Enabling arbitrary rotation camera-motion using multi-sprites with minimum coding cost

Citation for published version (APA):
Farin, D. S., & With, de, P. H. N. (2006). Enabling arbitrary rotation camera-motion using multi-sprites with minimum coding cost. *IEEE Transactions on Circuits and Systems for Video Technology*, *16*(4), 492-506. https://doi.org/10.1109/TCSVT.2006.872781

DOI:
10.1109/TCSVT.2006.872781

Document status and date:
Published: 01/01/2006

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Enabling Arbitrary Rotational Camera Motion Using Multisprites With Minimum Coding Cost

Dirk Farin, *Member, IEEE*, and Peter H. N. de With, *Senior Member, IEEE*

***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. We have found that the counter-intuitive approach of splitting the background into several independent parts can reduce the overall amount of data. Furthermore, we show that in the general case, the synthesis of a single background sprite is even impossible and that the scene background must be sent as multiple sprites instead. For this reason, we propose an algorithm that provides an optimal partitioning of a video sequence into independent background sprites (a *multisprite*), resulting in a significant reduction of the involved coding cost. Additionally, our sprite-generation algorithm ensures that the sprite resolution is kept high enough to preserve all details of the input sequence, which is a problem especially during camera zoom-in operations. Even though our sprite generation algorithm creates multiple sprites instead of only 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 for the sprite VOP is reduced by a factor of about 2.6 or even higher, depending on the sequence.**

***Index Terms*—Image segmentation, motion analysis, motion compensation, sprite coding, video coding, video signal processing.**

I. Introduction

ONE specific video-encoding tool in the MPEG-4 standard is the coding of a scene background as a static panoramic image (*sprite*). For a moving camera, 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].

Additional to the increased coding efficiency that can be obtained with sprite coding, the transmission of the background as a panoramic background image enables the adaptation of the video to properties of the decoder in several ways. For low bit-rate applications, it may be interesting to reduce the quality of the background to enhance the quality of foreground objects with the saved bit rate. An additional possibility is to adapt the aspect ratio of the recorded video (e.g., 4:3) to the aspect ratio of the decoder (e.g., 16:9) by filling the extra space with more data from the background sprite. Finally, since the sprite coding involves an estimation of global camera motion, the decoder can display the video also with a virtually static camera or let the user control the motion of this virtual camera.

In this paper, we take a closer look at the coding efficiency of sprites and relate this to the image formation geometry and the camera motion. We will identify four problems of previously proposed sprite generation algorithms and propose a new algorithm to solve these problems. Interestingly enough, it will be shown that transmitting the background in a single sprite is generally not the most efficient approach and that the amount of data can be reduced by splitting the background into several separate sprites. Moreover, we clarify 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, which makes the use of independent sprites a necessity. We also address the optimal placement of the reference frame for defining the sprite coordinate system and we consider a possible loss of resolution during camera zoom-in sequences.

In [2], coding with multiple sprites was proposed to reduce the distortions in a sprite. However, the distortions that the authors observed are mainly due to using the affine motion-model for sprite generation, which is an inappropriate model for rotational camera motion. Hence, the multiple sprites can only reduce the perceived distortion, but cannot achieve a geometrically correct sprite construction. Moreover, the proposed partitioning method is a heuristic, which is not theoretically motivated.

If we examine the derivation of the projective motion model from the image formation equations of a rotating camera, we observe that the motion model cannot be applied for large camera rotation angles. But even for small rotation angles, sprite coding can be inefficient since the perspective deformation increases rapidly with the rotation angle. We propose to solve this inherent problem of the projective motion model by distributing the background image data over a set of independent sprites instead of trying to code the entire sequence with a single sprite. Despite the increased overhead of using multiple sprites, the total amount of data can be considerably smaller than in the single-sprite case.

A sprite is typically synthesized in the encoder by first applying a global-motion estimator [3]–[5] 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 earlier, 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

Manuscript received August 21, 2004; revised January 24, 2006. This paper was recommended by Associate Editor K. Aizawa.

The authors are with the Technische Universiteit Eindhoven, 5600 MB Eindhoven, The Netherlands (e-mail: d.s.farin@tue.nl).

Digital Object Identifier 10.1109/TCSVT.2006.872781

1051-8215/$20.00 © 2006 IEEE

the reference frame. Alternatively, Massey and Bender [6] 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 optimal reference frame to minimize the synthesized sprite size.

A further problem that has not yet been treated in the literature is the problem of camera zoom-in operations. If the camera performs a zoom-in, the visible part of the scene becomes smaller, but the relative resolution increases. When the zoomed image is aligned to the sprite background, the sprite area that is covered by the image is smaller. If we do not want to lose the increased resolution of the input, we also have to increase the resolution of the sprite. Otherwise, the input image would be scaled down to the coarser sprite resolution and fine detail would be lost. To prevent this unfavorable loss of resolution, our sprite generation algorithm can incorporate a constraint that ensures that the input resolution is never decreased during the warping process. As a result, sprite coding will not cause any loss of resolution and, consequently, the quality of the decoder output will increase.

The remainder of the paper is structured as follows. Section II gives an introduction to the projective camera model used for MPEG-4 sprite coding. Limitations of the model are revealed and the concept of multisprites is introduced in Section III. Section IV derives a classification method to detect camera configurations for which no appropriate projective transform onto a sprite plane exists. In Section V, the three idealized examples of pure camera zoom-out, zoom-in, and camera rotation are analyzed. It will be shown theoretically that using multisprites can in fact reduce the total sprite size. Furthermore, the resolution-preservation constraint is derived from the zoom-in example. Section VI 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 multisprite partitioning algorithm is described in Section VII, while experimental results are presented in Section VIII. The paper closes with Section IX, giving an overview how the algorithm can be integrated into a video-object segmentation framework, and conclusions in Section X.

II. Basics of Sprite Construction

Let us first examine the physical image formation process for a rotation only camera. The restriction to a rotating and zooming camera is necessary since with 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.[1]

Let us define the three-dimensional (3-D) world coordinate system as right handed and let the camera be located at its origin (Fig. 1). The camera captures a number of images i with different rotations $\mathbf{R}_i$ and focal lengths f_i. In a rotated local image coordinate system, where the frontal viewing direction is along the positive z-axis, the corresponding 3-D position of each image pixel $(\hat{x}, \hat{y})$ in the focal plane is $(\hat{x}, \hat{y}, f_i)^\top$. For simplicity of notation, we assume that the origin of image coordinates is at the principal point, which can usually be assumed to be at the center of the image. Now let the virtual sprite plane be placed orthogonal to the z axis of the world coordinate system at a distance f_s. The projection of the image point $(\hat{x}, \hat{y})$ can then be determined by

$$\begin{aligned}\begin{pmatrix}x'\\y'\\w'\end{pmatrix} &= \underbrace{\begin{pmatrix} f_s & 0 & o_x\\ 0 & f_s & o_y\\ 0 & 0 & 1\end{pmatrix}}_{\substack{\text{intrinsic camera}\\ \text{parameters: } \mathbf{K}_s}} \underbrace{\begin{pmatrix} r_{00} & r_{01} & r_{02}\\ r_{10} & r_{11} & r_{12}\\ r_{20} & r_{21} & r_{22}\end{pmatrix}}_{\substack{\text{extrinsic parameters}\\ \text{(camera rotation): } \mathbf{R}_i}} \begin{pmatrix}\hat{x}\\ \hat{y}\\ f_i\end{pmatrix}\\ &= \mathbf{K}_s \cdot \mathbf{R}_i \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & f_i\end{pmatrix}}_{\substack{\text{shift onto focal}\\ \text{plane: } \mathbf{K}_i^{-1}}} \begin{pmatrix}\hat{x}\\ \hat{y}\\ 1\end{pmatrix} = \mathbf{H}_i \begin{pmatrix}\hat{x}\\ \hat{y}\\ 1\end{pmatrix}\end{aligned} \tag{1}$$

where (o_x, o_y) is the position of the principal point on the sprite plane, and the resulting sprite coordinates (x', y', w') are given in *homogeneous coordinates* [7]. Multiplying the intrinsic and extrinsic transformation matrices together, we obtain the combined 3×3 matrix $\mathbf{H}_i$, describing the projection of the homogeneous image coordinates onto the sprite plane. The transformations $\mathbf{H}_i$ are unknown and consequently, they have to be estimated from the input sequence. Since the relative placement of the sprite plane to the input frames is still undefined, the transforms $\mathbf{H}_i$ cannot be determined directly. Instead, we can only estimate the inter-image transforms $\mathbf{H}_{i;k} = \mathbf{H}_i^{-1}\mathbf{H}_k$, which map pixels from image k onto corresponding pixels in image i.

Because homogeneous coordinates are used to describe the transformation, the parameterization using the matrices $\mathbf{H}_{i;k}$ is invariant to scaling and one additional constraint has to be introduced to eliminate the extra degree of freedom. Usually, the normalization $h_{22} = 1$ is chosen,[2] 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 ordinary Euclidean coordinates, we obtain the well-known projective 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 camera motion $\mathbf{H}_{i;i+1}$ between neighboring frames with a robust parametric motion-estimation technique [5]. 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. To obtain the image-to-sprite transforms $\mathbf{H}_i$,

[1] In fact, it is also possible to generate sprites for arbitrary camera motion including translation if the background is planar. In this trivial case, the background plane can be used directly as the sprite image. However, the limitation of background planarity is so strict that this case is of little practical use.

[2] Note that this normalization fails if $h_{22} = 0$. In the above parameterization, this is the case if the angle between both image-planes is 90°.

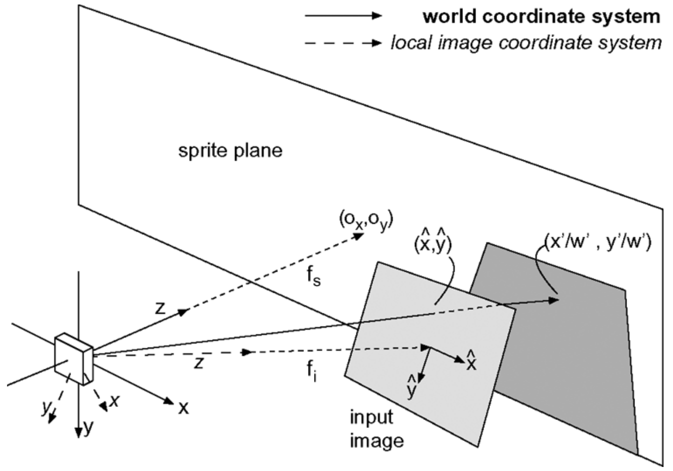


Fig. 1. Rotating camera is located at the origin of the world coordinate system. The sprite plane is assumed to be orthogonal to the z-axis. Input images are at a distance to the origin that is equal to the focal length when the image was taken. A point $(\hat{x}, \hat{y})$ on the image is projected onto the sprite position $(x'/w', y'/w')$.

we have to select one of the input images as the reference frame and place the sprite plane at the same position.[3]

In the literature, sprite generation is usually carried out by initializing the sprite with the first frame of the video sequence and then successively warping the following frames to this reference. Note that this is in fact a special case of the more general procedure described above, which is obtained when we assume that the transform of the first frame $\mathbf{H}_0$ is an identity matrix and consequently $\mathbf{H}_i = \mathbf{H}_{0;i}$. However, this predefined selection of the sprite plane position is generally not the best decision as will be shown in the subsequent sections.

Moreover, it is often proposed in the literature (see, e.g., [1], [2], [8]) to use a simplified motion model for sprite generation. In most cases, the affine motion model, which is a special case of the projective 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, which is actually the most frequent camera motion in practice. Consequently, it is generally not possible to use the affine model without introducing geometric distortions in the sprite.

III. Limitations of the Single Sprite Approach

MPEG-4 sprite coding is based on the previously described projective motion model, which in geometric terms is a plane-to-plane mapping. Thus, the sprite image can be envisioned as the projection of the 3-D world onto a plane. This is illustrated in Fig. 2(a), 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 becomes larger. For projection angles that exceed 90°, input image pixels cannot be projected consistently onto the planar sprite anymore (see, e.g., the pixel position $\mathbf{p}$).

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. If this 180° limitation is neglected and a wide camera pan is still forced into a single background sprite [9], very strong image distortions are inevitable. In practice, the usable viewing angle is even smaller since the perspective deformation 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. Unfortunately, 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. In this sense, the sprite coding is rather *inefficient*, since it uses a high resolution for transmitting the sprite although this extra resolution is never displayed.

An alternative representation for background images would be to use spherical or cylindrical image mosaics [10] for the background image, instead of a planar mapping. However, this approach has several disadvantages compared to using the projective motion model. First, generation of spherical/cylindrical mosaics requires that the internal camera parameters like focal length and the principal point of the camera are known. Even though these values can be estimated from the parameters of the projective motion model, the estimation is difficult, since the calculation is numerically sensitive. Furthermore, the estimation of the transformation parameters for cylindrical and spherical mosaics requires complicated nonlinear optimization techniques, and the reconstruction at the decoder is computationally expensive because transcendental functions are required. Finally, the obtained cylindrical/spherical background is not compliant with the MPEG-4 video coding standard, since MPEG-4 only supports the projective transformation model.

In the remainder of 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 interval. 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 perspective deformations do not accumulate much in the multisprite case, so that larger parts of the sprite can be transmitted in a lower resolution. Fig. 2(b) depicts the same scene as in Fig. 2(a), but using a two-part multisprite instead of only one single sprite. Two advantages of the multisprite approach can be observed.

- First, the complete scene can be represented in the multisprite because additional sprite planes can be placed as required to cover an arbitrarily large field of view.
- Second, 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 multisprite generation finds the optimal partitioning of a video sequence into multisprites and also determines for each sprite the optimal placement in 3-D space. 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 into its optimization process. These include the specification of a maximum sprite buffer size at the decoder or a

[3] Actually, later we will allow the sprite plane to lie at a different focal length f_s.

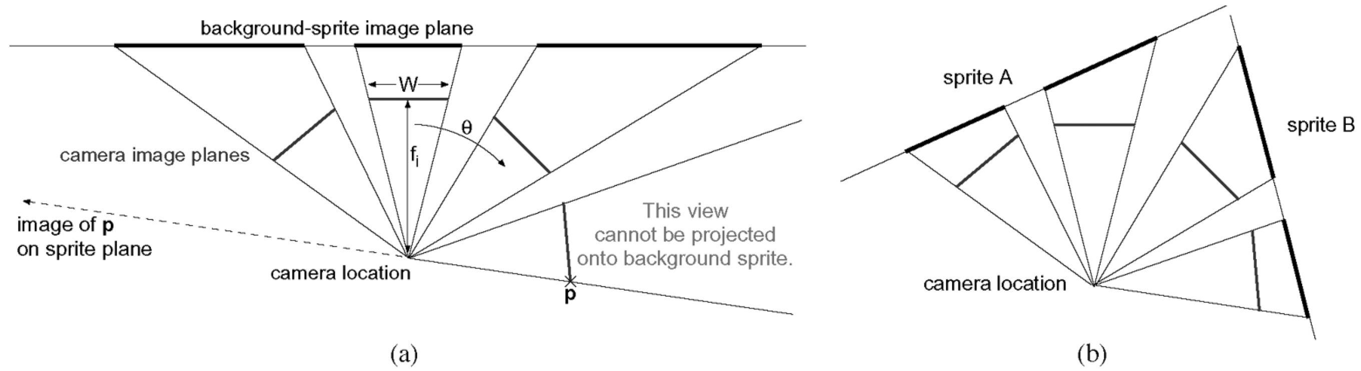


Fig. 2. (a) Top view of projecting the input frames onto the background sprite plane. The more the camera rotates away from the frontal view ($|\theta|$ increases), the larger the projection area on the sprite plane. For $|\theta| \geq 90^\circ$, the projection ray does not intersect the sprite plane. Hence, only 180° field of view can be covered with one sprite. (b) Using several sprites reduces the geometric deformation and allows to cover larger viewing angles.

resolution-preservation constraint which prevents loss of detail during camera zoom-in operations.

IV. Detecting Degenerated Transforms

As we have seen in Fig. 2(a), the projective transform maps image points using a central projection through the origin onto the flat sprite plane. As a consequence, only points in the half-space in front of the camera should be projected onto the sprite plane, since points on the back-side would be mapped ambiguously onto the same points. However, when applying the projective transform without special treatment for objects behind the camera, these objects are also projected through the optical center onto the sprite plane, where they will appear up-side down. As an example, the point $\mathbf{p}$ in Fig. 2(a) lies on the right side behind the camera and would be mapped onto the left side of the sprite. In the following, we will call a transformation which maps some image points from behind the camera onto the sprite plane as *degenerated* (see Fig. 3). These transforms must be avoided in the sprite generation process.

Usually, the camera motion $\mathbf{H}_{i;i+1}$ between successive frames is small and the problem of degenerated transforms will not be apparent. However, the concatenation of the frame-to-frame motions to determine the frame-to-sprite transform can lead to this degenerated case which maps points from behind the camera to the other side. Since the sprite construction process only knows the camera motion in the formulation of the eight-parameter motion-model from (3), no direct knowledge about the 3-D layout is available and an appropriate detection of a degenerated transform has to be performed using only the parameters of the eight-parameter motion-model.

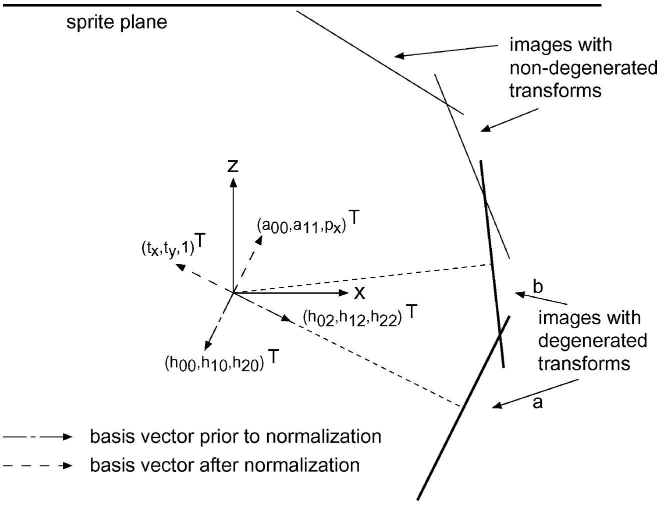

sprite plane
images with
non-degenerated
transforms
z
$(a_{00}, a_{11}, p_x)^T$
$(t_x, t_y, 1)^T$
x
$(h_{02}, h_{12}, h_{22})^T$
$(h_{00}, h_{10}, h_{20})^T$
b
images with
degenerated
transforms
a
basis vector prior to normalization
basis vector after normalization

Fig. 3. Top view onto a horizontal pan setup. Images a and b include pixels that are at the backside of the camera; their projection onto the sprite plane must be avoided. The matrix columns $(h_{0i}, h_{1i}, h_{2i})^\top$ correspond to the basis vectors of the rotated and scaled coordinate system. Since the basis vector (h_{02}, h_{12}, h_{22}) corresponds to the rotated viewing direction, a negative h_{22} indicates a rotation of more than 90 deg away from the frontal view onto the sprite. A scaled version of the basis vectors can also be found in the inhomogeneous formulation as $(a_{00}, a_{10}, p_x)^\top$, $(a_{01}, a_{11}, p_y)^\top$, $(t_x, t_y, 1)^\top$. However, because of the normalization process which sets $h_{22} = 1$, these basis vectors may swap their orientation. This is depicted for the input image a. For image b, the matrix entry h_{22} is >0 and no swapping occurs.

To derive an appropriate detection rule, let us consider again the image-formation (1). According to our assumption that the viewing direction is along the positive z-axis, the degenerated case occurs if the z-coordinate of a pixel after multiplication with the rotation matrix $\mathbf{R} = \{r_{ik}\}$ becomes ≤ 0. If $z = 0$, the point would be projected to infinity, which we also subsume into the degenerated case. Since the intrinsic camera-parameters matrix $\mathbf{K}_s$ and the shift of the image onto its focal plane by the matrix $\mathbf{K}_i^{-1}$ does not modify the sign of this value in the bottom row, the degenerated case for a specific point $(\hat{x}, \hat{y})$ can be detected with the condition

$$w' = h_{20}\hat{x} + h_{21}\hat{y} + h_{22} \leq 0. \tag{4}$$

However, since $\mathbf{H}_i$ is scaling invariant and since the motion parameters are normalized to $h_{22} = 1$ in the formulation of the eight-parameter model, the sign of all matrix entries h_{ik} can change because of the normalization, and the above test condition would be reversed to its opposite. Therefore, the condition must be modified to be invariant to the normalization.

To derive a suitable condition for the normalized parameters, we have to identify when the normalization altered the signs. Consider the case where $h_{22} < 0$ prior to the normalization process. In this case, the normalization process changes the sign of all matrix entries. Since each column of the rotation matrix represents the direction of a rotated basis vector, changing the signs of all matrix entries h_{ik} will swap the directions of all of those basis vectors. Because we assumed that the coordinate system is originally right-handed, each swap of a basis vector will change the orientation of the coordinate system, so that after the three basis vector swaps, the basis becomes left-handed. To detect this, we can observe the sign of the determinant D of the matrix of normalized parameters

$$D = \begin{vmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ p_x & p_y & 1 \end{vmatrix}. \tag{5}$$

If the determinant $D > 0$, the coordinate system is right-handed (it is not necessarily equal to unity since the length of the basis vectors is not unity), otherwise, it is left-handed. Note that the matrix entry h_{22} corresponds to the z-coordinate of the basis-vector in z direction. Since the camera looks along the z-axis, a negative h_{22}, or equivalently, $D < 0$, corresponds to a rotation of more than 90° away from the frontal viewing position, so that the camera is looking into the opposite direction.

Finally, this lets us derive the condition to decide whether a point $\mathbf{p} = (\hat{x}, \hat{y})$ is projected onto the sprite in a nondegenerated way. For this, we start with (4) using normalized parameters, obtaining the condition $p_x\hat{x} + p_y\hat{y} + 1 \leq 0$. Combining this with the sign of D leads to the final condition

$$D \cdot (p_x\hat{x} + p_y\hat{y} + 1) \quad \begin{cases} > 0, & \text{nondegenerated case} \\ \leq 0, & \text{degenerated case.} \end{cases} \tag{6}$$

To decide if an image as a whole would be mapped nondegenerated onto the sprite plane, we examine the four corner points of the image, which all must be transformed in a nondegenerated way.

V. Examples for Single-Sprite Inefficiencies

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 multisprites. However, note that the algorithm described in Section VII is not limited to these special cases, but finds the optimum solution for any real-world sequence.

A. 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 on the sprite plane is constantly increasing. At first, using a single sprite is advantageous, because most of the image was already visible in the previous image. However, when the zoom continues, the situation will eventually change, so that 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. 15).

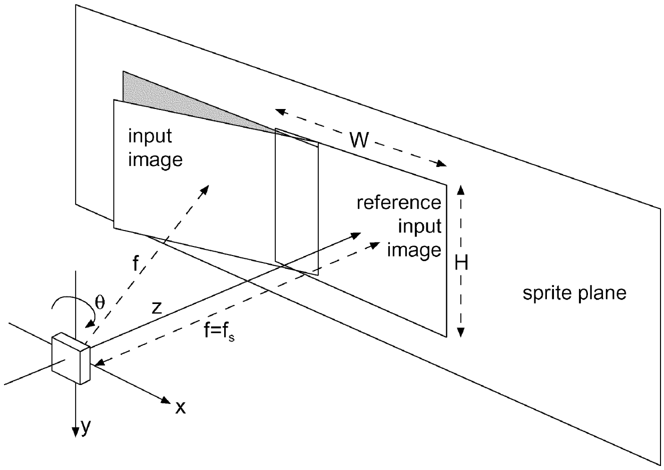


Fig. 4. Setup for the example case of horizontal camera pan.

If we denote the zoom factor between two successive frames as s and the image size as $W \times H$, the sprite size after n frames will be WHs^{2n}. Considering the alternative, in which a two-part multisprite is constructed with each sprite comprising only half of the frames, the total size of the multisprite is $2 \cdot WHs^{2n/2}$. Consequently, coding the scene as a two-part multisprite results in a lower total sprite area iff

$$WHs^{2n} > 2 \cdot WHs^{n} \quad \leftrightarrow \quad n > \log_s 2. \tag{7}$$

Generalizing this result, it is easy to derive that a p-part multisprite gives a smaller sprite area than a $(p-1)$-part multisprite, provided that the sequence length n satisfies

$$\begin{aligned} pWHs^{\frac{2n}{p}} &< (p-1)WHs^{\frac{2n}{(p-1)}} \\ \log_s p + \frac{2n}{p} &< \log_s(p-1) + \frac{2n}{p-1} \\ n &> \frac{p(p-1)}{2}\left(\log_s p - \log_s(p-1)\right). \end{aligned} \tag{8}$$

B. Example Case: Horizontal Camera Pan

Alternatively, let us now assume a camera setup where the camera only performs rotation around the vertical axis (camera pan, Fig. 4). 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 from the camera which is equal to the focal length f. 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 α. 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 the origin of the input image coordinate system is positioned at the image center, it is sufficient to consider only

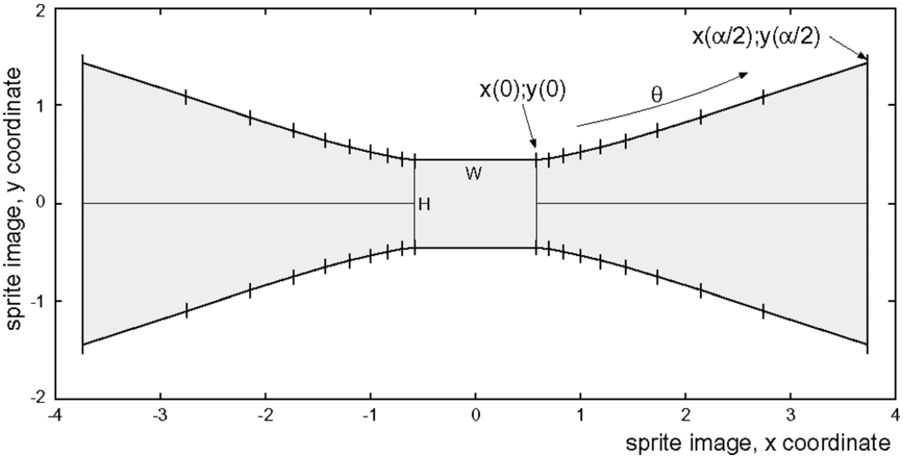


Fig. 5. Sprite area that is covered by projecting images from a rotating camera onto the sprite plane in the setup of Fig. 4. Regular intervals of camera rotation are depicted with small vertical marks along the area contour. Note that the projection area increases much faster at larger rotation angles.

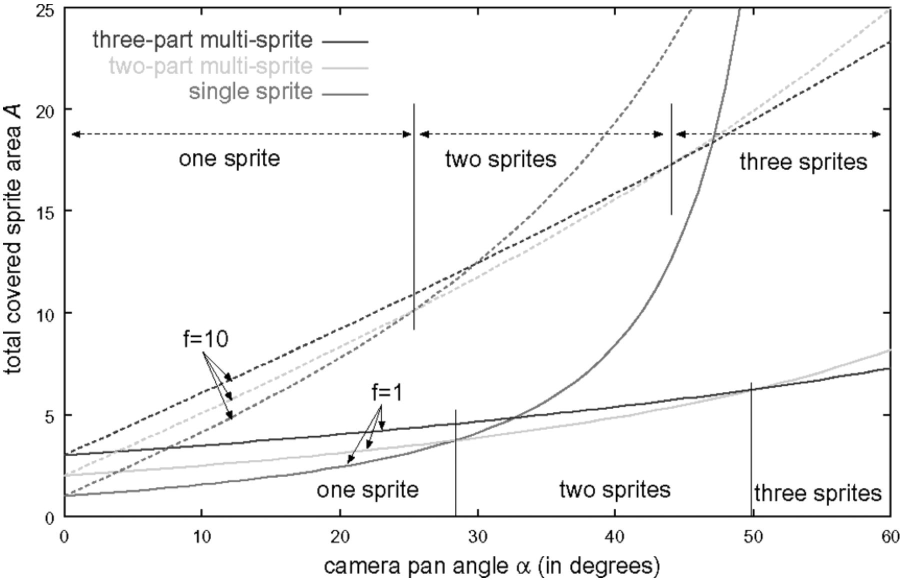


Fig. 6. Covered sprite area for horizontal camera pan at two different focal lengths. The reference image size is 1. If the camera rotation exceeds a specific angle, the area to be coded in the multisprite case is lower than for a single sprite.

one corner of the image, because the other corners can be obtained by mirroring the x and y coordinates.

Using the abbreviations $c_\theta = \cos\theta$ and $s_\theta = \sin\theta$, our camera model in this example is

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{pmatrix} \begin{pmatrix} \hat{x} = \frac{W}{2} \\ \hat{y} = \frac{H}{2} \\ \hat{z} = f \end{pmatrix}$$
$$= \begin{pmatrix} fc_\theta \frac{W}{2} + f^2 s_\theta \\ f \cdot \frac{H}{2} \\ -s_\theta \frac{W}{2} + fc_\theta \end{pmatrix}. \qquad (9)$$

Hence, if the camera is rotated by an angle θ, the top right image corner $(W/2, H/2)$ projects to (see also Fig. 5)

$$x(\theta) = \frac{fc_\theta \frac{W}{2} + f^2 s_\theta}{-s_\theta \frac{W}{2} + fc_\theta}, \quad y(\theta) = \frac{f\frac{H}{2}}{-s_\theta \frac{W}{2} + fc_\theta}. \qquad (10)$$

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

$$A(\alpha) = W \cdot H + 4 \int_{\theta=0}^{\theta=\frac{\alpha}{2}} y(\theta) \frac{\mathrm{d}x}{\mathrm{d}\theta} \mathrm{d}\theta, \qquad (11)$$

where the integral covers one of the four symmetric "*wings*" of the sprite. Fig. 6 depicts the total covered sprite area A for two different camera setups, one using $f = 1$ (wide-angle) and the other for $f = 10$ (tele). For both setups, three alternatives were examined using (11). The first one is the coding with an ordinary sprite,[4] whereas the other two alternatives are using multisprites with two or three parts. In the multisprite cases, the total pan angle was divided into equal parts and a separate sprite was generated for each part. Hence, the total sprite area that is required for an n-part sprite is $n \cdot A(\alpha/n)$. Fig. 6 depicts the

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

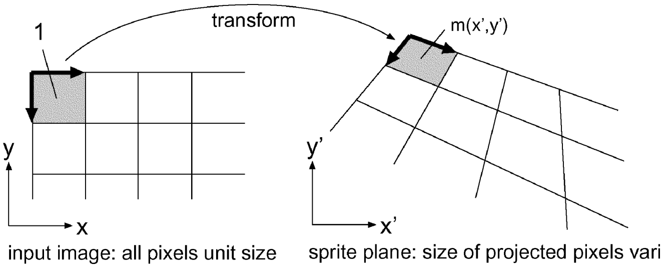

1
transform
m(x',y')
y
x
y'
x'
input image: all pixels unit size
sprite plane: size of projected pixels vari

Fig. 7. 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.

sprite area A, depending on the total pan angle α for the different setups and number of sprites used. For very low pan angles, it is clear that the ordinary sprite construction is more efficient, since the multisprite coding has the overhead of multiple transmission of mainly the same content. However, because of the fast increasing geometric distortion in the single-sprite case, the two-part multisprite becomes more efficient for pan angles over about 25° ($f = 10$). Finally, for angles exceeding approximately 45°, using a three-part sprite gets even more efficient.

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

Let us first define a magnification factor $m_l(x', y')$ that indicates for each pixel in the sprite, by which factor its size has been magnified with respect to the input image l. To prevent quality loss, $m_l(x', y')$ should always be ≥ 1 (project to the same size or larger). Obviously, this will not be the case during zoom-in sequences, but it can also be violated for rotational motion. Hence, we will now not concentrate on the pure zoom-in case only, but indicate the solution for the general case.

Because we want to ensure that $m_l(x', y') \geq 1$ for all pixels of the whole video sequence, we have to determine the minimum $m_l(x', y')$ for the whole sequence and increase the sprite resolution by the reciprocal value. Since the motion model includes perspective deformation, the scaling factor is not constant over a single input frame (see Fig. 7). The local scaling factor can be computed using the Jacobian determinant of the geometric transformation (3), which maps the input image coordinate system to the sprite coordinate system. Consequently

$$\begin{aligned} m_l(x', y') &= \begin{vmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} \end{vmatrix} \\ &= \frac{1}{D^2}\left[\begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} - \begin{vmatrix} a_{00} & a_{01} \\ p_x & p_y \end{vmatrix} y' \right. \\ &\qquad \left. + \begin{vmatrix} a_{10} & a_{11} \\ p_x & p_y \end{vmatrix} x' \right] \end{aligned} \tag{12}$$

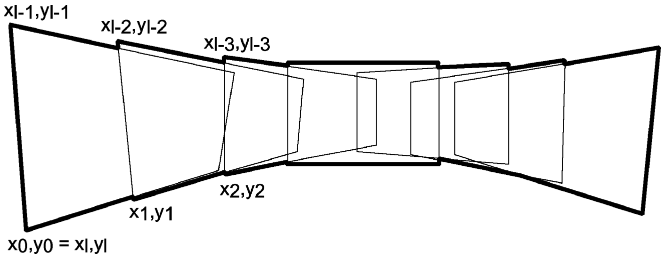


Fig. 8. Boundary polygon around the sprite area is computed as the combined outlines of all transformed quadrilaterals. For simplicity of notation, we double the last point $(x_l, y_l) = (x_0, y_0)$.

where $D = p_x x + p_y y + 1$ is the denominator of the motion model equations.[5] For nondegenerated image projections, $m_l(x', y')$ 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 $\bar{m}_l = \min_{x',y'}\{m_l(x', y')\}$ over a complete input image l, we only have to compute $m_l(x', y')$ for the four image corners and select the minimum value.

We will now consider a sprite which is built from input frame i to k. Let $\bar{m}_{i;k} = \min_{l;i \leq l \leq k} \bar{m}_l$ 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 $1/\bar{m}_{i;k}$. 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 VI-D.

VI. Sprite Cost Definitions

Optimization toward 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.

A. Bit-Stream Length

The obvious choice for defining the sprite coding cost is the bit-stream length itself. However, this definition is not practical, because of the high computational complexity required. The optimization algorithm for determining the optimal sprite arrangement (see Section VII) 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 easy to compute have to be pursued.

B. 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, (11) cannot be used since the covered area is composed of discrete projections. Instead, we describe the coded sprite area using a polygon x_i, y_i along the sprite border (see Fig. 8). Whenever an image is added to the sprite,

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

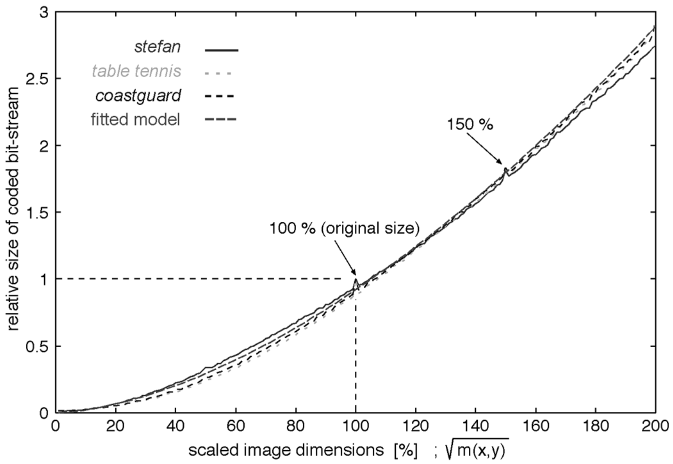


Fig. 9. Increase of bit-stream length for rescaled image resolutions. Resizing the image area by a factor $m(x, y)$ increases the bit-stream size by about $m(x, y)^{1.6/2}$, which is a bit less than a linear increase relative to the image area.

the quadrilateral of the image border is combined with the boundary polygon around the sprite to represent the new contour. The polygon area can be calculated rapidly using Green's theorem by

$$\|\cdot\|_A = \frac{1}{2} \sum_{i \in \{0;\ldots;l-1\}} x_i y_{i+1} - x_{i+1} y_i. \tag{13}$$

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 reference frame k. In order to compute the contour polygon for another reference frame i, we only have to apply $\mathbf{H}_{\mathrm{i;k}}$ to every point of the contour polygon and recompute the polygon area.

The sprite area criterion assumes that the bit-stream length is proportional to the number of coded macroblocks. 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 differing local resolution. Since the amount of image detail per block decreases when the image is magnified in the projection, the relative coding cost per block also decreases. This is not reflected with the sprite area cost definition $\|\cdot\|_A$, 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 toward making magnified areas more costly than when using the theoretically optimal *bit-stream length* cost definition of the previous section.

To determine the relationship between the resolution scaling factor and the bit-stream length, we scaled images from several sequences to different sizes and compared the bit-stream length after coding the scaled images as MPEG-4 sprite images. All the coding parameters were held constant during the experiment. The results are depicted in Fig. 9. Even though the input images have very different content, the relationships between scaling factor and increase of bit-stream length seem to be comparable. Small peaks in bit-stream size can be observed at 100%, 150%, and 50%. Since bilinear interpolation was used for the scaling, which smoothes the image a little bit, the bit rate decreases. At integer and other regular scaling factors, the pixels are sampled without effective interpolation, which explains the slightly higher bit rate. It can be seen that, as assumed, the bit-stream length does not increase linearly with the image size, but only with an exponent of 1.6/2. Up to now, we assume a simple linear relationship, but future work might try to compensate for this effect by integrating a *detail-loss factor*. However, we do not expect a significant difference since for the optimal sprite partitionings, we observed that $m(x, y)$ is close to 1 over large parts of the sprite.

C. 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 (which we will denote by $\|\cdot\|_B$) 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 ($\|\cdot\|_B$) differs not much from using the really covered sprite area ($\|\cdot\|_A$). The explanation is that an optimal multi-sprite arrangement will have as little perspective deformation as possible. Hence, the covered sprite area will be almost rectangular and obviously, the bounding box is a good approximation for almost rectangular shapes.

D. 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-in as reference. As we have described in Section V-C, this would lead to a poor quality for the decoded images at the end of the zoom sequence. 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 $\bar{m}_{i;k}$ and multiplying the area size with $\bar{m}^{-1}_{i;k}$. This correction factor reflects the potential resolution increase which is carried out in the final sprite synthesis. Note that increasing the sprite resolution by the factor $\bar{m}_{i;k}$ corresponds to shifting the sprite plane in 3-D closer to the origin ($f'_s = \bar{m}^{-0.5}_{i;k} f_s$).

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 setting the cost to 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 because the transform would be degenerated. This case is detected using the test condition derived in Section IV.

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

$$\left\|S^r_{i;k}\right\|_C = \begin{cases} \infty, & \text{if frame range } i;k \text{ cannot be projected onto a single sprite} \\ \infty, & \text{if } \|S^r_{i;k}\|_B \text{ exceeds the maximum sprite buffer size} \\ \frac{\|S^r_{i;k}\|_A}{\bar{m}_{i;k}}, & \text{else.} \end{cases} \tag{14}$$

It is easy to see that for any sensible definition of sprite coding cost, the cost is monotone for and the end of the frame range. More specifically, for a frame range $a;b$ with $a \leq i$ and $k \leq b$, it holds that $\|S_{a;b}\| \geq \|S_{i;k}\|$, since a sprite over a range $a;b$ must also contain at least the same information as the sprite constructed from every subrange $i;k$.

We use the combined cost definition $\|\cdot\|_C$ in the optimization, since it is fast to compute and it also ensures that the obtained sprite fits into the decoder sprite buffer.

VII. Multisprite Partitioning Algorithm

To find the best multisprite configuration, the algorithm has to determine the optimal range of input frames for each sprite image in the multisprite, and additionally, for each sprite the optimal reference frame.

The multisprite 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 best reference coordinate system for each of these frame ranges by selecting that input frame as a reference, for which the sprite area for this frame range would be smallest. The second step partitions the complete input sequence into frame ranges, such that the total sprite coding cost is minimized.

A. 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 consider the cost $\|S^r_{i;k}\|$ for all reference frame placements r with $i \leq r \leq k$. Since we can choose the optimal reference frame for each of the sprite ranges independently, we select the placement for which the sprite cost is lowest. The sprite cost for optimal placement of the reference is denoted with

$$\left\|S^*_{i;k}\right\| = \min_r \left\|S^r_{i;k}\right\|. \tag{15}$$

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, it is assumed for simplicity that the cost definition is based on the sprite bounding box, but the same principle can also be applied to the area computation.

We begin with computing all bounding boxes for the case that the first frame in a range is selected as reference frame $(S^r_{r;k})$. These costs can be computed efficiently for all k 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 $S^r_{i;r}$. When both directions are processed, 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. 10). Let us denote the computation of the enclosing bounding box of two sprites $S^r_{i;r}$ and $S^r_{r;k}$ as $\|S^r_{i;r}; S^r_{r;k}\| = \|S^r_{i;k}\|$. The difference to using the sprite size $\|S^r_{i;k}\|$ directly is that this would require a look-up in a 3-D array of precomputed cost-values. By splitting the sprite range into two parts, namely, the range preceding the reference frame and the remaining range after the reference frame, precomputed sprite costs can be determined with lower memory requirements, since only two triangular matrices are stored.

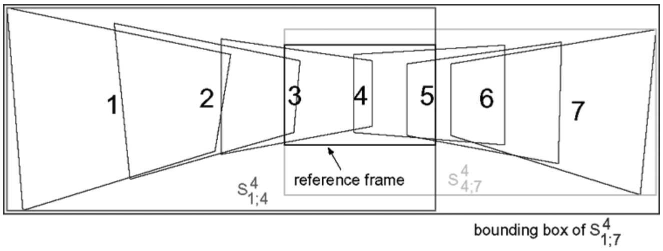


Fig. 10. Sprite for frames 1–7 with frame 4 as the reference frame. In this case, the bounding-box for $S^4_{1;7}$ has been computed by combining the bounding boxes of $S^4_{1;4}$ and $S^4_{4;7}$.

Consequently, when we determine $\|S^*_{i;k}\|$ by searching for the r that results in the minimum area bounding box, we do not use (15) directly, but combine the cost using the two sprite halves as

$$\left\|S^*_{i;k}\right\| = \min_r \left\|S^r_{i;r}; S^r_{r;k}\right\|. \tag{16}$$

The 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 subsequent optimization algorithm. Additionally, we store the reference frame r for each $S^*_{i;k}$ as it was found in the minimization (16). This value is not needed for the optimization, but the final sprite image generation uses the information for selecting the reference coordinate system.

B. Optimal Sequence Partitioning

In the sequence partitioning step, the input frames are divided into separate ranges, so that the total cost to code the sprites for all the frame ranges is minimal. More formally, let $P = ((1, p_1 - 1), (p_1, p_2 - 1), \ldots, (p_{n-1}, N))$ be a partitioning of the video sequence of length N into n subsequences. The optimization problem can then be formulated as determining the partitioning P^* for which the sum of all sprite costs is minimal

$$P^* = \arg\min_P \sum_{(i,k) \in P} \left\|S^*_{i;k}\right\|. \tag{17}$$

This minimization problem can actually be viewed as a minimum-cost path search in a graph, where the graph nodes correspond to the input frames plus an additional dummy start node, $V = \{0, \ldots, N\}$. The graph is fully connected with directed edges $E = \{(i;k) | i, k \in V; i < k\}$. Each edge $(i;k)$ is attributed with edge costs $\|S^*_{i+1;k}\|$. Every path from the start node 0 to N defines a possible partitioning, where each edge

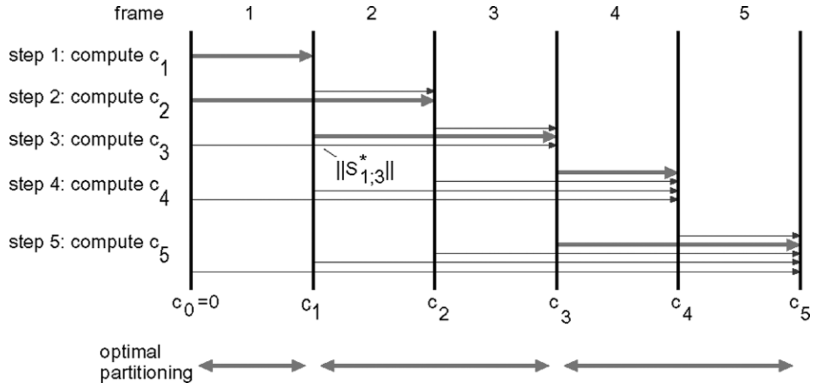


Fig. 11. Determining the optimal sequence partitioning. Each state c_k is assigned the minimum coding cost for a partitioning ending in frame k. Each arrow represents the cost for the sprite built from the covered frames. For each c_k with $k \geq 1$, the sprite that results in the minimum cost in node 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.

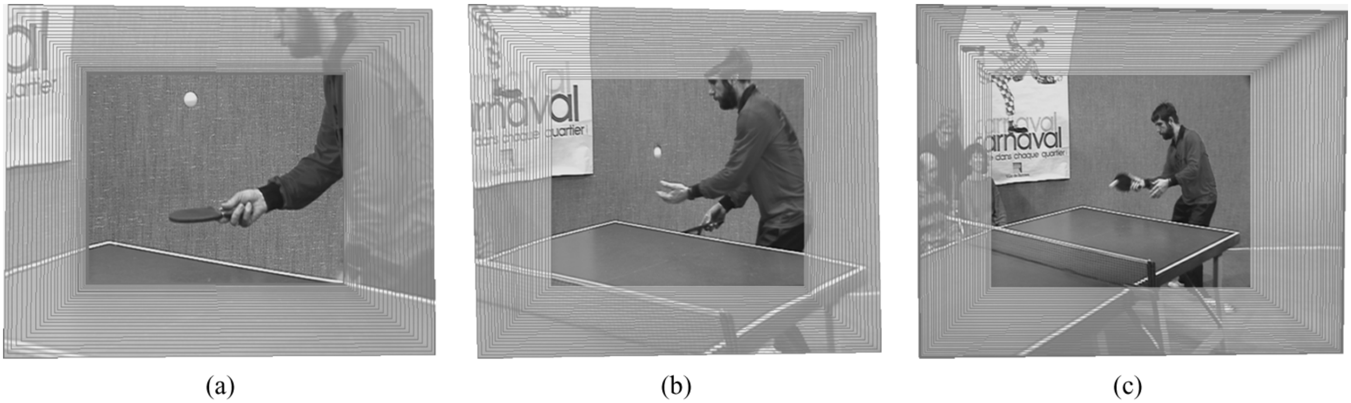


Fig. 12. Multisprite synthesized from a long zoom-out operation. The sequence is partitioned into three separate sprites of almost the same size. The input image selected as reference is shown in a darker shade. A single-part sprite generated from the same sequence has a size of 1687 × 1516. (a) Frames 1–51, 603 × 500. (b) Frames 52–77, 587 × 501. (c) Frames 78–132, 585 × 478.

on the path corresponds to one frame range for which a sprite is generated. Consequently, the minimum cost path gives the minimum-cost partitioning P^*. The shortest path search can be carried out using a standard Dijkstra algorithm or A^* search.

However, because of the regular graph structure, the minimization problem can also be computed with a simple iterative algorithm (Fig. 11). For each image i, we compute the minimum cost $c_k (c_0 = 0)$ of a partitioning ending in image k as

$$c_k = \min_{i \in [1,k]} \left\{ c_{i-1} + \left\| S^*_{i;k} \right\| \right\}. \tag{18}$$

The index i denotes the beginning of the last subsequence 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.

When searching through the possible values of i in (18), a common case is that the sprite cost $\|S^*_{i;k}\|$ will reach ∞ when a cost definition according to (14) is used. As the cost cannot decrease if the frame range is extended (see Section VI-D), an efficient way is to carry out the search for i backward, starting with k and stopping the search early if ∞ is obtained for the sprite cost $\|S^*_{i;k}\|$.

VIII. Experiments and Results

We have implemented the algorithm with the sprite cost definition of Section VI-D. This section describes the algorithm results for the three sequences *Table-tennis*, *Rail*, and *Stefan*. The sequences *Table-tennis* and *Stefan* are well-known test sequences, whereas the *Rail* sequence was recorded from a public DVB broadcast. In Figs. 12–17, we indicate the frame range which was used to generate each sprite, the bounding-box size, and the covered sprite area in 1000-pixel units. The obtained sprite sizes are also summarized in Table I.

From the *Table-tennis* sequence, the first camera shot consisting of 132 frames has been selected. This camera 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 multisprite (Fig. 12). Compared with the size of an ordinary single-part sprite, the area of the multisprite 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.

The *Rail* sequence (Fig. 13) contains a complicated camera rotation. It starts with the camera looking downwards and continues with the camera rotating to the left and around its optical

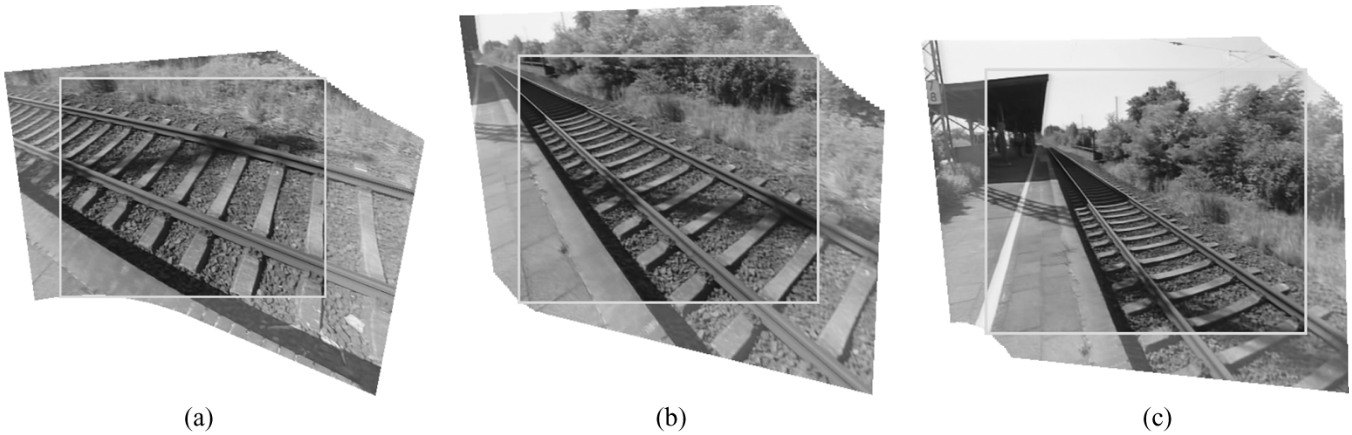


Fig. 13. Multisprite for the *Rail* sequence. The sequence shows a camera rotation around two axes at the same time. At the beginning of the sequence, the camera is looking down. It turns left and around its optical axis until it looks left in the last frame. Indicated for each sprite is the size of the bounding-box and the covered sprite area in 1000-pixel units. Reference frames are depicted in a darker shade. (a) 1–45, 541 × 445, area: 170 k. (b) 46–82, 479 × 446, area: 171 k. (c) 83–140, 455 × 387, area: 152 k.

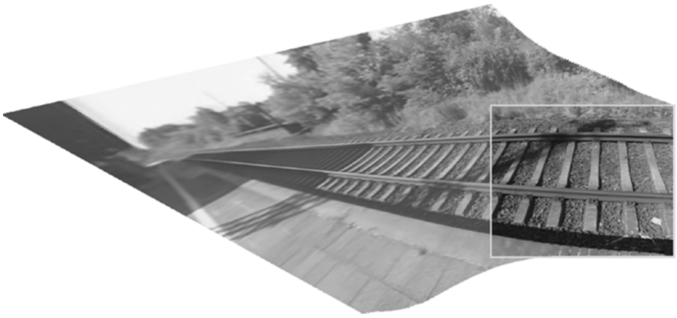


Fig. 14. .Frames 1–82 integrated into a single background sprite (1264 × 592, area: 427 k). The attempt to integrate the entire *Rail* sequence into a single background sprite fails because of the complicated camera motion. The camera performs an approximate 90° rotation around two axes.

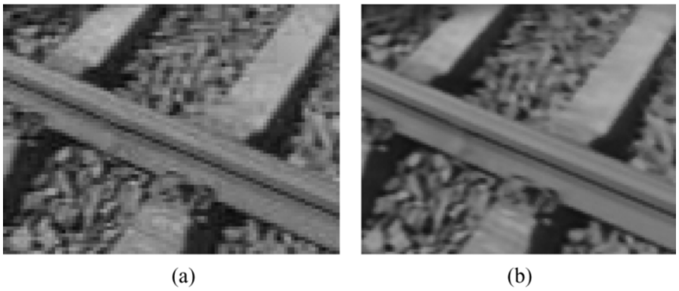


Fig. 15. Super-resolution effect. Since many input frames are integrated in the background synthesis step based on an accurate motion-model, a high-resolution image can be derived from a sequence of low-resolution images. (a) Input. (b) Sprite reconstruction.

axis at the same time. At the end, the camera is looking to the left side. Integration of the complete sequence into a single sprite leads to a very strong deformation of the input frames which makes the conventional approach rather impractical (Fig. 14). Applying the multisprite algorithm to the sequence results in a three-part multisprite, where each of the sprites shows only little perspective deformations.

Since the *Rail* sequence does not contain foreground objects, it was possible to measure the quality of the sprite reconstruction compared to the input sequence. We measured the reconstruction alone on uncompressed sprites by synthesizing the sprites from the input sequence and then applying the global-motion compensation on the sprites to reconstruct the input sequence again. The measurements were carried out using three different types of sprite construction: multisprite coding with integration of the scale-factor $\bar{m}_{i;k}$, without the scale-factor, and a heuristic sprite partitioning. In the heuristic sprite partitioning, the sprite was built iteratively until the sprite width exceeded a threshold. The threshold was chosen such that the first sprite covers frame 1–82 (see Fig. 14), which equals the frame range of the first two sprites obtained from the multisprite partitioning. Fig. 18 depicts the reconstruction quality of the different approaches. Apart from the fact that the multisprite reconstruction clearly outperforms the single-sprite reconstruction by about 1 dB, it can also be seen that the integration of the scaling factor in fact increases the reconstruction quality in the last part of the sequence.

Since the reconstruction from the sprite is always based on the static sprite, whereas the input is a moving image sequence, variations in the image apart from camera motion cannot be reconstructed from the sprite. Even if there are no perceivably moving objects in the sequence, the input images can still vary, e.g., because of motion-blur during a fast camera pan. Moreover, the camera optic can also deform the image by radial lens distortion, which cannot be represented in the sprite. Hence, it is clear that the sprite reconstruction cannot be perfect. On the other hand, since many input frames are combined when synthesizing the background sprite, a super-resolution effect occurs, so that the amount of detail in the sprite is even higher than in the original video. This can be observed in Fig. 15, which shows a magnification of part of the Fig. 13(a). Since the input was originally MPEG-2 compressed, it shows some noise, which is not present in the sprite reconstruction. Furthermore, clearly more detail is visible in the sprite reconstruction. Consequently, a decrease in PSNR compared to the input does not necessarily correspond to a reduction of perceived quality.

For the MPEG-4 sequence *Stefan*, we first attempted to generated an ordinary sprite image for the complete 300 frames. However, because the total viewing angle during the sequence

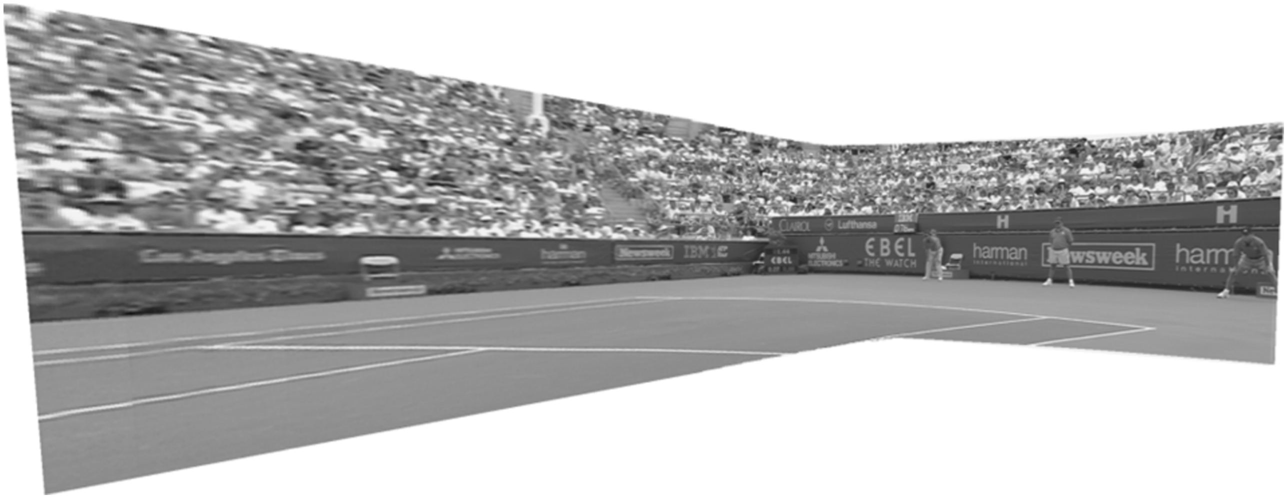
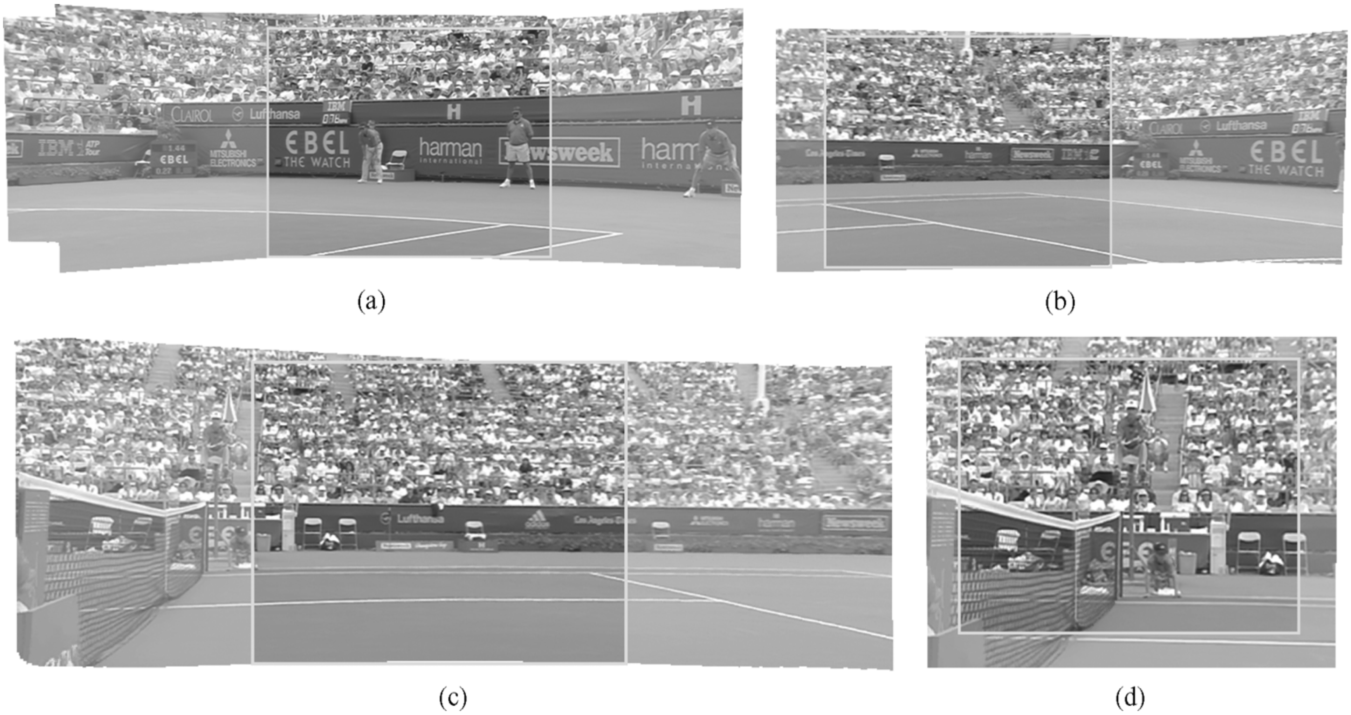


Fig. 16. Sprite synthesized from the *Stefan* sequence. Only the first 255 frames can be used since it is impossible to create the sprite for the complete sequence if the first frame is selected as reference. Sprite resolution is 2445 × 1026 pixels, area: 1208 k.



(a) (b) (c) (d)

Fig. 17. Multisprites synthesized using the described algorithm. The respective reference frames are depicted in a darker shade. Note that the long camera pan is partitioned 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). (a) 1–240, 926 × 339, area: 272 k. (b) 241–255, 699 × 296, area: 195 k. (c) 256–292, 830 × 318, area: 233 k. (d) 293–300, 431 × 350, area: 141 k

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 Fig. 16. Applying the multisprite algorithm on the complete sequence resulted in a four-part multisprite, which is shown in Fig. 17. We have measured that the total required sprite size for the multiple-sprite approach is a factor of 2.6 smaller than for the single-sprite case. However, note that the multisprite covers 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. 17(d)]. 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 can be coded with a lower resolution, which outweighs the overhead of an additional sprite.

TABLE I
COMPARISON OF SPRITE SIZES USING SINGLE SPRITES AND THE MULTISPRITE APPROACH. THE AREA OF THE BOUNDING BOX AND THE COVERED SPRITE AREA ARE GIVEN IN UNITS OF 1000 PIXELS

	single sprite		multi sprite	
sequence	bounding box	covered area	b. box	area
Table-T. (1–132)	2557k (292%)	2540k (295%)	875k	860k
Rail	N/A	N/A	630k	492k
Rail (1–82)	748k (152%)	427k (125%)	493k	340k
Stefan	N/A	N/A	936k	841k
Stefan (1–255)	2509k (481%)	1208k (264%)	521k	457k

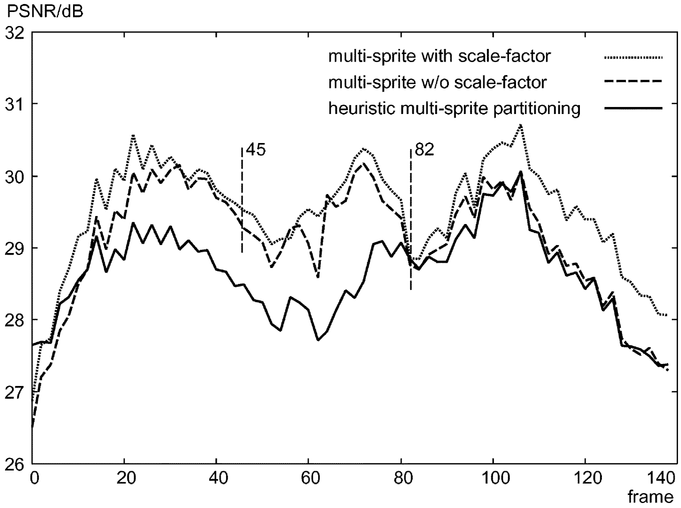


Fig. 18. Comparison of sprite reconstruction quality for the *Rail* sequence (depicted in Fig. 16).

IX. INTEGRATION INTO AN OBJECT-ORIENTED VIDEO CODING SYSTEM

The described multisprite partitioning algorithm can be integrated easily into a framework for video-object segmentation. An overview of our multisprite based segmentation system is depicted in Fig. 19. Processing starts with a feature-based global-motion estimator [5]. This estimator identifies prominent corner features in each input frame and establishes correspondences between matching features. The parameters of the projective motion-model are then estimated from the displacements of these features. To differentiate between foreground and background motion, a robust regression algorithm based on the RANSAC algorithm [11] is employed. The advantage of a feature-based estimator is its robustness against fast camera motion or illumination changes. Its disadvantage, however, is the insufficient estimation accuracy for building the background sprite directly from the estimated motion parameters.

The motion parameters from the feature-based estimator are used in the subsequent multisprite partitioning step to determine the frame ranges that will be used for each of the background sprites.

Starting with the reference frames, the next step refines the motion parameters by carrying out a direct motion estimation based on a gradient-descent search [12]. Since the input frames are aligned to the reconstructed mosaic, no accumulation of estimation errors can occur. Hence, the estimation accuracy is superior to the feature-based approach in the first step, where motion was computed between successive frames and where errors could accumulate in the concatenation of transforms. Since the multisprite partitioning ensures that no degenerated transforms occur within a sprite, this long-term prediction is possible.

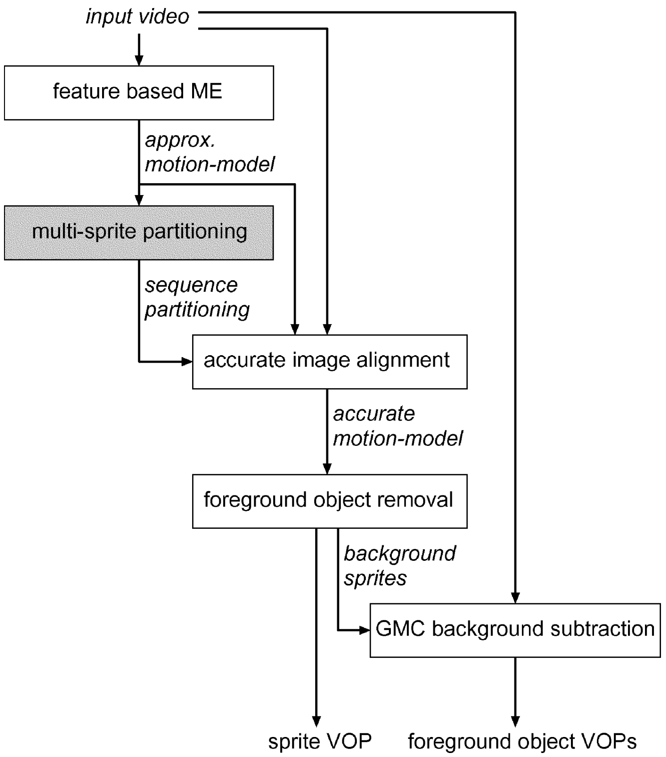


Fig. 19. Framework of our complete video-object segmentation system including the multisprite partitioning algorithm.

The *foreground object removal* step synthesizes a virtual background image without the foreground objects using the algorithm described in [13]. The algorithm is an extension of pixel-wise temporal median filtering though the stack of motion-compensated images of the considered frame range. If foreground objects are located near the boundary of the sprite area at the beginning or end of the considered frame range, the background reconstruction algorithm would not be able to remove the foreground objects, since there is too little information available for these sprite areas. Therefore, we extend the frame range, which is used to reconstruct the background, to include as much information as possible (see Fig. 20).

The background sprites obtained from the last step can be used in a standard MPEG-4 encoder. To transmit the multisprite, two approaches are possible. The sprites can be transmitted sequentially, where a new sprite must be sent just in time to show a continuous video at the decoder. However, this requires a high peak data-rate to send the new sprite. The second approach is to compose the individual sprites of the multisprite into a single sprite image, where the sprites are placed independently beneath each other. The motion parameters can be modified such that the correct sprite image is decoded from the sprite buffer. The second approach does not require a high peak data-rate, but needs more decoder memory for the sprite buffer. Future work

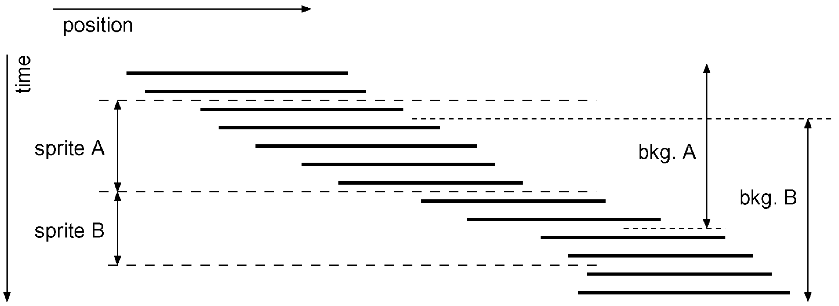


Fig. 20. Determining the frame ranges for background synthesis. The video is depicted as a stack of motion-compensated frames. To obtain an optimal suppression of foreground objects, the frame range of each sprite is extended to include all frames that overlap with the sprite area.

might combine the multisprite partitioning with optimized decoder-buffer management, such that the decoder buffer for example always contains two sprites, where one is displayed while the other is updated.

In our system, foreground objects are obtained using a background subtraction technique [14], [15]. Objects are detected by a high difference between the input frame and the motion-compensated background sprite. In this step, the same multisprite approach is used to cover the entire input sequence with pure background images.

X. Conclusions

This paper has shown that partitioning a background sprite into several independent parts results in clearly reduced coding cost and better image quality at the same time. Our algorithm computes the optimal partitioning of a sequence, the reference frame for each partition, and associated scaling factors. As a consequence, the proposed algorithm solves the subsequent problems. It removes the limitations of camera motion and enables to use sprite coding for arbitrary rotational camera motion. It selects optimal reference frames and defines multisprite partitions to considerably reduce the required amount of data for coding the background sprite. Finally, it ensures that no quality degradation occurs during camera-zoom operations, thereby increasing the reconstruction quality of the sprite. All this is achieved while remaining compatible to the MPEG-4 standard.

Clearly, the reduction of sprite area depends on the type of camera motion in the sequence, e.g., for the *Stefan* sequence, a reduction by a factor of at least 2.6 has been achieved. Moreover, note that the proposed algorithm can synthesize sprites for all kinds of camera motion, which cannot be achieved with previous approaches. This presents a generalization that is not only important for the coding of background sprites, but also for other image analysis algorithms like a video-object segmentation based on background subtraction. These algorithms also require a complete coverage of the background environment with a set of background images.

References

[1] H. Watanabe and K. Jinzenji, “Sprite coding in object-based video coding standard: MPEG-4,” in *Proc. World Multiconf. SCI 2001*, 2001, vol. XIII, pp. 420–425.

[2] S.-Y. Chien, C.-Y. Chen, Y.-W. Huang, and L.-G. Chen, “Multiple sprites and frame skipping techniques for sprite generation with high subjective quality and fast speed,” in *Proc. IEEE Int. Conf. Multimedia and Expo (ICME)*, 2002, pp. 785–788.

[3] 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. Boca Raton, FL: CRC, Sep. 2003, pp. 561–591.

[4] A. Smolic and J. Ohm, “Robust global motion estimation using a simplified M-estimator approach,” in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, 2000, pp. 868–871.

[5] D. Farin and P. H. N. de With, “Evaluation of a feature-based global-motion estimation system,” *SPIE Vis. Commun. Image Process.*, pp. 1331–1342, Jul. 2005.

[6] M. Massey and W. Bender, “Salient stills: process and practice,” *IBM Syst. J.*, vol. 35, no. 3 & 4, pp. 557–573, 1996.

[7] R. Hartley and A. Zisserman*, Multiple View Geometry in Computer Vision*. Cambridge, U.K.: Cambridge Univ. Press, 2000.

[8] 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, Symp.*, Apr. 1999, pp. 239–242.

[9] Y. Lu, W. Gao, and F. Wu, “Efficient background video coding with static sprite generation and arbitrary-shape spatial prediction techniques,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 5, pp. 394–405, May 2003.

[10] R. Szeliski and H.-Y. Shum, “Creating full view panoramic image mosaics and environment maps,” *Comput. Graph.*, vol. 31, pp. 251–258, 1997.

[11] M. A. Fischler and R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography,” *Commun. ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[12] R. Szeliski, Image Mosaicing for Tele-Reality Apllications Digital Equipment Corp., Cambridge Res. Lab, May 1994, Tech. Rep. CRL 94/2.

[13] D. Farin, P. H. N. de With, and W. Effelsberg, “Robust background estimation for complex video sequences,” in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, 2003, vol. 1, pp. 145–148.

[14] T. Aach and A. Kaup, “Bayesian algorithms for adaptive change detection in image sequences using markov random fields,” *Signal Process.: Image Commun.*, vol. 7, pp. 147–160, 1995.

[15] D. Farin and P. H. N. de With, “Misregistration errors in change detection algorithms and how to avoid them,” in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2005, vol. 2, pp. 438–441.

[16] 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. Circuits Syst. Video Technol.*, vol. 7, no. 1, pp. 130–145, Feb. 1997.

[17] F. Dufaux and F. Moscheni, “Background mosaicking for low bit rate video coding,” in *Proc. IEEE Int. Conf. Image Process.*, 1996, vol. 1, pp. 673–676.

[18] H. Nicolas, “Optimal criterion for dynamic mosaicking,” in *Proc. IEEE Int. Conf. Image Process.*, Oct. 1999, vol. 4, pp. 133–137.

[19] K. Jinzenji, H. Watanabe, S. Okada, and N. Kobayashi, “MPEG-4 very low bit-rate video compression using sprite coding,” in *Proc. IEEE Int. Conf. Multimedia Expo*, Aug. 2001, pp. 2–5.

[20] P. S. Heckbert, “Fundamentals of texture mapping and image warping,” M.S. thesis, Dept. Elect. Eng. Comp. Sci., Univ. California, Berkeley, Jun. 1989.

[21] M. Kourogi, T. Kurata, J. Hoshino, and Y. Muraoka, “Real-time image mosaicing from a video sequence,” in *Proc. IEEE Int. Conf. Image Process.*, Oct. 1999, vol. 4, pp. 133–137.

Dirk Farin (M'06) received the Dipl.-Inf. degree in computer science from the University of Stuttgart, Stuttgart, Germany, in 1999, and the Ph.D. degree from the University of Mannheim, Mannheim, Germany, for his work on automatic video-object segmentation and object modeling.

After graduating from the University of Stuttgart, he was Research Assistant at the Department of Circuitry and Simulation, University of Mannheim, where he started his research on video-object segmentation. He joined the Department of Computer Science IV, University of Mannheim in 2001. Since 2004, he has been with the Video Coding and Architecture Group, Technical University of Eindhoven, Eindhoven, The Netherlands. Apart from video-object segmentation, his research interests include video compression, content analysis, and 3-D reconstruction. Currently, he is involved in a joint project of Philips and the Technical University of Eindhoven about the development of video capturing and compression systems for 3-D television. He developed popular open-source and commercial software including an MPEG-2 decoder, two MPEG-2 encoders, libraries with computer-vision algorithms, and image-format conversion software.

Dr. Farin received a Best Student Paper Award at the SPIE Visual Communications and Image Processing Conference in 2004 for his work on multisprites, and two Best Student Paper Awards at the Symposium on Information Theory in the Benelux in 2001 and 2003. In 2005, he organized a special session about sports-video analysis at the IEEE International Conference on Multimedia and Expo. He is member of the program committee of the IEEE International Conference on Image Processing and Reviewer for several journals including IEEE MULTIMEDIA, IEEE IMAGE PROCESSING, and IEEE CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY.

Peter H. N. de With (M'81–SM'97) received the M.Sc. degree in electrical engineering from the University of Technology, Eindhoven, The Netherlands, in 1984 and the Ph.D. degree from the University of Technology, Delft, The Netherlands, in 1992, for his work on video bit-rate reduction for recording applications.

He joined Philips Research Laboratories, Eindhoven, The Netherlands, in 1984, where he became a member of the Magnetic Recording Systems Department. From 1985 to 1993, he was involved in several European projects on SDTV and HDTV recording. In this period, he contributed as a coding expert to the DV standardization. In 1994, he became a member of the TV Systems group, where he was leading the design of advanced programmable video architectures. In 1996, he became Senior TV Systems Architect and in 1997, he was appointed as Full Professor at the University of Mannheim, Mannheim, Germany, in the faculty of computer engineering. In 2000, he joined CMG Eindhoven as a principal consultant and he became professor at the University of Technology Eindhoven, in the faculty of electrical engineering. He has written numerous papers on video coding, architectures and their realization. Regularly, he is a Teacher of the Philips Technical Training Centre and for other post-academic courses.

In 1995 and 2000, Dr. de With co-authored papers that received the IEEE CES Transactions Paper Award. In 1996, he obtained a company Invention Award. In 1997, Philips received the ITVA Award for its contributions to the DV standard. He is a program committee member of the IEEE CES and IEEE ICIP, chairman of the Benelux Working Group on Information Theory, member of the former scientific board of CMG, ASCI, and various other working groups.