	TOP-SPIN	TRANSBUNNY
BRUTE	Y	40.19	40.29	43.19	40.10	43.93	42.48	43.91
	U	38.77	41.57	53.51	47.92	53.36	42.75	53.36
	V	41.01	41.83	52.50	46.80	52.33	43.85	52.33
P-BRUTE	Y	40.35	40.30	43.22	40.20	44.10	42.50	44.57
	U	39.67	41.60	53.66	48.31	53.98	42.73	55.58
	V	42.52	41.86	52.50	47.20	53.08	43.82	54.51
LSQ6 (LSQ-ONE)	Y	42.20	42.79	44.51	41.71	45.63	42.85	46.14
	U	45.03	44.59	54.51	48.89	52.45	43.67	57.34
	V	46.70	45.01	53.80	48.04	51.70	45.46	56.67
HYBRID-SMPL	Y	42.45	42.99	44.95	42.10	46.17	43.75	46.52
(BRUTE)	U	45.55	46.64	55.34	50.00	55.83	44.95	58.13
	V	47.45	47.06	54.50	49.05	55.12	46.09	57.34
HYBRID-SMPL	Y	42.46	42.99	44.95	42.10	46.20	43.74	46.55
(P-BRUTE)	U	45.60	46.67	55.34	50.00	55.83	44.95	58.13
	V	47.52	47.06	54.50	49.10	55.12	46.09	57.34
HYBRID-EDGE	Y	42.25	42.91	44.89	42.11	45.96	43.51	46.52
(BRUTE)	U	45.63	46.64	54.91	49.56	52.57	44.93	57.34
	V	47.38	47.06	54.10	48.64	51.90	46.14	56.67
HYBRID-EDGE	Y	42.26	42.91	44.89	42.11	45.98	43.51	46.52
(P-BRUTE)	U	45.65	46.64	54.91	49.56	52.69	44.93	57.34
	V	47.38	47.06	54.10	48.64	51.90	46.14	56.67

Table 8: Average PSNR for the YUV channels for each algorithm. These numbers include the effects of using lossily compressed motion parameters.

5.1 Choosing the Best LSQ Method

As shown in figure 3 there are several variations with each of the model-based algorithms. We initially consider the performance of each of the four least squares algorithms to determine which one performs best. The average PSNR and average bitrate for these schemes are presented in tables 6 and 7. Both are calculated under the assumption that the 2D motion matrices are coded losslessly. Furthermore, these bitrates do not include the overhead required by the motion matrices. However, it is possible to determine which least squares technique performs best based on these results.

Tables 6 and 7 show that across all the animations LSQ-ALL performs worse than LSQ-ONE and that LSQ6 and LSQ9 are roughly equivalent. Since LSQ6 requires only six motion parameters rather than the nine required by LSQ9, the overhead required by the motion parameters is smaller for LSQ6 than LSQ9. Based on these observations we choose LSQ6(LSQ-ONE) as the least squares algorithm for the hybrid schemes.

5.2 Predictive Brute Force Results

The simplest model-based scheme is P-BRUTE. Like the BRUTE scheme, it only accounts for translational block motion. The search window size for all of the BRUTE and P-BRUTE results presented in this section is [-16,16]. As shown in table 8, the average PSNR for the Y channel is slightly better for P-BRUTE than BRUTE across all of the animations. The main advantage of P-BRUTE is that for a fixed reconstruction quality it typically requires a smaller search window than BRUTE. Given a large enough search window, BRUTE and P-BRUTE will perform equivalently in terms of quality. With smaller search windows the differences between the two brute force schemes are more dramatic as reported in [9].

In table 9, we consider the average bitrate per frame required by P-BRUTE for each of the animations in the database. The first row of the table lists the average motion parameter overhead per frame, and the second row lists the total average bitrate per frame. The third row shows the percentage of average total bits per frame saved by using a model-based algorithm instead of BRUTE. With P-BRUTE for example, this number is computed by differencing the average total bits per frame for P-BRUTE and BRUTE, and then dividing the difference by the average total bits per frame for BRUTE. The P-BRUTE algorithm requires fewer bits than BRUTE for most of the animations. However, it does slightly worse than BRUTE for the CUBES-SPIN

Algorithm	BALLS	CUBES-SPIN	FINDSPOT	ROTBUNNY	SCALEBUNNY	TOP-SPIN	TRANSBUNNY
BRUTE	2.718	2.741	2.365	3.117	2.410	1.720	1.672
	38.291	32.913	20.511	40.030	20.225	26.273	19.667
	——	——	——	——	——	——	——
P-BRUTE	2.974	3.300	2.373	3.208	2.451	1.724	1.663
	36.257	33.428	20.450	38.444	19.487	26.402	17.036
	5.31%	**-1.56%**	**2.97%**	**3.96%**	**3.65%**	**-0.49%**	**13.38%**
LSQ6 (LSQ-ONE)	3.670	4.295	5.472	4.760	4.955	2.879	4.322
	28.085	23.312	19.347	31.953	18.173	23.623	15.758
	26.65%	**29.17%**	**5.68%**	**20.18%**	**10.15%**	**10.09%**	**19.88%**
HYBRID-SMPL	3.539	4.226	5.180	4.660	4.785	2.583	4.176
(BRUTE)	25.729	20.869	18.255	29.434	16.536	21.240	15.185
	32.82%	**36.59%**	**11.00%**	**26.47%**	**18.24%**	**19.16%**	**22.79%**
HYBRID-SMPL	3.569	4.268	5.183	4.664	4.786	2.602	4.163
(P-BRUTE)	25.759	20.903	18.264	29.416	16.512	21.320	15.135
	32.73%	**36.49%**	**10.96%**	**26.52%**	**18.36%**	**18.86%**	**23.04%**
HYBRID-EDGE	3.761	4.321	5.298	4.703	4.879	2.766	4.243
(BRUTE)	26.286	21.004	18.593	29.784	17.770	21.782	15.366
	31.35%	**36.18%**	**9.35%**	**25.59%**	**12.14%**	**17.09%**	**21.87%**
HYBRID-EDGE	3.767	4.329	5.300	4.709	4.875	2.767	4.234
(P-BRUTE)	26.296	21.002	18.598	28.768	17.715	21.781	15.323
	31.33%	**36.19%**	**9.33%**	**25.64%**	**12.41%**	**17.10%**	**22.09%**

Table 9: Average bitrate per frame, measured in kilobits, resulting from the different algorithms. The first number in each row is the average number of bits per frame required to code the motion parameters. The second number is the total average bitrate per frame which is computed by adding the motion parameter overhead to the average bitrate per frame required to code the compressed error frames. The third number is the difference between the average total bitrate per frame BRUTE and that algorithm, as a percentage of the average total bitrate per frame for BRUTE.

and TOP-SPIN animations.

In picking the mode of the optical flow vectors for centering a brute force search, the basic assumption behind P-BRUTE is that many pixels in a block will have similar optical flow vectors. If however, a block contains many different optical flow vectors, the mode vector may not be representative of the block motion. Both the CUBE-SPIN and TOP-SPIN animations contain rotating objects, and the optical flow vectors for adjacent pixels within these rotating objects may have very different optical flow vectors. Thus, the mode optical flow vector is probably not a good predictor of the motion for these blocks and P-BRUTE performs poorly on these animations.

5.3 Six Parameter LSQ Results

The LSQ6(LSQ-ONE) algorithm performs much better than BRUTE both in terms of average PSNR and average total bitrate per frame. In particular, for the BALLS animation using LSQ6(LSQ-ONE), the average PSNR per frame in the Y channel is 2.1 dB higher than the average PSNR with BRUTE on this animation. Similarly the average total bitrate per frame for the BALLS animation is 26.65 percent less than the average total bitrate per frame with the BRUTE algorithm.

Although the LSQ6(LSQ-ONE) algorithm performs better than BRUTE for all the animation sequences, with the FINDSPOT, SCALEBUNNY, and TOP-SPIN animations it only achieves a 5.68 to 10.15 percent gain in average bitrate per frame over BRUTE. The scaling motion in SCALEBUNNY occurs in small increments and the object color varies slowly over the surface of the bunny. Thus, the translational motions found by BRUTE work well on this sequence and LSQ6(LSQ-ONE) does not have much of an advantage over BRUTE. The FINDSPOT sequence similarly contains a lot of incremental scaling on the bunny object and the gains with LSQ6(LSQ-ONE) are small. In the TOP-SPIN animation many blocks contain object edges and LSQ6(LSQ-ONE) does not reconstruct these blocks very well. Because all the surfaces in this animation are texture mapped with textures containing high frequency edges, small inaccuracies in the motion parameters can cause the error frame to contain a large amount of information. The LSQ6(LSQ-ONE) algorithm does much better than BRUTE with animations containing lots of rotations such as CUBE-SPIN and BALLS.

5.4 Hybrid Algorithm Results

The hybrid algorithms are the most accurate of all the model-based techniques we consider as shown in tables 8 and 9. The differences between the BRUTE and P-BRUTE versions of the hybrid algorithms are minor, mainly because the search window of [-16, 16] is relatively large for both algorithms.

While HYBRID-SMPL performs slightly better than HYBRID-EDGE both in terms of average PSNR and average bitrate per frame, its algorithmic complexity makes it less practical than HYBRID-EDGE. Across all of the animations, the average PSNR of HYBRID-EDGE(BRUTE) is within 0.24 dB of HYBRID-SMPL (BRUTE) and the average total bitrate of HYBRID-EDGE(BRUTE) is within 2% of HYBRID-SMPL(BRUTE).

As described in section 3 the complexity of HYBRID-EDGE is dependent on the number of blocks that con-

Animation	Average number edge or new blocks	Average percentage edge or new blocks
BALLS	116.65	18.66%
CUBES-SPIN	114.85	18.38%
FINDSPOT	92.29	14.77%
ROTBUNNY	97.08	15.53%
SCALEBUNNY	72.34	11.57%
TOP-SPIN	49.53	7.92%
TRANSBUNNY	96.69	15.47%

Table 10: The average number and percentage per frame of blocks containing object edges or new information. The percentages given in the third column are based on scenes rendered at 400×400 pixels with 16×16 blocks for a total of 625 blocks per frame.

tain edges and new information in each frame. Table 10 presents the average number of blocks per frame containing edges or new information for each for the animations in the database.

Compared to BRUTE, the HYBRID-EDGE(BRUTE) scheme performs significantly better. As shown in table 8, HYBRID-EDGE (BRUTE) produces a PSNR that is between 1.03 dB and 2.63 dB higher than BRUTE across all the animations. The HYBRID-EDGE(BRUTE) algorithm also gives between 9.35 and 36.18 percent smaller average total bitrates per frame than BRUTE.

6 CONCLUSIONS

One approach to performing motion estimation on synthetic animations is to treat them as video sequences and use standard image-based block motion estimation methods. In this paper we have shown how one can take advantage of information used in rendering the animation to perform fast, accurate block-based motion estimation. We have found that our model-based algorithms perform better than the image based BRUTE method in terms of complexity, quality of compressed error frames and bit rate.

Both our LSQ6 and HYBRID-EDGE algorithms generate six parameters per block. We use vector quantization for compressing this motion information. A slightly different approach for compressing this motion information would be to first factor each 2D transformation matrix into the simple transformations that comprise it: translations in x and y, a rotation in the xy plane, scalings in x and y, shears, and perspective warps. This decomposition is described in [1]. This factorization should decorrelate some of the motion parameters and thereby allow us to quantize the parameters more effectively.

Another model-based approach for doing motion compensation would be to run the least square algorithm on larger blocks like 64×64 to compensate for rotations etc. and then use brute force motion estimation on compensated 16×16 subblocks. This would have the advantage of sending less motion information, while allowing us to exploit the model-based motion information to account for non-translational block motion.

In the motion estimation system we have described, the estimation is being done open loop. That is, the encoder determines the motion parameters using original frames only. We are working on a closed-loop system, in which the encoder performs the motion estimation using a reconstructed frame N and an original frame N. This more closely models how the motion parameters will be used by the decoder, since the decoder only has access to a reconstructed frame N. This model-based motion estimation system will be part of a complete end to end compression system which we are currently designing.

7 ACKNOWLEDGMENTS

We would like to thank Marc Levoy and Anoop Gupta for their fruitful discussions over the course of this project.

References

[1] James Arvo, editor. *Graphics Gems II*. Academic Press, Inc., 1991.

[2] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[3] P. A. Chou, T. Lookabaugh, and R. M. Gray. Optimal pruning with applications to tree-structured source coding and modelling. *IEEE Transactions on Information Theory*, 35:299–315, 1989.

[4] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1991.

[5] Brian K. Guenter, Hee Cheol Yun, and Russell M. Mersereau. Motion compensated compression of computer animation frames. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 297–304, August 1993.

[6] A. C. Hung and T. H. Meng. Parallel array architechures for motion estimation. In T. Valero, M.; Lang, Sun-Yuan Kung, and J. Fortes, editors, *Proceedings of the International Conference on Application Specific Array Processors*, pages 214–235. IEEE, September 1991.

[7] D. LeGall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[8] E. A. Riskin and R. M. Gray. A greedy tree growing algorithm for the design of variable rate quantizers. *IEEE Transations on Signal Processing*, 39:2500–2507, 1991.

[9] Dan S. Wallach, Sharma Kunapalli, and Michael F. Cohen. Accelerated MPEG compression of dynamic polygonal scenes. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 193–197. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.