# Minimizing MPEG-4 Sprite Coding-Cost Using Multi-Sprites

Dirk Farin[a], Peter H.N. de With[b], Wolfgang Effelsberg[a]

[a]Dept. of Computer Science IV, University of Mannheim, Germany
[b]LogicaCMG / Eindhoven University of Technology, The Netherlands
farin@uni-mannheim.de

## ABSTRACT

Object-oriented coding in the MPEG-4 standard enables the separate processing of foreground objects and the scene background (*sprite*). Since the background sprite only has to be sent once, transmission bandwidth can be saved. This paper shows that the concept of merging several views of a non-changing scene background into a single background sprite is usually not the most efficient way to transmit the background image. We have found that the counter-intuitive approach of splitting the background into several independent parts can reduce the overall amount of data. For this reason, we propose an algorithm that provides an optimal partitioning of a video sequence into independent background sprites (a *multi-sprite*), resulting in a significant reduction of the involved coding cost. Additionally, our algorithm results in background sprites with better quality by ensuring that the sprite resolution has at least the final display resolution throughout the sequence. Even though our sprite generation algorithm creates multiple sprites instead of a single background sprite, it is fully compatible with the existing MPEG-4 standard. The algorithm has been evaluated with several test-sequences, including the well-known *Table-tennis* and *Stefan* sequences. The total coding cost could be reduced by factors of about 2.7 or even higher.

**Keywords:** MPEG-4 video coding, sprite coding, image mosaicing.

## 1. INTRODUCTION

One specifiy video-encoding tool in the MPEG-4 standard is the coding of a scene background as a single panoramic image (*sprite*). In general, this background image is larger than the actual video format since all views of the background are combined into a single image. The decoder can reconstruct the current background view from the sprite based on a small set of transmitted camera parameters. Hence, the sprite itself needs to be transmitted only once, which can result in a substantial improvement of coding efficiency.[1]

Interestingly enough, transmitting the background in a single sprite is not generally the most efficient approach. This paper shows that coding the background as several separate sprites can result in an overall reduced amount of data. Moreover, we show that when using the projective motion model of MPEG-4, it is only possible to cover at most 180° field of view in a single sprite. Hence, splitting the video sequence into segments and synthesizing independent sprites is not only advantageous to reduce coding cost, but it is also a necessity to cover all possible input sequences. Therefore, we present an algorithm that computes the optimal partitioning of a video sequence in terms of *minimum coding cost*.

A sprite is typically synthesized in the encoder by first applying a global-motion estimator[2] to determine camera-motion. The input frames are then warped into the reference coordinate system of the sprite to get a seamless mosaic. To our knowledge, optimal selection of the reference coordinate system has not been discussed previously, although this has direct implications for the generated sprite size and resolution. The usual approach to date is to use the first frame of the input sequence as a reference frame. Alternatively, Massey and Bender[3] propose to use the middle frame of a sequence, which results in a more symmetric sprite shape if the camera performs a continuous panning motion. Instead of using a heuristic reference frame placement, our algorithm also computes the best reference frame to minimize the synthesized sprite size.

A further problem, that has not yet been considered in literature is the problem of camera zoom-in operations. If the camera performs a zoom-in, the sprite area that is covered by the input gets smaller. Expressed in another way, this means that the pixel distance in this sprite area decreases, and hence, the sprite resolution should be increased. Otherwise, the input image would be scaled down to the coarser sprite resolution and fine detail

would be lost. To prevent this unfavourable effect, our sprite generation algorithm can incorporate a constraint ensuring that no input frame is warped to a size smaller than the input resolution. As a result, sprite coding will not cause any loss of resolution and, consequently, the quality of the decoder output will be increased.

The remainder of the paper is structured as follows. Section 2 gives an introduction to the projective camera model used for MPEG-4 sprite coding. Limitations of the model are revealed and the concept of multi-sprites is introduced. In Section 3, the three idealized examples of pure camera zoom-out, zoom-in, and camera rotation are analyzed. It will be shown that using multi-sprites can in fact reduce the total sprite size. Furthermore, the resolution preservation constraint is derived from the zoom-in example. Section 4 presents several definitions of sprite coding cost, differing in accuracy and computation speed. Moreover, it is shown how to incorporate practical constraints like a limited sprite buffer size. The multi-sprite partitioning algorithm is described in Section 5, while experimental results are presented in Section 6. The paper closes with conclusions in Section 7.

## 2. BASICS OF SPRITE CONSTRUCTION

Let us first review the physical image generation process. We assume that the camera position is fixed at the coordinate system origin. If we would allow translatorial camera motion, objects at different depths would move with different speeds because of the parallax effect. This would make it impossible to align the background images into a seamless mosaic. For this reason, we can model the image formation process with a camera rotation, followed by the projection onto the camera image plane. Using homogeneous coordinates[4] $(x', y', w')$ for pixel positions, the projection of 3-D coordinates $\hat{x}, \hat{y}, \hat{z}$ onto the image can be described as

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \underbrace{\begin{pmatrix} f & 0 & o_x \\ 0 & f & o_y \\ 0 & 0 & 1 \end{pmatrix}}_{\substack{\text{intrinsic camera} \\ \text{parameters}}} \underbrace{\begin{pmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{pmatrix}}_{\substack{\text{extrinsic parameters} \\ \text{(camera rotation)}}} \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{pmatrix} = \mathbf{H_i} \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{pmatrix}, \tag{1}$$

where $f$ is the camera focal length and $(o_x; o_y)$ is the camera principal point in the image. Multiplying the intrinsic and extrinsic transformation matrices together, we obtain the $3 \times 3$ matrix $\mathbf{H_i}$, describing the projection for input frame $i$. When input images $i, k$ contain two views of the same scene viewed with differing camera parameters, the points in image $k$ can be mapped to $i$ using the transform $\mathbf{H_{i;k}} = \mathbf{H_i H_k^{-1}}$. The transformation matrix $\mathbf{H_{i;k}}$ thus describes image motion between frame $i$ and $k$. Because homogeneous coordinates are used, the matrices $\mathbf{H_{i;k}}$ are scaling invariant and the normalization $h_{22} = 1$ is usually applied, leading to

$$\mathbf{H_{i;k}} = \begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{pmatrix} \sim \begin{pmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ p_x & p_y & 1 \end{pmatrix}. \tag{2}$$

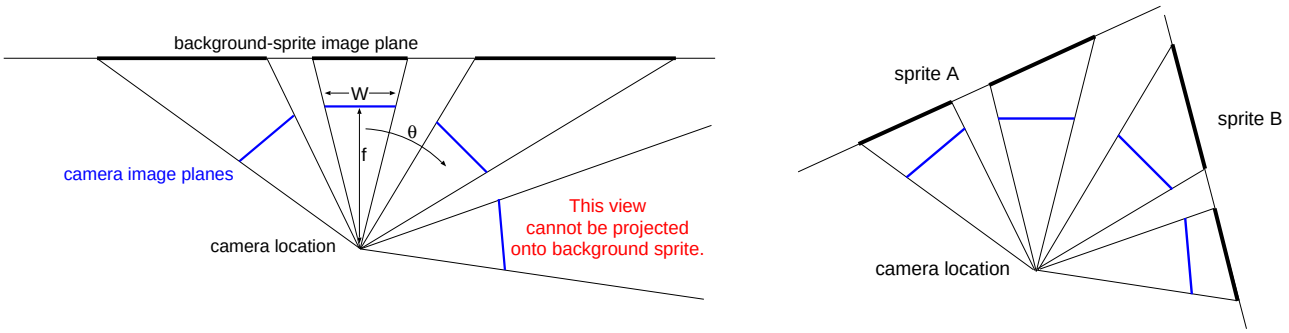
Writing the transform with non-homogeneous coordinates, we obtain the well-known perspective motion model*

$$x' = \frac{a_{00}x + a_{01}y + t_x}{p_x x + p_y y + 1}, \quad y' = \frac{a_{10}x + a_{11}y + t_y}{p_x x + p_y y + 1}. \tag{3}$$

We compute the motion model $\mathbf{H_{i;i+1}}$ between neighboring frames with a robust parametric motion-estimation technique.[2] All other $\mathbf{H_{i;k}}$ for arbitrary $i, k$ can be determined using transitive closure. The inverse $\mathbf{H_{k;i}}$ of $\mathbf{H_{i;k}}$ is computed using matrix inversion.

MPEG-4 sprite coding is based on the previously described perspective motion model, but instead of transmitting the eight projection matrix parameters, four control points specifying the motion of the image corners

*In the literature,[1,5,6] it is often proposed to use a simplified motion model for sprite generation. In most cases, the affine motion model, which is a special case of the perspective model with $p_x = p_y = 0$, is applied. Because the affine model is linear, less complex motion-estimation algorithms can be applied. However, the affine motion model cannot accurately describe camera pan or tilt. Consequently, it is generally not possible to use the affine model for geometrically undistorted sprites.




(a) The use of ordinary sprites leads to large geometric distortions.

(b) Multi-sprites reduce the distortion.

**Figure 1.** (a) Top-view of projecting the input frames onto the background sprite plane. The more the camera rotates away from the frontal view ($|\theta|$ increases), the larger the projection area on the sprite plane. For $|\theta| \geq 90°$, the projection ray does not intersect the sprite plane. Hence, only 180° field of view can be covered with one sprite. (b) Using several sprites reduces the geometric distortion and allows to cover larger viewing angles.

are transmitted. However, both representations can be converted easily into each other.[7] The background sprite is synthesized at the encoder by mapping all frames from a sequence into a common, planar reference coordinate system. Thus, the background sprite image can be envisioned as the projection of the 3-D world onto a plane. This is illustrated in Fig. 1a, which shows a top-view of a camera that rotates around its vertical axis. If the camera rotates away from the frontal view position, the area on the sprite that is covered by each image projection gets larger. For projection angles that exceed 90°, input image pixels cannot be projected onto the planar sprite anymore.

As a consequence, ordinary MPEG-4 sprites have the direct limitation that only 180° field of view can be represented in a single sprite image. In practice, the usable viewing angle is even smaller, since the perspective distortion increases rapidly when the camera rotates away from its frontal view position. Consequently, the required sprite size also increases quickly during a camera pan with short focal length. But even though some input images are projected onto a larger area in the sprite than their original size, this does not result in an increased resolution at the decoder output, since the image will be scaled down to its original resolution again at the decoder. It can be concluded that the sprite-coding is *inefficient* in the sense that it uses a high resolution for transmitting the sprite although this extra resolution is never displayed.

In this paper, we propose a more efficient coding technique based on partitioning the video sequence into several intervals and calculating a separate background sprite for each of them. Although some parts of the background may be transmitted twice, the overall sprite area to be coded is reduced. This counter-intuitive property results from the fact that the geometric distortions from camera operations do not accumulate as much in the multi-sprite case, so that larger parts of the sprite can be transmitted in a lower resolution. Figure 1b depicts the same scene as in Fig. 1a, but using a two-part multi-sprite instead of only one single sprite. Two advantages of the multi-sprite approach can be observed. First, the complete scene can be represented in the multi-sprite because additional sprite planes can be placed such that an arbitrarily large field of view can be covered. Secondly, the total projected area becomes smaller, since the sprite plane onto which the input is projected can be switched to a different sprite plane, if this results in a smaller projected area.

Our algorithm for multi-sprite generation finds the optimal partitioning of a video sequence into multi-sprites and also determines for each sprite the optimal reference coordinate system. Different sprite cost definitions can be selected to adapt the optimization criteria to different application requirements. Finally, the proposed algorithm also allows to integrate additional constraints in its optimization process. These include the specification of a maximum sprite buffer size at the decoder or a resolution preservation constraint which prevents loss of detail during camera zoom-in operations.

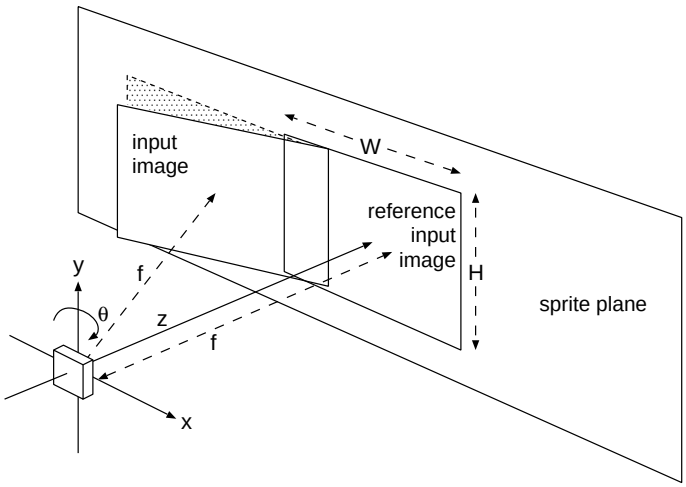



**Figure 2.** Set-up for the example case of horizontal camera pan.

## 3. EXAMPLE PROBLEM CASES

Let us first describe some idealized examples to clarify why ordinary MPEG-4 sprites are inefficient in the general case, and how this problem can be alleviated using multi-sprites. However, note that the algorithm described in Section 5 is not limited to these special cases, but finds the optimum solution for any real-world sequence.

### 3.1. Example case: camera zoom-out

As a first example, we consider the case that the camera is performing a continuous zoom-out operation. Since each image covers a larger view than the previous one, the projection area in the sprite is constantly increasing. At first, the sprite-mode is advantageous, because most of the image was already visible in the previous image. However, when the zoom continues, a point occurs where the increase of total sprite size outweighs the reuse of the already existing background content and it would be better to start with a new sprite (also see the real-world example in Fig. 8).

If the zoom factor between two neighboring frames is $s$ and the image size is $W \times H$, the sprite size after $n$ frames will be $WHs^{2n}$. The alternative construction is to create a two-part multi-sprite, with each sprite constructed from only half of the frames. The total size of the multi-sprite is thus $2WHs^{2n/2}$. Consequently, coding the scene as a two-part multi-sprite results in a lower total sprite area iff

$$WHs^{2n} \;>\; 2 \cdot WHs^{2\frac{n}{2}} \quad \leftrightarrow \quad n > \log_s 2. \tag{4}$$

Generalizing this result, a $p$ part multi-sprite gives a smaller sprite area as a $(p-1)$ part multi-sprite, provided that the sequence length $n$ satisfies

$$n \;>\; \frac{p(p-1)}{2}\big(\log_s p - \log_s (p-1)\big). \tag{5}$$

### 3.2. Example case: horizontal camera pan

Alternatively, let us now assume a camera set-up where the camera only performs rotation around the vertical axis (camera pan, Fig. 2). Input images are assumed to have normalized size $W \cdot H = 1$ and aspect ratio $W : H = 4 : 3$. Furthermore, we assume that the sprite plane is placed at a distance equal to the focal-length $f$ from the camera. Hence, if the camera is in the frontal view position, input images projected onto the sprite plane remain at the same size. If the camera leaves this frontal view position, the projection area on the sprite increases. In the following, we observe the sprite size resulting from a camera pan with angle $\alpha$. Obviously, the sprite size will be minimal if the pan is performed symmetrically. This means that when starting from the frontal view position, we rotate the camera $\alpha/2$ to the left and an equal amount $\alpha/2$ to the right. Since we assume that

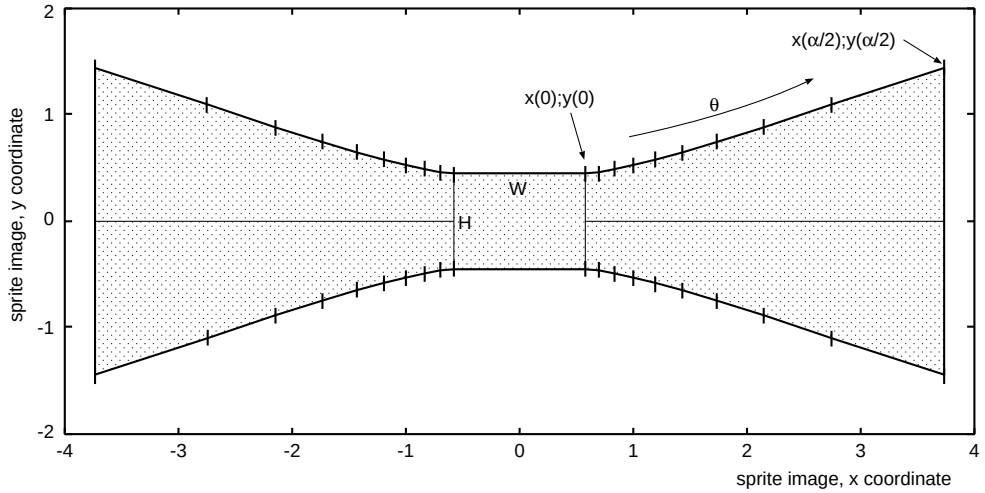



**Figure 3.** The sprite area that is covered by projecting images from a rotating camera onto the sprite plane. Regular intervals of camera rotation are depicted with small vertical marks along the area contour. Note that the projection area increases much faster at larger rotation angles.

the origin of the input image coordinate system is positioned at the image center, it is sufficient to consider only one corner of the image, since the other corners can be obtained by mirroring the $x$ and $y$ coordinates. With the abbreviations $c_\theta = \cos\theta$ and $s_\theta = \sin\theta$, our camera model is

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{pmatrix} \begin{pmatrix} \hat{x} = W/2 \\ \hat{y} = H/2 \\ \hat{z} = f \end{pmatrix} = \begin{pmatrix} f c_\theta \frac{W}{2} + f^2 s_\theta \\ f\frac{H}{2} \\ -s_\theta \frac{W}{2} + f c_\theta \end{pmatrix}. \tag{6}$$

Hence, if the camera is rotated by an angle $\theta$, the top right image corner $W/2; H/2$ projects to (Fig. 6)

$$x(\theta) = \frac{f c_\theta \frac{W}{2} + f^2 s_\theta}{-s_\theta \frac{W}{2} + f c_\theta} \quad ; \quad y(\theta) = \frac{f\frac{H}{2}}{-s_\theta \frac{W}{2} + f c_\theta}. \tag{7}$$

Consequently, the area $A$ that is covered by image content can be calculated by

$$A = W \cdot H + 4 \int_{\theta=0}^{\theta=\alpha/2} y(\theta) \frac{\mathrm{d}x}{\mathrm{d}\theta} \mathrm{d}\theta. \tag{8}$$

Figure 4 depicts the total covered sprite area $A$ for two different camera set-ups, one using $f = 1$ (wide-angle) and the other for $f = 10$ (tele). For both set-ups, three alternatives were examined using Eq. (8). The first one is the coding with an ordinary sprite[†], whereas the other two alternatives are using multi-sprites with two or three parts. In the multi-sprite cases, the total pan angle was divided into equal parts and a separate sprite was generated for each part. Figure 4 depicts the sprite area $A$ depending on the total pan angle $\alpha$ for the different set-ups. For very low pan angles, it is clear that the ordinary sprite construction is more efficient, since the multi-sprite coding has the overhead of multiple transmission of mainly the same content. However, because of the fast increasing geometric distortion of the single sprite case, the two-part multi-sprite becomes more efficient for pan angles over about 25° ($f = 10$). Finally, for angles over about 45°, using a three-part sprite gets even more efficient.

### 3.3. Example case: camera zoom-in

Another sprite generation problem, which is different from the above two cases, occurs if the camera performs a zoom-in after the reference frame. Since the resolution of the input frame is reduced when the image is mapped

[†]But including the optimal selection of the reference frame.

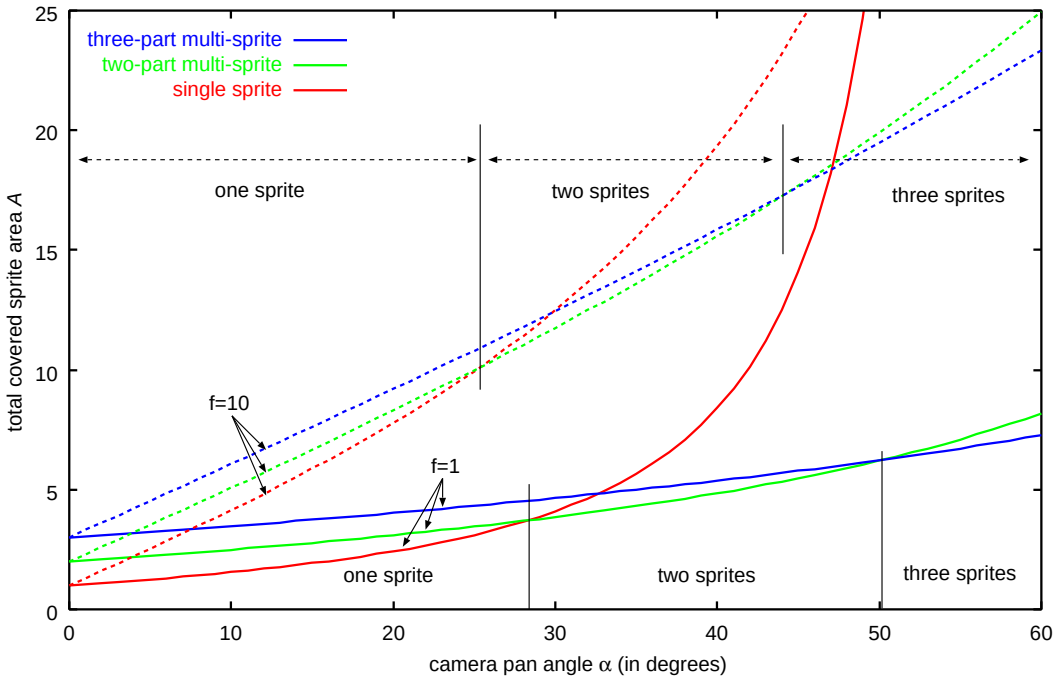



**Figure 4.** The covered sprite area for horizontal camera pan at two different focal-lengths. Reference image size is 1. If the camera rotation exceeds a specific angle, the area to be coded in the multi-sprite case is lower than for a single sprite.

into the sprite, the resulting output quality at the decoder is reduced because fine detail of the input frames is lost. To prevent this undesirable property, we have to introduce a constraint to ensure that the sprite resolution is never lower than the corresponding input resolution.

First, we calculate the magnification factor $m(x', y')$ which indicates for each pixel in the sprite, by which factor its size has been scaled with respect to the input image. To prevent quality loss, $m(x', y')$ should always be $\geq 1$ (project to the same size or larger). However, this will not be the case for general video sequences. Hence, we have to determine the minimum $m(x', y')$ for the whole sequence and increase the sprite resolution by the reciprocal value. Note that the minimum value for $m(x', y')$ will never exceed unity, because for the reference frame $m(x', y') = 1$. Since the motion model includes perspective distortion, the scaling factor is not constant over a single input frame (see Fig. 5). The local scaling factor can be computed using the Jacobian determinant of the geometric transformation (Eq. 3) which maps the input image coordinate system to the sprite coordinate system. Consequently,

$$m(x', y') = \begin{vmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} \end{vmatrix} = \frac{1}{D^2}\left[ \begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} - \begin{vmatrix} a_{00} & a_{01} \\ p_x & p_y \end{vmatrix} y' + \begin{vmatrix} a_{10} & a_{11} \\ p_x & p_y \end{vmatrix} x' \right], \tag{9}$$

where $D = p_x x + p_y y + 1$ is the denominator of the motion model equations [‡]. For non-degenerated image projections, $m(x', y')$ is monotonic in $x'$ and $y'$, and its minimum value over the image area can be found in one of the image corners. Hence, to determine the minimum value $m_l = \min_{x',y'} m(x', y')$ over a complete input image $l$, we only have to compute $m(x', y')$ for the four image corners and select the minimum value.

Second, we consider a sprite which is built from frame $i$ to $k$. Let $m_{i;k} = \arg\min_{l;\, i \leq l \leq k} m_l$ be the minimum scaling factor of all frames between $i$ and $k$. To preserve the full input resolution for all frames that were merged into the sprite, the sprite resolution has to be scaled up by a factor of $1/m_{i;k}$. The increase of coding cost induced by the enlarged sprite area can be integrated into the definition of coding cost as will be shown in Section 4.4.

[‡]For the affine motion model, which is a special case of the perspective transformation, $p_x, p_y$ are zero and, hence, $D = 1$. The pixel scale is then simply the determinant of the affine matrix and independent of $x, y$.

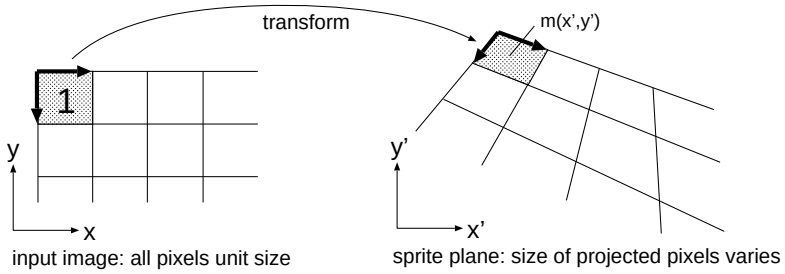



**Figure 5.** Change of local resolution. The input image (left) is warped to the sprite coordinate system (right). In general, this transformation will change the size of a pixel.

## 4. SPRITE COST DEFINITIONS

Optimization towards minimum sprite coding-cost requires a formal definition of coding cost. Thus, let $S^r_{i;k}$ be the sprite which is constructed using input frames $i$ to $k$ and which uses frame $r$ as its reference coordinate system. The following sections propose several definitions of costs $||S^r_{i;k}||$ which differ in accuracy and computational complexity. Finally, we show how constraints can be introduced into the optimization process by combining several cost definitions.

### 4.1. Bit-stream length

The obvious choice for defining the sprite coding cost is the bit-stream length itself. However, this definition is not practicable, because of the high computational complexity. The optimization algorithm for determining the optimal sprite arrangement (see Section 5) requires the cost for coding sprites of all possible frame ranges and reference frames. Calculating these costs is the most computation intensive part of the algorithm. For this reason, estimates which are easily computed will be pursued.

### 4.2. Coded sprite area

As an approximation to the actual bit-stream length, we can use the sprite area that is covered with image content. In a real implementation, Eq. (8) cannot be used, since the covered area is composed of discrete projections. Instead, we describe the coded sprite area using a polygon $x_l; y_l$ along the sprite border. Whenever an image is added to the sprite, the quadrilateral of the image border is joined with the polygon to represent the new contour. The polygon area can be calculated rapidly using Green's theorem by

$$||\cdot||_A = \frac{1}{2}\sum_l x_l y_{l+1} - x_{l+1} y_l. \tag{10}$$

Computing the sprite area for sprites over the same frame range, but with a different reference frame, can be simplified. Obviously, the relative placements of the input frame projections stay the same, regardless of the reference coordinate system. Hence, the contour polygon only has to be computed once, say, for frame $k$. In order to compute the contour polygon for another frame $i$, we only have to apply $\mathbf{H_{i;k}}$ to every point of the contour polygon and recompute the polygon area.

The sprite-area criterion assumes that the number of coded macroblocks is proportional to the bit-stream length. This is only the case if on the average, the block content does not depend on the sprite construction process. However, as we have seen previously, different areas of the sprite are synthesized with variable local resolution. Since the amount of image detail per block decreases when the projected image size increases, the relative coding cost per block also decreases. This is not reflected with the sprite area cost definition, which only considers the sprite area, regardless of the detail that is left. Hence, when using an area-based cost definition, there will be a small bias towards making magnified areas more costly than when using the *bit-stream length* cost definition of the previous section. Future work might try to compensate for this effect by integrating a *detail-loss factor*[§] into the definition based on area cost.

[§]First experiments have shown that the bit-stream length does not increase quadratically to the image scaling factor, but with an exponent of about 1.6. Interestingly, this factor is almost independent of the input image. This leads to a proposal with an area-correction factor of $m(x,y)^{1.6/2}$ to get a better approximation of bit-stream length.

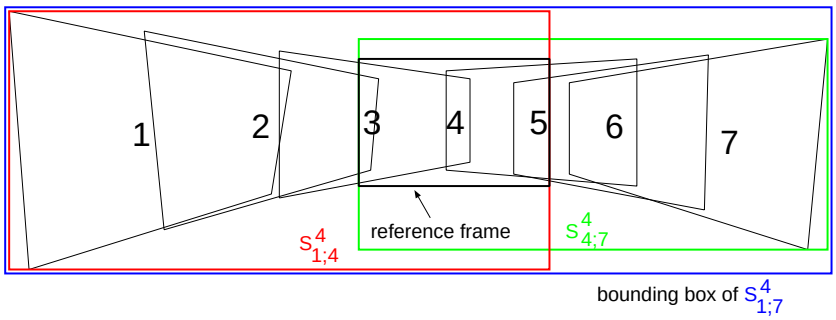



**Figure 6.** Sprite for frames 1 to 7 with frame 4 as the reference frame. In this case, the bounding-box for $S^4_{1;7}$ has been computed by combining the bounding boxes of $S^4_{1;4}$ and $S^4_{4;7}$.

### 4.3. Sprite buffer size

A further approximation to the real bit-stream length, providing a quick computation, is to take the area of the bounding box $||\cdot||_B$ around the sprite. Also note that the bounding box size is equal to the required sprite buffer size at the decoder. Hence, optimizing for the bounding box size is equivalent to minimizing memory requirements for sprite storage at the decoder. Except for rare extreme cases, the result when using the bounding box as an optimization criterion differs not much from using the really covered sprite area. The explanation is that an optimal multi-sprite arrangement will have as little distortion as possible. Hence, the covered sprite area will be almost rectangular and obviously, the bounding box is a good approximation for almost rectangular shapes.

### 4.4. Adding a resolution preservation constraint and limiting sprite buffer requirements

A cost definition based only on sprite area gives inappropriate results if the camera zooms into the scene. Since the algorithm tries to minimize the total sprite area, it will select the frame at the beginning of the zoom as reference. As we have described in Section 3.3, this would lead to a poor quality for the decoded images at the end of the zoom. Hence, we have to constrain the solution such that the local scale $m(x', y')$ in the sprite $S^r_{i;k}$ never falls below unity. This is achieved by calculating the magnification factor $m_{i;k}$ and multiplying the area size with $m^{-1}_{i;k}$. This correction factor reflects the potential resolution increasement which is carried out in the final sprite synthesis.

A further constraint may be a limited sprite buffer size at the decoder. For example, the MPEG-4 profile Main@L3 (CCIR-601 resolution) defines a maximum sprite buffer size of 6480 macroblocks. Consequently, the encoder has to consider this maximum size in its sprite construction process. We can include this constraint into the cost function by defining the cost infinity when the sprite size exceeds the buffer size limitation.

Finally, we also set the cost to infinity if the input image cannot be projected onto the sprite plane (Fig. 1a). This case can be detected by observing the orientation of the projected image corners. If the rotation angle becomes so large that the projection ray does not intersect with the sprite plane, the perspective transformation in Equation (3) will project the point backwards onto the other side of the sprite. If we assume that the image corners are sorted clockwise, we can detect this case by searching for a counter-clockwise point placement.

Adding the described constraints to the area cost definition results in the following combined cost definition

$$||S^r_{i;k}||_C = \begin{cases} \infty & \text{if input frame range } i;k \text{ cannot be projected onto a single sprite,} \\ \infty & \text{if } ||S^r_{i;k}||_B \text{ exceeds the maximum sprite buffer size,} \\ \frac{1}{m_{i;k}} ||S^r_{i;k}||_A & \text{else.} \end{cases} \tag{11}$$

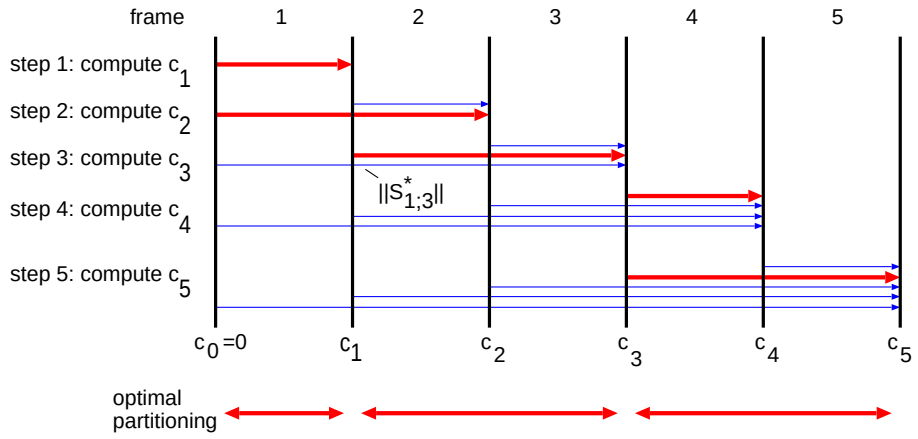



**Figure 7.** Determining the best sequence partitioning. Each state $c_k$ is assigned the minimum coding cost for a partitioning ending in frame $k$. Each arrow represents the cost for the sprite built from the covered frames. For each $c_k$ with $k \geq 1$, the sprite that results in the minimum cost in node $c_k$, is marked with a bold arrow. Tracing back the bold arrows from the last node ($c_5$) gives the optimal partitioning with minimum cost.

## 5. MULTI-SPRITE PARTITIONING ALGORITHM

To find the best sequence partitioning, the algorithm has to determine the optimal set of frames for each multi-sprite part, and additionally, for each sprite the optimal reference frame.

The multi-sprite partitioning algorithm comprises two main steps. In the first step, it computes the cost for coding a sprite $S_{i;k}$ for all possible input frame ranges $i;k$. Moreover, it determines the sprite reference-coordinate system by selecting one of the input frames as a reference. The second step partitions the complete input sequence into frame ranges, such that the total sprite coding cost is minimized.

### 5.1. Cost matrix calculation and reference frame placement

In this preprocessing step, we prepare all the sprite costs required in the main optimization step. For each pair of frames $i$, $k$ with $(i \leq k)$, we compute the cost $||S^r_{i;k}||$ for all reference frame placements $r$ with $i \leq r \leq k$ and select the $r$ that minimizes the cost $||S^*_{i;k}|| = \min_r ||S^r_{i;k}||$.

The enumeration of all possible configurations of $i, k$, and $r$ may seem computationally complex, but can be calculated efficiently for most cost definitions (including $||\cdot||_A$, $||\cdot||_B$, and $||\cdot||_C$) using a two-step approach. In the following, we assume that the cost definition is based on the sprite bounding box, but the same principle can be applied to the area computation. We begin with computing all bounding boxes for the case that the first frame is selected as reference frame ($S^r_{r;k}$). This can be computed efficiently by starting with the bounding box of $S^r_{r;r}$, which has simply the input image size. Each $S^r_{r;k}$ can now be computed iteratively from its predecessor $S^r_{r;k-1}$ by enlarging the predecessor's bounding box to include frame $k$. The same process is repeated in the backward direction to compute all $S^r_{i;r}$. Now, is it possible to quickly determine the bounding box for $S^r_{i;k}$ by computing the enclosing bounding box of $S^r_{i;r}$ and $S^r_{r;k}$ (see Fig. 6).

The second step determines $||S^*_{i;k}||$ by searching for the $r$ that results in the minimum area bounding box. These results are stored in an upper triangular data matrix consisting of the values $||S^*_{i;k}||$. These values serve as the input data for the following optimization algorithm. Additionally, we store the reference frame that minimizes the sprite cost for each $S^*_{i;k}$. This value is not needed for the following optimization step, but the actual sprite-image generation uses the information for selecting the reference coordinate system.

### 5.2. Optimal sequence partitioning

Let $P = \big((1, p_1 - 1), (p_1, p_2 - 1), (p_2, p_3 - 1), \ldots, (p_{n-1}, N)\big)$ be a partitioning of the video sequence of length $N$ into $n$ sub-sequences. We now minimize the total sprite size over all possible partitions by determining

$$P^* = \arg\min_P \sum_{(i,k)\in P} ||S^*_{i;k}||. \tag{12}$$

The minimization problem can be computed efficiently with an iterative algorithm. For each image $i$, we compute the minimum cost $c_k$ ($c_0 = 0$) of a partitioning ending in image $k$ as

$$c_k = \min_{i \in [1,k]} \left\{ c_{i-1} + ||S^*_{i;k}|| \right\}. \tag{13}$$

The index $i$ denotes the beginning of the last sub-sequence in the partitioning up to frame $k$. For each image, we store the $i$ for which the minimum was obtained. Tracing back these stored $i$-values, starting at frame $N$, results in the optimal partitioning with respect to total sprite size (Fig. 7).

## 6. EXPERIMENTS AND RESULTS

We have implemented the algorithm with the sprite cost definition of Section 4.4. For the experiments described below, we did not limit the sprite-buffer size in the decoder. This section describes the algorithm behaviour for the two sequences *Table-tennis* and *Stefan*.

From the *Table-tennis* sequence, the first shot, consisting of 132 frames has been selected. This shot shows a long zoom-out starting from a close-up of the player's hand to a wide-angle view of the complete player. Our algorithm prevents the sprite from growing too large by splitting the sequence into a three-part multi-sprite (Fig. 8). Compared with the size of an ordinary single-part sprite, the area of the multi-sprite is a factor of 2.9 smaller. The resolution-preservation constraint enforced that the first frame of each part was selected as the reference frame. Since the first frames appear with the highest resolution in the sprites, optimal reconstruction quality is assured.

For the MPEG-4 sequence *Stefan*, we first tried to generated an ordinary sprite-image for the complete 300 frames. However, because the total viewing angle during the sequence is too large, it is not possible to synthesize a single background sprite. When adding images after frame 255 (which is approximately in the middle of the final fast camera pan), the geometric distortion increases very quickly. Hence, we used only the first 255 frames for building the sprite. The resulting sprite is shown in Figure 9. Applying the multi-sprite algorithm on the complete sequence resulted in a four part multi-sprite, which is shown in Figure 10. We have measured that the total required sprite size for the multiple-sprite approach is a factor of 2.7 smaller than for the single sprite case. However, note that the multi-sprite covered the complete 300 frames of the sequence, while the ordinary sprite covers only the first 255 frames. The effect of the resolution-preservation constraint can be observed in the fourth sprite (Fig. 10d). Here, the algorithm decided to use the last frame of the camera zoom-in as a reference to preserve the full resolution. This also explains why the algorithm separated the last 45 frames (256-300) into two separate sprites. If all frames would have been combined into a single sprite, all frames would be scaled up to preserve the resolution of the last frame. However, by splitting the sequence into two sprites, frames 256-292 could be coded with a lower resolution, which outweighs the overhead of an additional sprite.

## 7. CONCLUSIONS

This paper showed that partitioning a background sprite into several independent parts results in a clearly reduced coding cost and better resolution at the same time. Our proposed algorithm computes the optimal partitioning of a sequence, the reference frame for each partition, and associated scaling factors. Clearly, the reduction of sprite area depends on the type of camera motion in the sequence. For the *Stefan* sequence, a reduction by a factor of at least 2.7 has been achieved. Moreover, note that the proposed algorithm can synthesize sprites for all kinds of camera motion, which cannot be guaranteed with an ordinary sprite construction.

The proposed algorithm is compatible to the MPEG-4 standard since the multi-sprites can either be sent one after the other, such that at every time only a single sprite is in the decoder buffer. Alternatively, all multi-sprites can be laid out beneath each other within a larger image, which is transmitted to the decoder sprite buffer as a complete entity.

(a) frames 1–51, 603 × 500 (b) frames 52–77, 587 × 501 (c) frames 78–132, 585 × 478

**Figure 8.** Multi-sprite synthesized from a long zoom-out operation. The sequence is partitioned into three separate sprites of almost the same size. The center image has been selected as the reference coordinate system (shown in a darker shade). A single-part sprite generated from the same sequence has a size of 1687 × 1516.

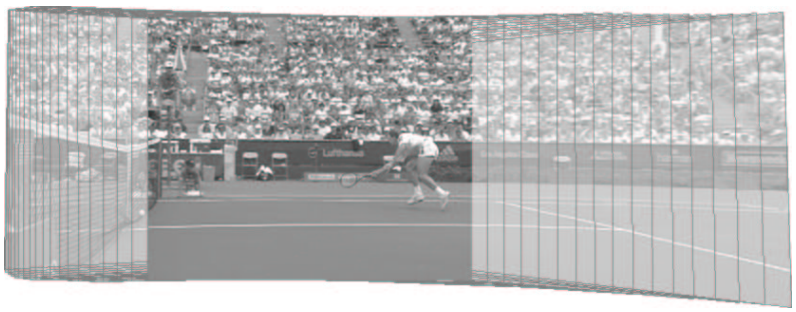
## REFERENCES


1. H. Watanabe and K. Jinzenji, "Sprite coding in object-based video coding standard: MPEG-4," in *Proc. World Multiconf. on SCI 2001*, **XIII**, pp. 420–425, 2001.
2. D. Farin, T. Haenselmann, S. Kopf, G. Kühne, and W. Effelsberg, "Segmentation and classification of moving video objects," in *Handbook of Video Databases: Design and Applications*, B. Furht and O. Marques, eds., CRC Press, Sept. 2003.
3. M. Massey and W. Bender, "Salient stills: Process and practice," *IBM Systems Journal* **35**(3 & 4), 1996.
4. R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, 2000.
5. A. Smolic, T. Sikora, and J.-R. Ohm, "Direct estimation of long-term global motion parameters using affine and higher order polynomial models," in *Proc. PCS'99, Picture Coding Symposium*, Apr. 1999.
6. M. C. Lee, W. Chen, C. B. Lin, C. Gu, T. Markoc, S. I. Zabinsky, and R. Szeliski, "A layered video object coding system using sprite and affine motion model," *IEEE Trans. on Circuits and Systems for Video Technology* **7**, pp. 130–145, Feb. 1997.
7. P. S. Heckbert, "Fundamentals of texture mapping and image warping," Master's thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989.
8. R. Szeliski and H.-Y. Shum, "Creating full view panoramic image mosaics and environment maps," *Computer Graphics* **31**(Annual Conference Series), pp. 251–258, 1997.
9. F. Dufaux and F. Moscheni, "Background mosaicking for low bit rate video coding," in *Proc. IEEE International Conference on Image Processing (ICIP)*, **1**, pp. 673–676, 1996.
10. H. Nicolas, "Optimal criterion for dynamic mosaicking," in *Proc. IEEE International Conference on Image Processing (ICIP'99)*, **4**, pp. 133–137, Oct. 1999.
11. K. Jinzenji, H. Watanabe, S. Okada, and N. Kobayashi, "MPEG-4 very low bit-rate video compression using sprite coding," in *Proc. IEEE International Conference on Multimedia and Expo (ICME)*, p. 2, Aug. 2001.
12. A. Smolic and J. Ohm, "Robust global motion estimation using a simplified M-estimator approach," in *Proc. IEEE International Conference on Image Processing (ICIP)*, 2000.
13. M. Kourogi, T. Kurata, J. Hoshino, and Y. Muraoka, "Real-time image mosaicing from a video sequence," in *Proc. IEEE International Conference on Image Processing (ICIP'99)*, **4**, pp. 133–137, Oct. 1999.
14. Y. Lu, W. Gao, and F. Wu, "Sprite generation for frame-based video coding," in *Proc. IEEE International Conference on Image Processing (ICIP)*, **1**, pp. 473–476, 2001.

**Figure 9.** Sprite synthesized from *Stefan* sequence. Only the first 255 frames can be used since it is impossible to create the sprite for the complete sequence if the first frame is selected as reference. Sprite resolution is 2445 × 1026 pixels.

(a) 1–241, 926 × 339

(b) 242–255, 699 × 296



(c) 256–292, 830 × 318

(d) 293–300, 431 × 350

**Figure 10.** Multi-sprites synthesized using the described algorithm. The respective reference frames are depicted in a darker shade. Note that the long camera pan is broken up into two separate sprites (b and c) and that the zoom-in at the end of the sequence is put into a separate sprite (d). Total sprite area of all partitions is 2.7 times smaller than in the single sprite case (which covered not all input frames).