The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny ...'
(Isaac Asimov)

Chapter 6

Multi-Sprite Backgrounds

The previous three chapters presented an algorithm to reconstruct a scene background image from a video sequence. In the MPEG-4 standard, this background image (sprite) can be coded and transmitted independently from the foreground objects. This separation saves bandwidth when the background sprite is sent less often or only once, and the image area that has to be coded comprises only the foreground objects. Whereas it seems optimal to combine as many images into one background sprite as possible, we have found that the counter-intuitive approach of splitting the background into several independent parts can reduce the overall amount of data needed to transmit the background sprite. Furthermore, we show that in the general case, the synthesis of a single background sprite is even impossible and that the scene background should 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 multi-sprite*), resulting in a significant reduction of the involved coding cost. The generated multi-sprite backgrounds are a generalization of the previously discussed background reconstruction algorithm, with the main difference that several background images are used throughout the sequence.*

6.1 Introduction

The background image that we reconstructed with the algorithms of the previous chapters serves two purposes. First, we need the pure background images for the foreground-object segmentation algorithm that is presented in the next chapter. But we can also use the obtained background images and motion parameters directly as input for an MPEG-4 encoder that supports sprite coding. The concept of sprite coding is that the static sprite image is reused for the decoding of many frames, where each output image shows a partial view of the sprite image. One main advantage of using sprite coding is to reduce the required bandwidth, since the background areas are only sent once with a common sprite image [194].

In this chapter, we take a closer look at the coding efficiency of sprites and relate this to the image formation geometry and the camera motion. 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 applying the projective motion model, which is used in 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 for camera zoom-in sequences.

In [25], coding with multiple sprites was proposed to reduce the distortions in a sprite. However, observed distortions 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.

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. Even for small rotation angles, sprite-coding can be inefficient, because 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.

Another open point, which we ignored in the last chapter, is the selection of a reference coordinate system. The usual approach to date is to use the first frame of the input sequence as the reference frame. Alternatively,

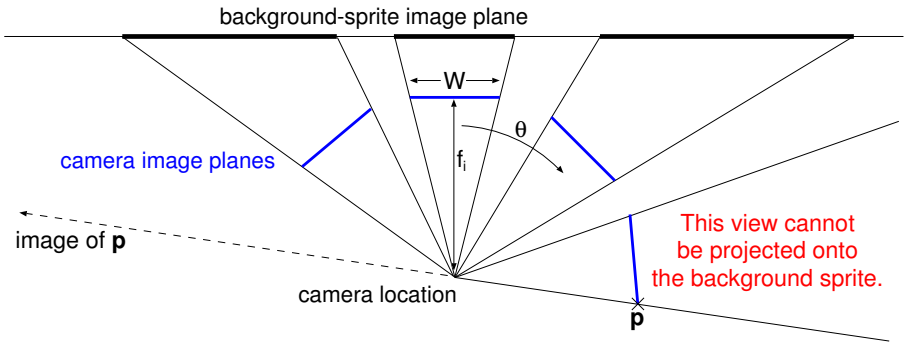
Massey and Bender [123] propose to use the middle frame of a sequence, which results in a more symmetric sprite shape if the camera performs a continuous panning motion. Instead of using a heuristic reference frame placement, our algorithm also computes the optimal reference frame to minimize the synthesized 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, it means that the sprite area that is covered by the image is smaller. If we do not want to lose the increased resolution of the input, it means that 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 unfavourable loss of resolution, our sprite generation algorithm can incorporate a constraint that ensures that the resolution of no input frame is 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.

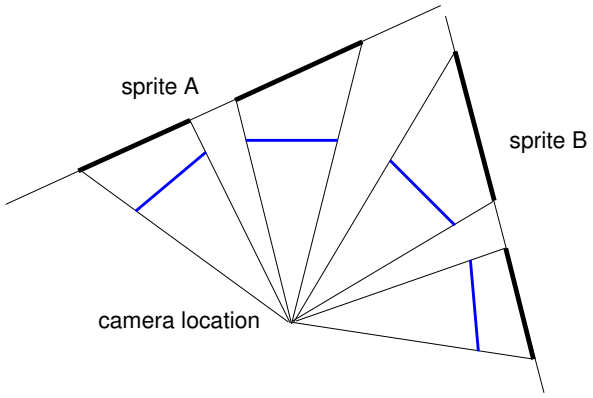
The remainder of the chapter is structured as follows. Section 6.2 reveals limitations of the MPEG-4 sprite model and introduces the concept of multi-sprites. Section 6.3 derives a classification method to detect camera configurations for which no appropriate projective transform onto a sprite plane exists. In Section 6.4, the three idealized examples of pure camera zoom-out, zoom-in, and camera rotation are analyzed. It will be shown theoretically 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 6.5 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 6.6, while experimental results are presented in Section 6.7. An overview how the algorithm can be integrated into a video-object segmentation framework is given in Section 6.8, and we discuss briefly how multi-sprites can be transmitted in standard MPEG-4 sprite VOPs in Section 6.10.

6.2 Limitations of the single-sprite approach

MPEG-4 sprite coding is based on the previously described projective motion model, which is defined as Eq. (2.12). In geometric terms, this transformation 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



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



(b) Multi-sprites reduce the deformations.

Figure 6.1: *(a) Top-view of projecting the input frames onto the 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 deformation and allows coverage of larger viewing angles.*

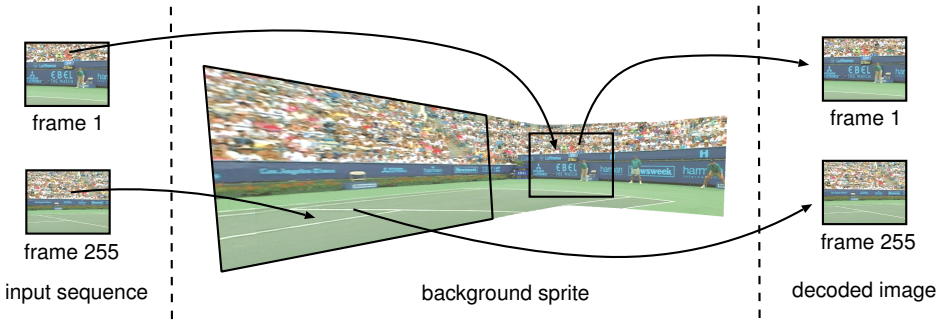


Figure 6.2: *MPEG-4 sprite coding is inefficient for large camera rotation angles. Frame 255 covers a much larger area in the sprite than in the original sequence. Hence, a magnified view is transmitted, but only the low resolution is displayed. Note that the reference frame 1 has the same size in the sprite as in the original sequence.*

in Fig. 6.1(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 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 [118, 119], 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 (Fig. 6.2).

An alternative representation for background images would be to use spherical or cylindrical image mosaics [180] for the background image, instead of a planar mapping (Fig. 6.3). However, this approach has several disadvantages compared to using the projective motion model. First, generation of spherical/cylindrical mosaics requires that the internal camera pa-

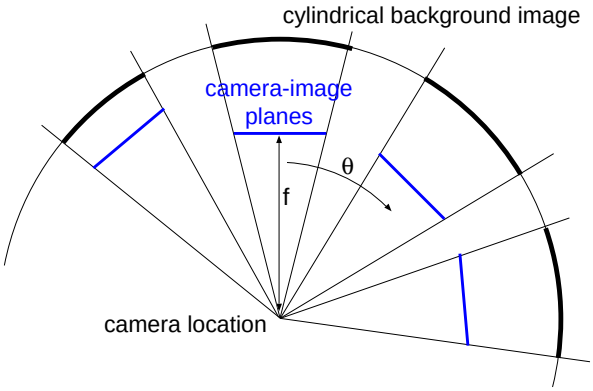


Figure 6.3: *Using a cylindrical background model.*

rameters 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 (see Chapter 12). Furthermore, the estimation of the transformation parameters for cylindrical and spherical mosaics requires complicated non-linear optimization techniques, and the reconstruction at the decoder is computationally expensive, because transcendental functions are required. Finally, and above all, 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 chapter, 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 perspective deformations do not accumulate much in the multi-sprite case, so that larger parts of the sprite can be transmitted in a lower resolution. Figure 6.1(b) depicts the same scene as in Fig. 6.1(a), 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 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 multi-sprite generation finds the optimal partitioning of a video sequence into multi-sprites 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 resolution-preservation constraint, which prevents loss of detail during camera zoom-in operations.

6.3 Detecting degenerated transforms

As we have seen in Figure 5.5, the projective transform maps image positions 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 Figure 6.1(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 Figure 6.4). 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 appear. 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 Eq. (2.12), no direct knowledge about the three-dimensional 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.

To derive an appropriate detection rule, let us consider again the image-formation Equation (5.1). 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 zero or negative. 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

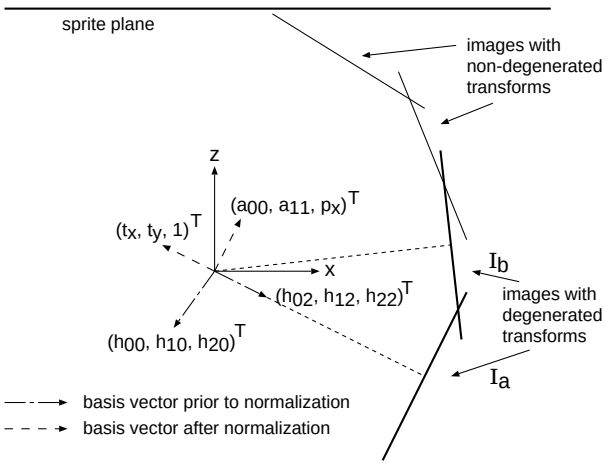


Figure 6.4: *Top view of a horizontal pan set-up. Images* a *and* b *include pixels that are at the back-side of the camera; their projection onto the sprite-plane must be avoided. The matrix columns* $(h_{0i}, h_{1i}, h_{2i})^\top$ *correspond to the basis vectors of the rotated and scaled coordinate 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* $90°$ *degrees 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* I_a. *For image* I_b, *the matrix entry* h_{22} *is* > 0 *and no swapping occurs.*

its focal plane by the matrix $\mathbf{K}_i^{-1}$ does not modify the sign of this value, the degenerated case for a specific point $(\hat{x}, \hat{y})$ can be detected with the condition

$$w' = h_{20}\hat{x} + h_{21}\hat{y} + h_{22} \leq 0. \tag{6.1}$$

However, since $\mathbf{H}_i$ is scaling invariant and 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 now left-handed. To detect this, we can observe the sign of the determinant D of the matrix of normalized parameters

$$D = \begin{vmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ p_x & p_y & 1 \end{vmatrix}. \tag{6.2}$$

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 non-degenerated way. For this, we start with Eq. (6.1) 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

$$D \cdot (p_x\hat{x} + p_y\hat{y} + 1) \quad \begin{cases} > 0 & \text{non-degenerated case,} \\ \leq 0 & \text{degenerated case.} \end{cases} \tag{6.3}$$

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

6.4 Examples of single-sprite inefficiencies

Let us first describe some idealized examples to clarify why the 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 6.6 is not limited to these special cases, but finds the optimum solution for any real-world sequence.

6.4.1 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 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 the 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 Figure 6.17).

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 multi-sprite is constructed with each sprite comprising only half of the frames, the total size of the multi-sprite is $2 \cdot WHs^{2n/2}$. Consequently, coding the scene as a two-part multi-sprite results in a lower total sprite area iff

$$WHs^{2n} \;>\; 2 \cdot WHs^{n} \quad \leftrightarrow \quad n > \log_s 2. \tag{6.4}$$

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

$$\begin{aligned} pWHs^{2n/p} &< (p-1)WHs^{2n/(p-1)} \\ \log_s p + \frac{2n}{p} &< \log_s(p-1) + \frac{2n}{p-1} \\ n \;&>\; \frac{p(p-1)}{2}\big(\log_s p - \log_s(p-1)\big). \end{aligned} \tag{6.5}$$

6.4.2 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. 6.5). Input images are assumed to have normalized size $W \cdot H = 1$ and the 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

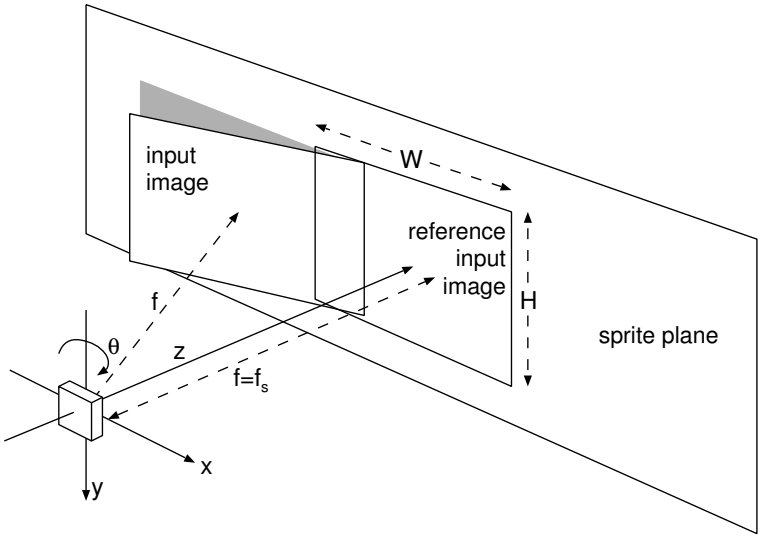


Figure 6.5: *Set-up for the example case of horizontal camera pan.*

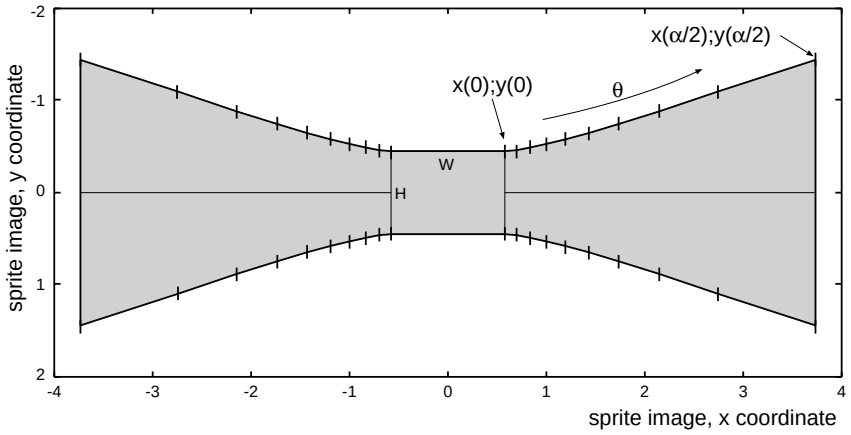


Figure 6.6: *The sprite area that is covered by projecting images from a rotating camera onto the sprite plane in the set-up of Figure 6.5. Regular intervals of camera rotation are depicted with small vertical marks along the area contour. Note that the projection area increases much faster at larger rotation angles.*

at the image center, it is sufficient to consider only 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

$$\begin{aligned}\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} &= \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{pmatrix} \begin{pmatrix} \hat{x} = W/2 \\ \hat{y} = H/2 \\ \hat{z} = f \end{pmatrix} \\ &= \begin{pmatrix} fc_\theta W/2 + f^2 s_\theta \\ f \cdot H/2 \\ -s_\theta W/2 + fc_\theta \end{pmatrix}.\end{aligned} \tag{6.6}$$

Hence, if the camera is rotated by an angle θ, the top right image corner $(W/2, H/2)$ projects to (see also Fig. 6.6)

$$x(\theta) = \frac{fc_\theta \frac{W}{2} + f^2 s_\theta}{-s_\theta \frac{W}{2} + fc_\theta} \quad ; \quad y(\theta) = \frac{f\frac{H}{2}}{-s_\theta \frac{W}{2} + fc_\theta}. \tag{6.7}$$

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

$$A(\alpha) = W \cdot H + 4 \int_{\theta=0}^{\theta=\alpha/2} y(\theta) \frac{\mathrm{d}x}{\mathrm{d}\theta} \mathrm{d}\theta, \tag{6.8}$$

where the integral covers one of the four symmetric *"wings"* of the sprite. Figure 6.7 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. (6.8). The first one is the coding with an ordinary sprite[1], 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. Hence, the total sprite area that is required for a n-part sprite is $n \cdot A(\alpha/n)$. Figure 6.7 depicts the sprite area A, depending on the total pan angle α for the different set-ups and number of sprites used. 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 in the single-sprite case, the two-part multi-sprite becomes more efficient for pan angles over about 25° ($f = 10$). Finally, for angles exceeding approximately 45°, using a three-part sprite becomes the most efficient partitioning.

[1] But with the optimal selection of the reference frame.

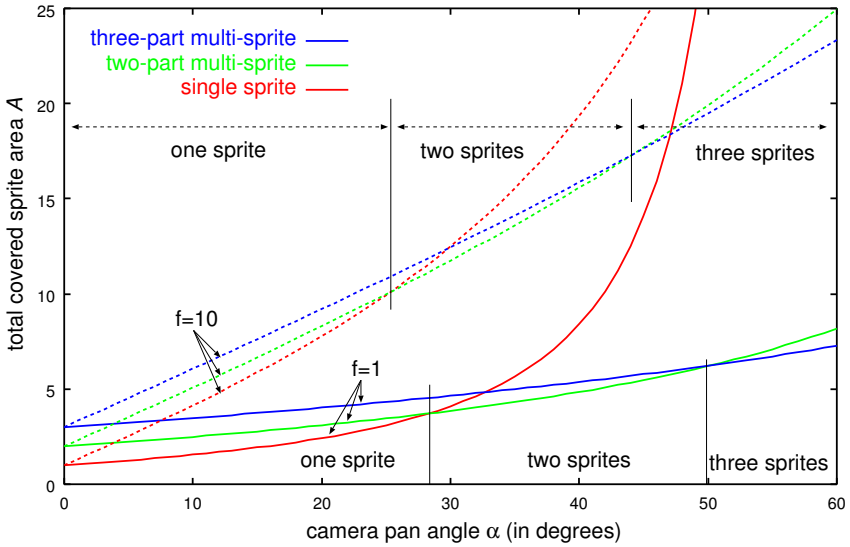


Figure 6.7: *The covered sprite area for horizontal camera pan at two different focal lengths. The reference image size is* 1. *If the camera rotation exceeds a specific angle, the area to be coded in the multi-sprite case is lower than for a single sprite.*

6.4.3 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 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, from now on, we will not only concentrate on the zoom-in case, but indicate the solution for the general case.

Because we want to ensure that $m_l(x', y') \geq 1$ for all pixels in 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

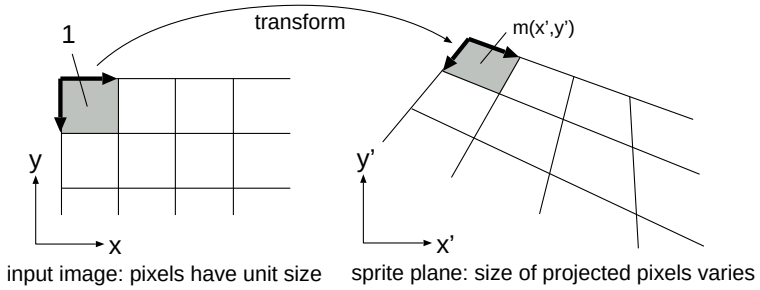


Figure 6.8: *Change of local resolution. The input image (left) is warped to the sprite coordinate system (right). In general, this transformation will change the size of a pixel.*

not constant over a single input frame (see Fig. 6.8). The local scaling factor can be computed using the Jacobian determinant of the geometric transformation Eq. (2.10), which maps the input-image coordinate system to the sprite coordinate system. Consequently,

$$m_l(x',y') = \begin{vmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} \end{vmatrix} = \frac{1}{D^2}\left[\begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} - \begin{vmatrix} a_{00} & a_{01} \\ p_x & p_y \end{vmatrix} y' + \begin{vmatrix} a_{10} & a_{11} \\ p_x & p_y \end{vmatrix} x' \right], \quad (6.9)$$

where $D = p_x x + p_y y + 1$ is the denominator of the motion model equations[2]. For non-degenerated 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 $\bar{m}_{i;k} = \min_{l;\, i \le l \le k} \bar{m}_l$ 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 6.5.4.

[2]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 and independent of x, y.

6.5 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.

6.5.1 Bitstream length

The obvious choice for defining the sprite coding-cost is the bitstream 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 6.6) 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 more easy to compute will be pursued.

6.5.2 Coded sprite area

As an approximation to the actual bitstream length, we can use the sprite area that is covered with image content. In a real implementation, Eq. (6.8) 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 Figure 6.9). Whenever an image is added to the sprite, 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

$$|| \cdot ||_A = \frac{1}{2} \sum_{i \in \{0;\ldots;l-1\}} (x_i y_{i+1} - x_{i+1} y_i) \,. \tag{6.10}$$

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 bitstream length is proportional to the number of coded macroblocks. This is only the case if on

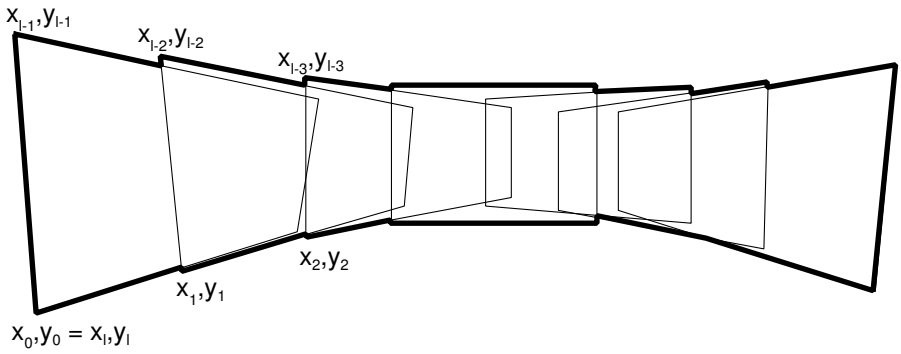


Figure 6.9: *The boundary polygon around the sprite area is computed as the combined outlines of all transformed quadrilaterals. For simplicity of notation, we double the last point* $(x_l, y_l) = (x_0, y_0)$.

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 cost definition of $||\cdot||_A$, 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 towards making magnified areas more costly than when using the theoretically optimal *bitstream length* cost definition of the previous section.

To determine the relationship between the resolution scaling factor and the bitstream length, we scaled images from several sequences to different sizes and compared the bitstream 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 Figure 6.10. Even though the input images have very different content, the relationships between scaling-factor and increase of bitstream length seem to be comparable. Small peaks in bitstream size can be observed at 100%, 150% and 50%. Since bi-linear 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 bitstream 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

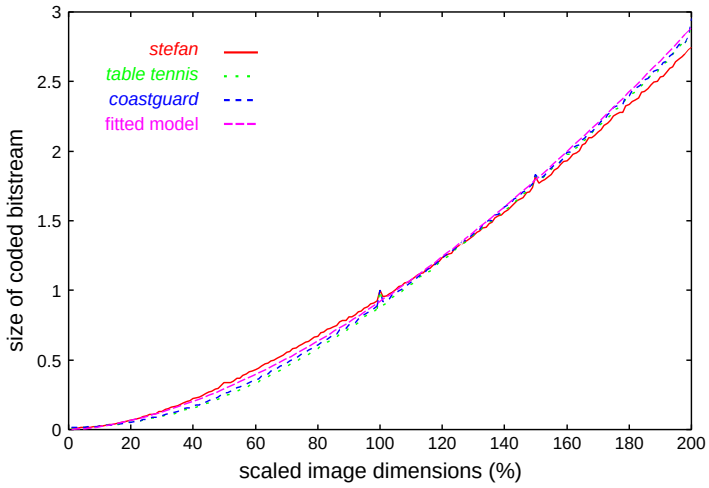


Figure 6.10: *Increase of bitstream length for rescaled image resolutions. Resizing the image dimensions each by a factor $\sqrt{m(x,y)}$ increases the bitstream size by about $m(x,y)^{1.6/2}$, which is a bit less than a linear increase relative to the image area.*

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.

6.5.3 Sprite buffer size

A further approximation to the real bitstream 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.

6.5.4 Adding a resolution preservation constraint and limiting sprite buffer requirements

A cost definition based only on the 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 6.4.3, 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 6.3.

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

$$||S^r_{i;k}||_C = \begin{cases} \infty & \text{if frame range } i;k \text{ cannot be projected onto a single sprite,} \\ \infty & \text{if } ||S^r_{i;k}||_B \text{ exceeds the maximum sprite buffer size,} \\ \frac{||S^r_{i;k}||_A}{\bar{m}_{i;k}} & \text{else.} \end{cases} \tag{6.11}$$

It is easy to see that for any sensible definition of sprite coding-cost, the cost is monotone for the beginning and the end of the frame range. More specifically, for a frame range $a;b$ with $a \leq i$ and $b \geq k$, 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 sub-range $i;k$.

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.

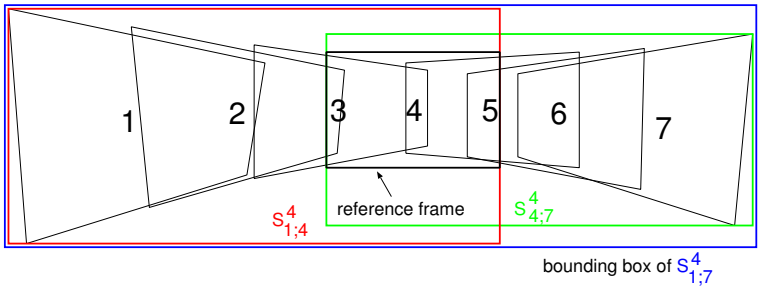


Figure 6.11: *Sprite for frames 1 to 7 with frame 4 as the reference frame. In this case, the bounding-box for $S^4_{1;7}$ has been computed by combining the bounding boxes of $S^4_{1;4}$ and $S^4_{4;7}$.*

6.6 Multi-sprite partitioning algorithm

To find the best multi-sprite configuration, the algorithm has to determine the optimal range of input 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 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.

6.6.1 Cost matrix calculation and reference frame placement

In this preprocessing step, we prepare all the sprite costs required for 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

$$||S^*_{i;k}|| = \min_r ||S^r_{i;k}||. \tag{6.12}$$

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. 6.11). 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 three-dimensional 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.

Consequently, when we determine $||S^*_{i;k}||$ by searching for the r that results in the minimum area bounding box, we do not apply Eq. (6.12) directly, but combine the cost using the two sprite halves as

$$||S^*_{i;k}|| = \min_r ||S^r_{i;r}; S^r_{r;k}||. \tag{6.13}$$

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 Eq. (6.13). This value is not needed for the optimization, but the final sprite-image generation uses the information for selecting the reference coordinate system.

6.6.2 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 frame ranges is minimal. More formally, let $P = \big((1, p_1 - 1), (p_1, p_2 - 1), \ldots, (p_{n-1}, N)\big)$ be a partitioning of the video sequence of length N into n sub-sequences. The optimization problem can then be formulated as determining the partitioning P^* for which the sum of all sprite costs is minimal:

$$P^* = \arg\min_P \sum_{(i,k)\in P} ||S^*_{i;k}||. \tag{6.14}$$

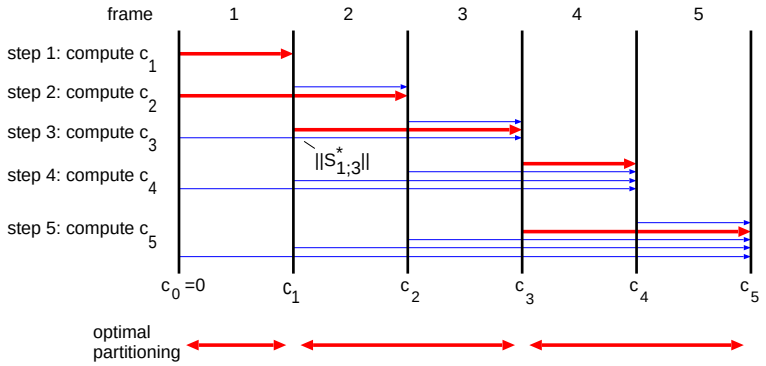


Figure 6.12: *Determination of the optimal sequence partitioning. Each state c_k is assigned the minimum coding cost for a partitioning ending in frame k. Each arrow represents the cost for the sprite built from the covered frames. For each c_k with $k \geq 1$, the sprite that results in the minimum cost in node c_k, is marked with a bold arrow. Tracing back the bold arrows from the last node (c_5) provides the optimal partitioning with minimum cost.*

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 $E = \{(i;k) \mid i,k \in V; i < k\}$. Each edge $(i;k)$ is attributed with edge costs $||S^*_{i+1;k}||$. Every path from the start node 0 to node N defines a possible partitioning, where each edge 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 simple iterative algorithm (Figure 6.12). For each image i, we compute the minimum cost c_k ($c_0 = 0$) of a partitioning ending in image k as

$$c_k = \min_{i \in [1,k]} \left\{ c_{i-1} + ||S^*_{i;k}|| \right\}. \tag{6.15}$$

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.

When searching through the possible values of i in Eq. (6.15), a common

case is that the sprite cost $||S^*_{i;k}||$ will reach ∞ when a cost definition according to Eq. (6.11) is used. As the cost cannot decrease if the frame range is extended (see Section 6.5.4), an efficient way is to carry out the search for i backwards, starting with k and stopping the search as soon as ∞ is obtained for the sprite cost $||S^*_{i;k}||$.

6.7 Experiments and results

We have implemented the algorithm with the sprite cost definition of Section 6.5.4. 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 the Figures 6.17–6.22, we indicate the frame range which was used to generate the sprite, the bounding-box size, and the covered sprite area in 1000-pixel units. The obtained sprite sizes are also summarized in Table 6.1.

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 multi-sprite (Fig. 6.17). 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.

The *rail* sequence (Fig. 6.18) 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 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. 6.19). Applying the multi-sprite algorithm to the sequence results in a three-part multi-sprite, 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. The reconstruction quality of uncompressed background sprites is measured 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

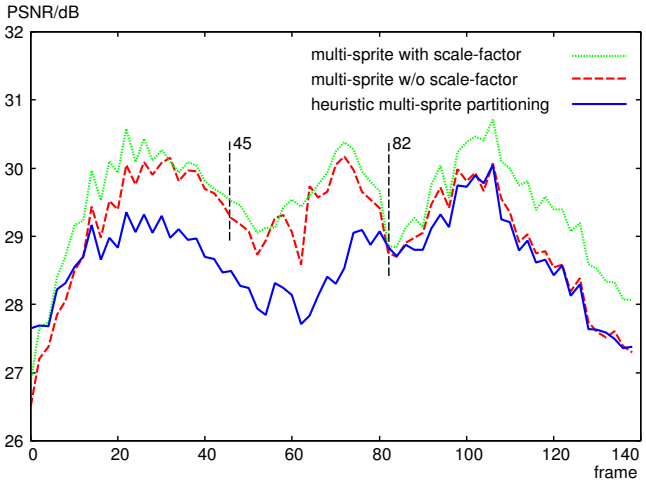


Figure 6.13: *Comparison of sprite reconstruction quality for the rail sequence from Fig. 6.18.*

different types of sprite-construction: multi-sprite 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 6.19), which corresponds to the frame range of the first two sprites obtained from the multi-sprite partitioning. Figure 6.13 depicts the reconstruction quality of the different approaches. Apart from the fact that the multi-sprite 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 a static sprite, although 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. 6.20, which shows a magnification of part of the Fig. 6.18(a). Since the input was

	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

Table 6.1: *Comparison of sprite sizes using single sprites and the multi-sprite approach. The area of the bounding-box and the covered sprite area are expressed in units of 1000 pixels.*

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 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 6.21. Applying the multi-sprite algorithm on the complete sequence resulted in a four-part multi-sprite, which is shown in Figure 6.22. 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 multi-sprite 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. 6.22(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 could be coded with a lower resolution, which outweighs the overhead of an additional sprite.

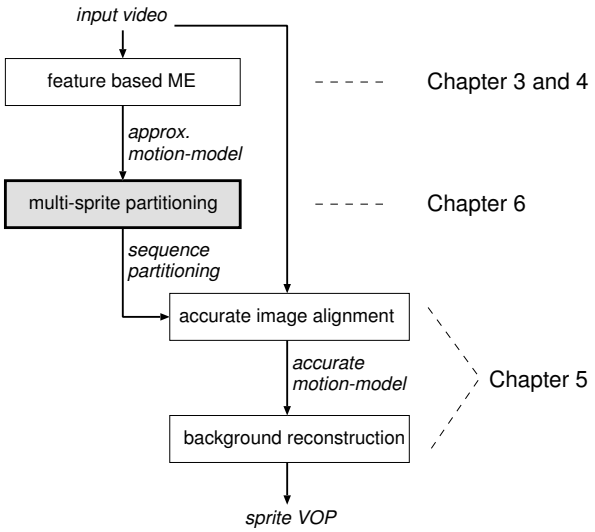


Figure 6.14: *The integration of the multi-sprite partitioning into the framework for background reconstruction.*

6.8 Integration into the segmentation system

This section discusses how the multi-sprite partitioning algorithm can be integrated into the video-object segmentation system. The segmentation system that we started to construct in Chapters 3 to 5 already included all the steps required to synthesize background sprites from the input sequence. However, it comprised all the limitations that we described in the beginning of this chapter.

These limitations can be removed by integrating the multi-sprite partitioning algorithm into our framework. The goal is to take the motion parameters from the feature-based motion estimator and determine the multi-sprite partitioning. This information is then used in the background reconstruction stage to actually synthesize the sprite images from the specified frame ranges.

An overview of our multi-sprite enabled segmentation system is depicted in Figure 6.14. Processing starts with a feature-based global-motion estimator as described in Chapters 3 and 4. This results in a set of approximate motion parameters which are used to compute the multi-sprite partitioning from this chapter. This partitioning defines from which input frames the respective background sprites are synthesized. The multi-sprite algorithm also computes the optimal reference frame, which is used in the refinement step for the long-term motion estimation. This motion parameters refine-

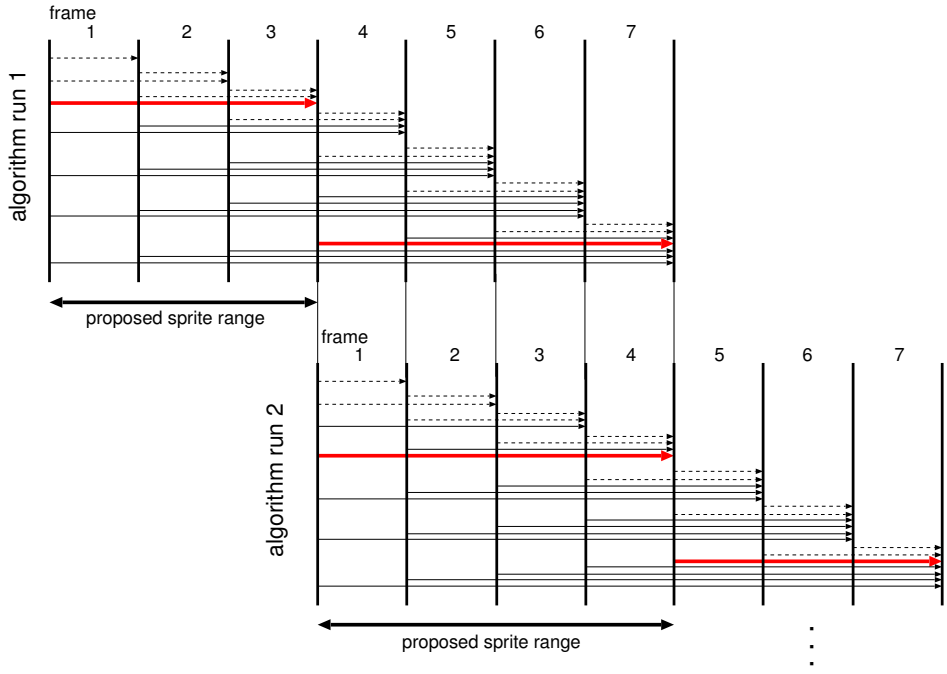


Figure 6.15: *Online calculation of multi-sprite partitionings. After s_{max} frames (here $s_{max} = 7$) have been collected, the optimal partitioning is computed. The first frame-range is returned and the computation proceeds when the buffer fills again to s_{max} frames. Note also that graph-edges spanning less than s_{min} frames have been disabled (dotted lines). In our example, $s_{min} = 3$.*

ment step and the background synthetization is carried out as described in Chapter 5.

6.9 Online calculation of constrained sprites

The presented multi-sprite algorithm computes the optimal partitioning for the complete video sequence. Therefore, it has to consider the motion parameters for all frames in its computation. This has the disadvantage that the sequence has to be processed in at least two passes. The first pass computes the motion parameters and the multi-sprite partitioning, based on which the second pass can synthesize the sprite images. However, for many applications, online processing of the sequence is desired such that virtually infinite input sequences can be partitioned with only a small delay.

In these cases, it is convenient to limit the number of frames per sprite to a maximum of s_{max}. Furthermore, for reasons that will be described in Section 8.2.4, it can also be required to set a minimum number of frames s_{min} that must be included in a sprite. The multi-sprite partitioning can be modified to an online algorithm as follows.

Instead of constructing the complete computation graph for the complete sequence, a graph covering only the first s_{max} frames is generated. All graph edges that span less than s_{min} frames are omitted (or their cost is set to ∞). Note that this graph can be built online while new input frames are received. When s_{max} frames are available, the minimum-cost path is computed as before. This path again defines a partitioning, but now we only output the first frame-range. The subsequent processing stages can then begin to work on the sprite defined by this frame-range in parallel. As new input images arrive at the multi-sprite partitioning algorithm, it again constructs the graph for the next s_{max} frames, as shown in Figure 6.15.

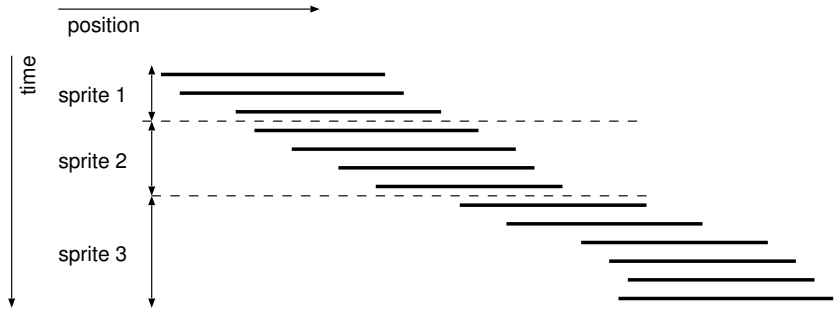
This algorithm does generally not result in a globally optimal solution, but it limits the maximum memory requirement and processing delay that is introduced by the sprite generation to s_{max} frames.

6.10 Coding multi-sprites in MPEG-4 streams

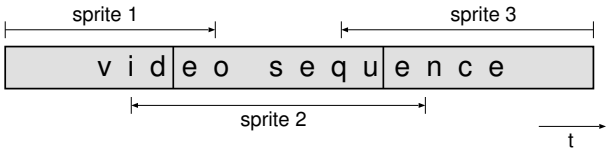
The obtained background sprites can be coded as a sprite VOP with a standard MPEG-4 encoder. To transmit the multi-sprite, we have to consider that we will switch between several sprites. This switching is not addressed in the MPEG-4 standard, but two approaches are possible to simulate this without modifications in the MPEG-4 encoder and decoder. In the first approach, the sprites are transmitted sequentially, where a new sprite is 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 assemble the individual sprites into a single sprite image, where the sprites are placed independently beneath each other. To select the correct sprite at the decoder, the top-left corner position of the sprite is added to the camera-parameters in order to decode the correct sprite. In particular, if the sprite is stored at a displaced position (t_x^0, t_y^0) in the sprite buffer, we send the motion parameters $\mathbf{H}'$, which are computed as

$$\mathbf{H}' = \begin{pmatrix} 1 & 0 & t_x^0 \\ 0 & 1 & t_y^0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{H}, \tag{6.16}$$

where $\mathbf{H}$ are the original camera-motion parameters. This second approach does not require a high peak data-rate, but needs more decoder memory for



(a) Sequence of motion-compensated images.



(b) Frames that are synthesized into the background sprites.

Figure 6.16: *Determining the frame ranges for background synthesis. (a) The video is depicted as a stack of motion-compensated frames. Along the borders of the sprite, foreground objects cannot be removed reliably, since too little temporal information is available. (b) To obtain an optimal suppression of foreground objects, the frame range of each sprite is extended to include more frames that overlap with the sprite area.*

the sprite buffer. Future work might combine the multi-sprite 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.

6.11 Conclusions

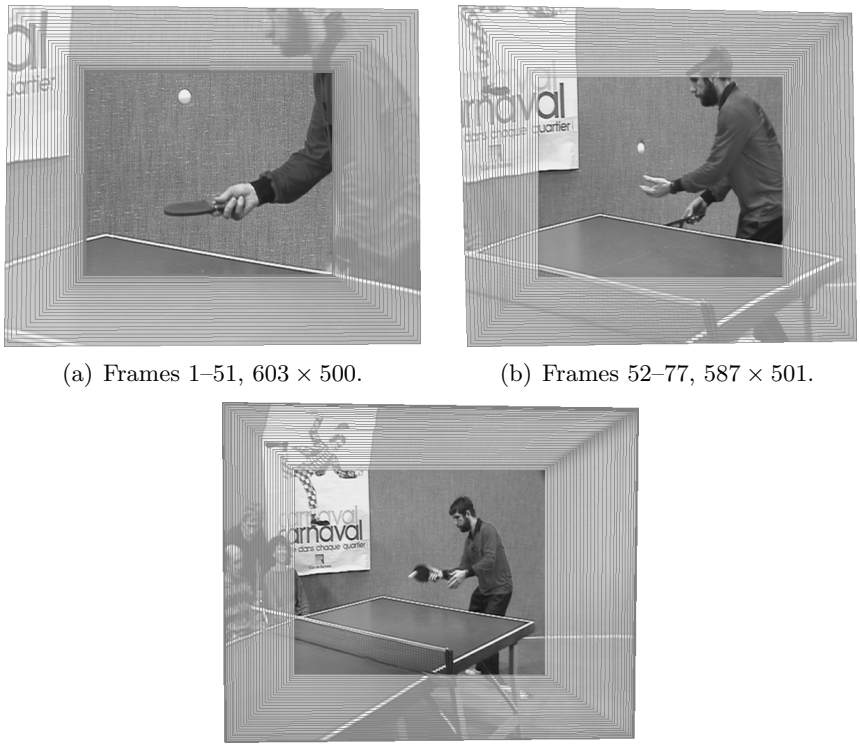
This chapter has shown that partitioning a background sprite into several independent parts results in a clearly reduced coding cost and better resolution at the same time. Our algorithm computes the optimal partitioning of a sequence, the reference frame for each partition, and associated scaling factors. As a consequence, the proposed algorithm solves the subsequent problems.

- It removes the limitation of camera motion and enables to use sprite coding for arbitrary rotational camera motion.

- It selects optimal reference frames and defines multi-sprite partitions to considerably reduce the required amount of data for coding the background sprite, and
- it increases the reconstruction quality of the sprite.

The above features are achieved while remaining fully compatible to the MPEG-4 standard.

Clearly, the reduction of sprite area depends on the type of camera motion in the sequence. For e.g. 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 guaranteed with previous approaches. The ability to handle arbitrary rotational camera motion is an important generalization to previous sprite-construction algorithms which in these cases would simply fail to create the sprite. Moreover, the generalization is not only important for the coding of the background sprite, but also for other image analysis algorithms like our video-object segmentation, which is based on background subtraction. This algorithm also requires a complete coverage of the background environment as obtained from the computed set of background images.

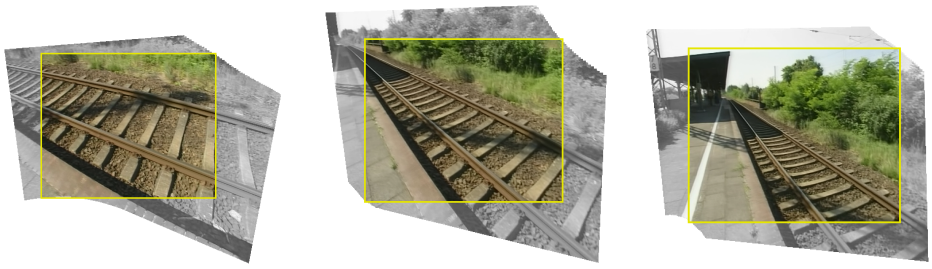


(a) Frames 1–51, 603 × 500.

(b) Frames 52–77, 587 × 501.

(c) Frames 78–132, 585 × 478.

Figure 6.17: *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). For comparison, a single-part sprite generated from the same sequence would result in a size of 1687×1516.*



(a) Frames 1–45, 541 × 445, area: 170k.

(b) Frames 46–82, 479 × 446, area: 171k.

(c) Frames 83–140, 455 × 387, area: 152k.

Figure 6.18: *Multi-sprite 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. For each sprite, the covered sprite area is indicated in 1000-pixel units and the size of the bounding-box. Reference frames are depicted in a darker shade.*

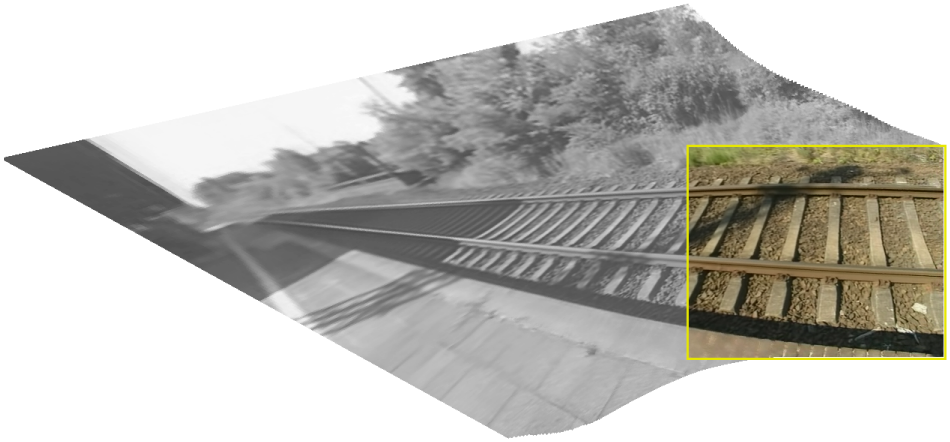


Figure 6.19: *Frames 1–82 integrated into a single background sprite (1264×592, area: 427k). The attempt to integrate the entire rail sequence into a single background sprite fails because of the complicated camera motion. The camera performs an approximate* 90° *rotation around two axes.*

(a) Input. (b) Sprite reconstruction.

Figure 6.20: *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.*

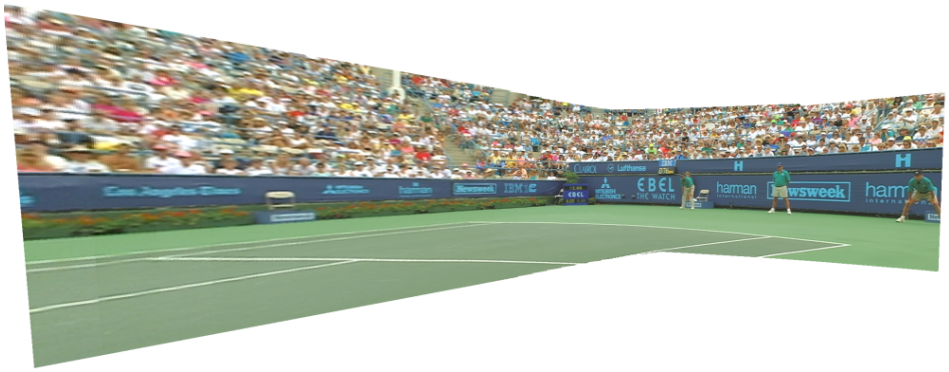
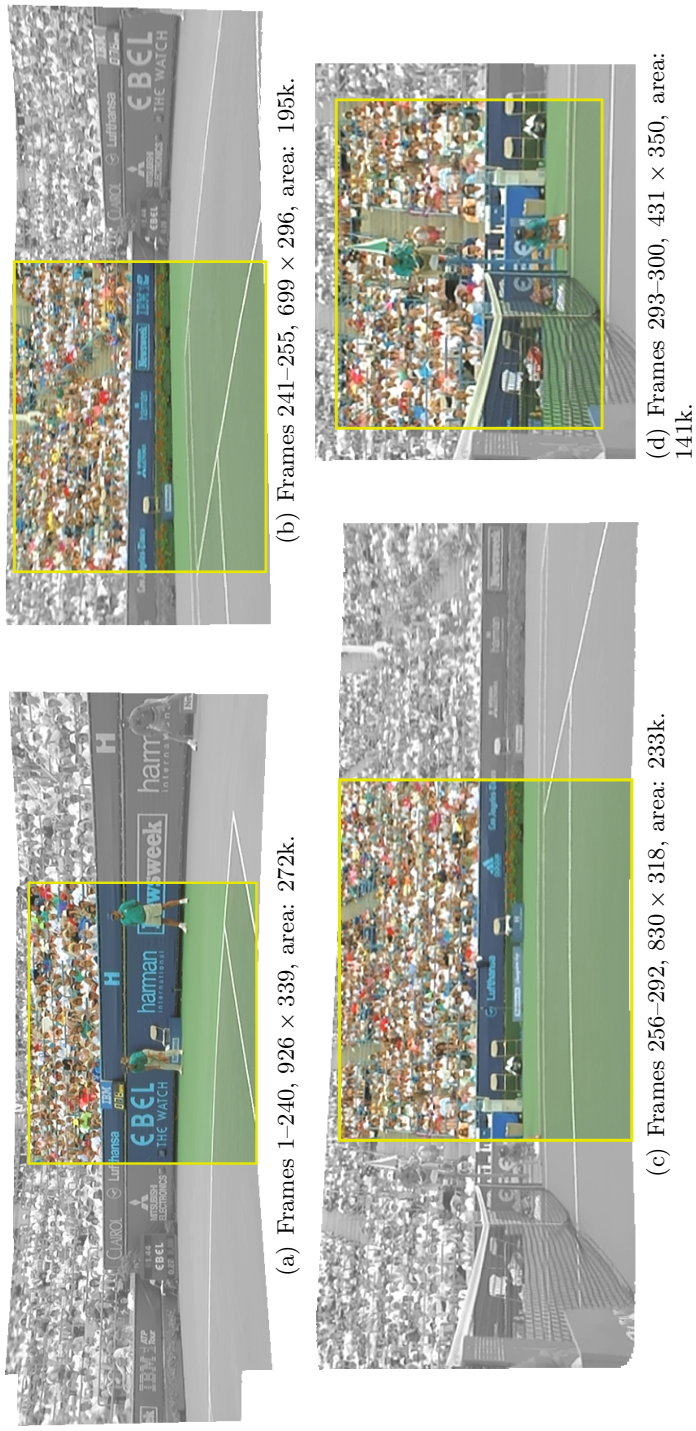


Figure 6.21: *Sprite synthesized from* stefan *sequence. Only the first 255 frames can be used, since it is impossible to create the sprite for the complete sequence if the first frame is selected as reference. Sprite resolution is 2445×1026 pixels, area: 1208k.*



(a) Frames 1–240, 926 × 339, area: 272k.

(b) Frames 241–255, 699 × 296, area: 195k.

(c) Frames 256–292, 830 × 318, area: 233k.

(d) Frames 293–300, 431 × 350, area: 141k.

Figure 6.22: *Multi-sprites synthesized using the described algorithm. The adopted 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).*