Segmentation in Video Data 8

A video sequence is in principle a sequence of images. The methods presented in the previous chapters therefore apply equally well to a video sequence as an image. We simply process one image at a time. There are, however, two differences between a video sequence and an image. First, working with video allows us to consider temporal information and hence segment objects based on their motion. This is discussed below in Sect. 8.2. Moreover, temporal information is the cornerstone of *tracking*, which is described in the next chapter. Second, video acquisition and image acquisition may not be the same, and that can have some consequences. Below, this is discussed.

8.1 Video Acquisition

A video camera is said to have a certain *framerate*. The framerate is a measure for how many images the camera can capture per second and is measured in Hertz (Hz). The framerate depends on the number of pixels (and the number of bits per pixel) and the electronics of the camera. Usually the framerate is geared toward a certain transmission standard like USB, Firewire, Camera Link, etc. Each of these standards has a certain *bandwidth*, which is the amount of data that can be transmitted per second. With a fixed bandwidth we are left with a choice between high resolution of the image and a high framerate. When one goes up the other one goes down. In the end the desired framerate and resolution will always depend on the concrete application.

Say we have a system including a camera with a framerate of 20 Hz. This means that a new image is captured every 50 ms. But it also means that the image processing algorithms can spend a maximum of 50 ms per image. To underline this we often talk about two framerates; one for the camera and one for the image processing algorithms. The overall framerate of a system is the smallest of the two framerates.

Another important factor in video acquisition is compression. Very often the captured video sequence needs to be compressed in order to insure a reasonable framerate/resolution. The more the video is compressed, the higher the framerate/resolution, but the worse the quality of the decompressed video. The question

DOI 10.1007/978-1-4471-2503-7_8, © Springer-Verlag London Limited 2012

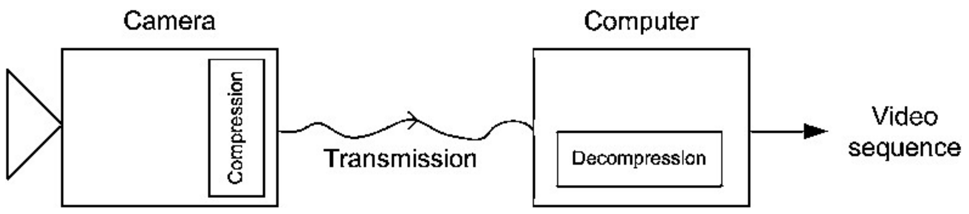

Fig. 8.1 Compression and decompression of video

is of course if the quality lose associated with video compression is a problem in a particular application or not? To be able to answer this we first need to understand what compression is, see Fig. 8.1.

Overall there exist two different types of compression; lossy and lossless. In the latter type the captured video in the camera and the decompressed video on the computer is exactly the same. This is virtually never used and hence not described further. In the former type of compression some information will be lost. Many different lossy video compression algorithms exist, but they all have a similar core. First of all they are developed with focus on the human mind in the sense that if a human looks at the captured video and the decompressed video, the difference should be as small as possible. That is, the information lost in video compression is optimized with respect to the human visual perception capabilities, i.e., a human will not notice the missing information. This may not be optimal from a computer's point of view, in the sense that the information lost in the compression can affect the image processing algorithms, but this is just how it is.

Humans are more sensitive to changes in the lighting than changes in the colors. The YC_bC_r color representation is therefore used, see Sect. 3.3.3, and the C_b and C_r components are compressed harder than the Y component. Another aspect of human perception involved in compression of video is the fact that humans are better at seeing gradual changes in an image as opposed to rapid changes. This fact is utilized by transforming the image into a new representation where the level of change is apparent. Rapid changes are then compressed harder than gradual changes.

Another main ingredient in video compression is to exploit that consecutive images usually do not change very much. To exploit this the image is first divided into a number of blocks. Each block is then used as a template to search for a matching block in the previous image. Template matching is used for this purpose, see Sect. 5.2.1. The two blocks are now subtracted and their difference is usually small and hence can be represented by fewer bits than the original block. This is done for all blocks in the input. The last component in video compression is similar to what is used for image, sound and text compression, namely *entropy coding*. This covers lossless methods that can compress based on the statistical nature of the data. For example, say we have the following pixel values: $2, 3, 3, 3, 3, 3, 3, 3, 67, 12, 12, 12, 12, 10$. Using entropy coding this can be written as $2, 3, 255, 6, 67, 12, 255, 3, 10$, where 255 indicates that the next value states the number of repetitions, that is: $3, 255, 6 = 3, 3, 3, 3, 3, 3, 3$. Originally we had 14 values and now we only have nine values, i.e., a compression factor of $14/9 = 1.56$.

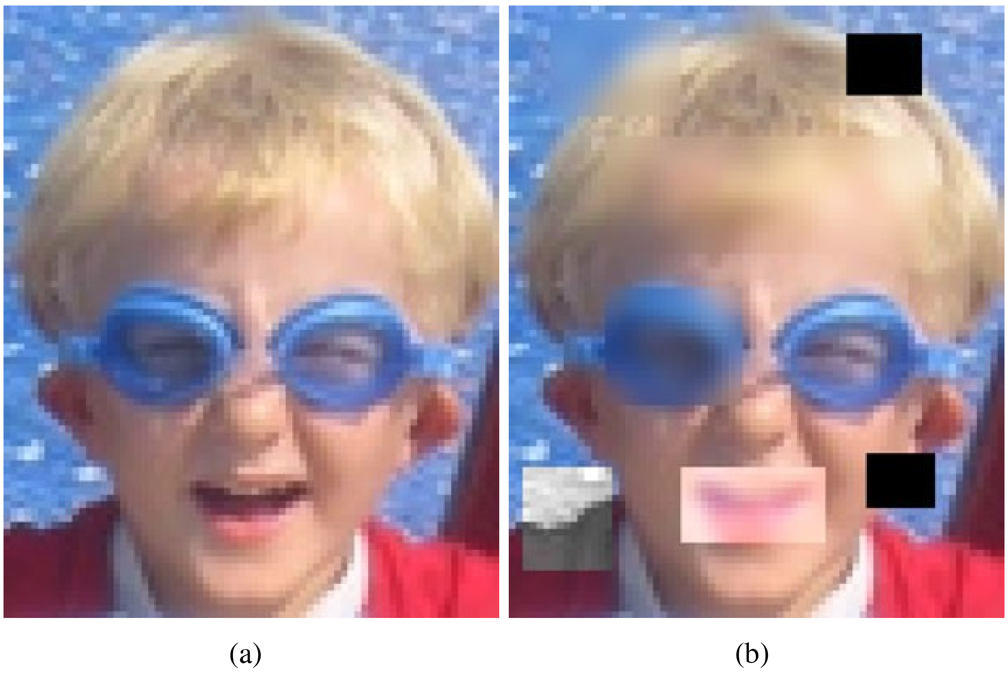

Fig. 8.2 Different blocking effects illustrated in a zoomed picture in order to increase visibility

How many bits needed to represent a compressed image in the video sequence depends on the content of the video. Sometimes we require more bits than are available in the bandwidth. This means that the compression method will have to delete some additional information, for example by a harder compression of the colors or by simply ignoring details of one or more blocks. The consequence of the latter can be that one or more blocks in the decompressed image contain less detail and hence appear blurry or do not contain any detail at all, i.e., will be black. Such phenomena are known as *blocking artifacts* and a few are illustrated in Fig. 8.2.

The point of all the above is that you as a designer need to look into these issues before doing video processing. It might be better to spend some extra money on a good camera (and transmission) producing good data compared to spending lots of time (in vain?) trying to compensate for poor data with clever algorithms. This is especially true if developing a system based on color processing. A compromise can be to use a cheap camera with poor quality video and then try to detect if blocking has occurred and if so delete such images from the video sequence.

No matter the type of camera and compression algorithm, the captured video sequence may contain motion blur due to motion in the scene, see Sect. 2.2. A similar problem is that the depth-of-field may not cover the entire FOV and hence moving objects may be blurred due to an incorrect focus. Processing video containing blur will possibly affect the results and should therefore be avoided is possible. One approach for doing so is to analyze each image and try to measure the level of blur. If too blurry the image should be deleted. The consequence of a blurred image is that the magnitudes of the edges in the image are small. So the blurriness can be measured by analyzing the edges in the image, see Sect. 9.3.1. Another approach is to compare the input image with a blurred version of the input image. If the input image contains a lot of edges, i.e., is sharp, then the two images will be significantly different. If they are similar it is likely that the input image was blurred in the first place.

8.2 Detecting Changes in the Video

In many systems we are interested in detecting what has changed in the scene, i.e., a new object enters the scene or an object is moving in the scene. For such purposes we can use image subtraction, see Chap. 4, to compare the current image with a previous image. If they differ, the difference defines the object or movement we are looking for. In the rest of this chapter we will elaborate on this idea and present an approach for detecting changes in video data.

8.2.1 The Algorithm

The algorithm for detecting changes in a video sequence consists of five steps:

1. Save reference image
2. Capture current image
3. Perform image subtraction
4. Thresholding
5. Filter noise

The algorithm can be performed in two different ways depending on the goal and assumptions. If the background in the scene can be assumed to be static then every new object entering the scene can in theory be segmented by subtracting an image of the background from the current image. This process is denoted *background subtraction* and illustrated in Fig. 8.3. The reference image of the background is captured as the first image when the system commences.

The other way the algorithm can be performed is when the assumption of a static background breaks down. For example if the light in the scene changes significantly, then an incoming image will be very different from the background even though no changes occurred in the scene. In such situations the reference image should be the previous image. The rationale is that the background in two consecutive images from a video sequence is probably very similar and the only difference is the new/moving object, see Fig. 8.4. Such methods are denoted *image differencing*.

The difference between the two ways the algorithm can be performed results in two different types of reference image: either the first in a sequence or the previous image. The remaining four steps in the algorithm are the same for the two algorithms and performed for each new image in the video.

Step Three

In *Step three* of the algorithm the reference image and current image are subtracted. Let us denote the reference image $r(x, y)$ and the current image $f(x, y)$. The resulting image, $g(x, y)$, is then given as

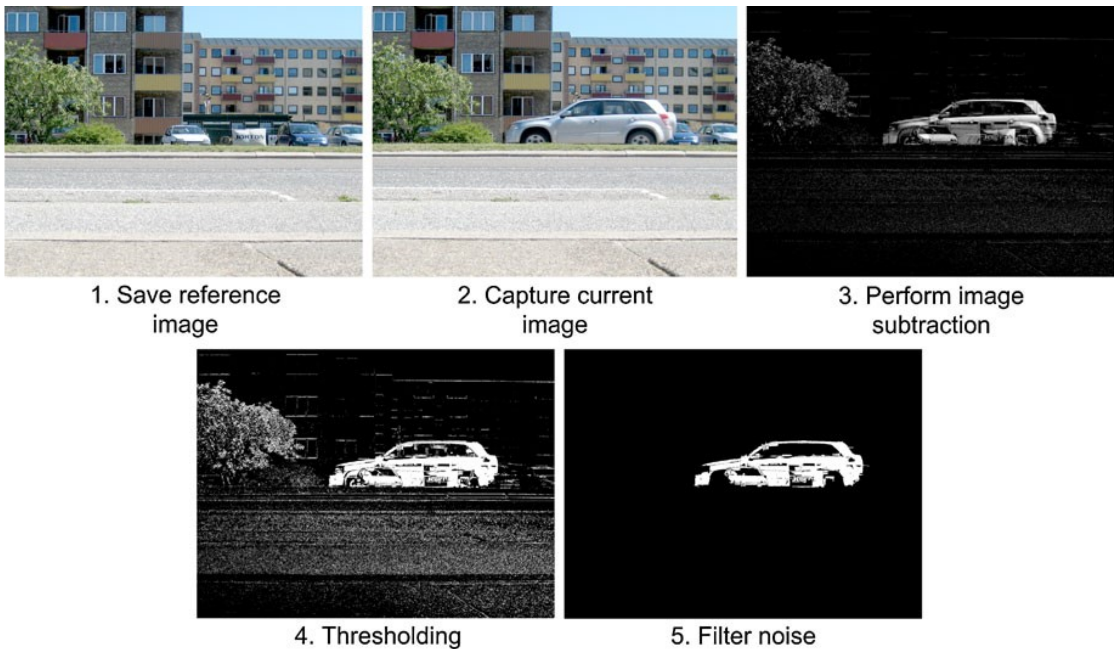

Fig. 8.3 The five steps of segmenting video data through background subtraction

$$g(x, y) = f(x, y) - r(x, y) \tag{8.1}$$

In Fig. 8.3 step three is shown. The car stands out in the resulting image since the pixel values of the car are different from the pixel values of the reference image. However, at some locations where a pixel in the reference image has a similar value to the pixel at the current image the resulting pixel value will be close to zero and hence will not stand out. This can for example be seen around the wheels of the car. As a designer you therefore need to introduce a background which is as different as possible from the object you intend to segment. For example, choosing a black background when segmenting white balls is a very good idea, whereas a white background is obviously not.

Another issue regarding image subtraction, is that negative values are very likely to appear in the resulting image $g(x, y)$. Say that your task is to segment objects when they pass by your camera. The objects are black and white, meaning that they have pixel values which are either black, $f(x, y) \cong 0$, or white $f(x, y) \cong 255$. You then design a gray background which has intensity values around 100, i.e., $r(x, y) \cong 100$, and perform image subtraction:

$$g(x, y) \cong \begin{cases} 0, & \text{where the object is not present;} \\ 155, & \text{where the object is white;} \\ -155, & \text{where the object is black} \end{cases} \tag{8.2}$$

A common error is to set a negative pixel to zero. If this is done then only the white parts of the object is detected. Note that whether $g(x, y) = 155$ or $g(x, y) = -155$ is equally important. The correct solution is to apply the absolute value of $g(x, y)$, $Abs(g(x, y))$, see Appendix B.

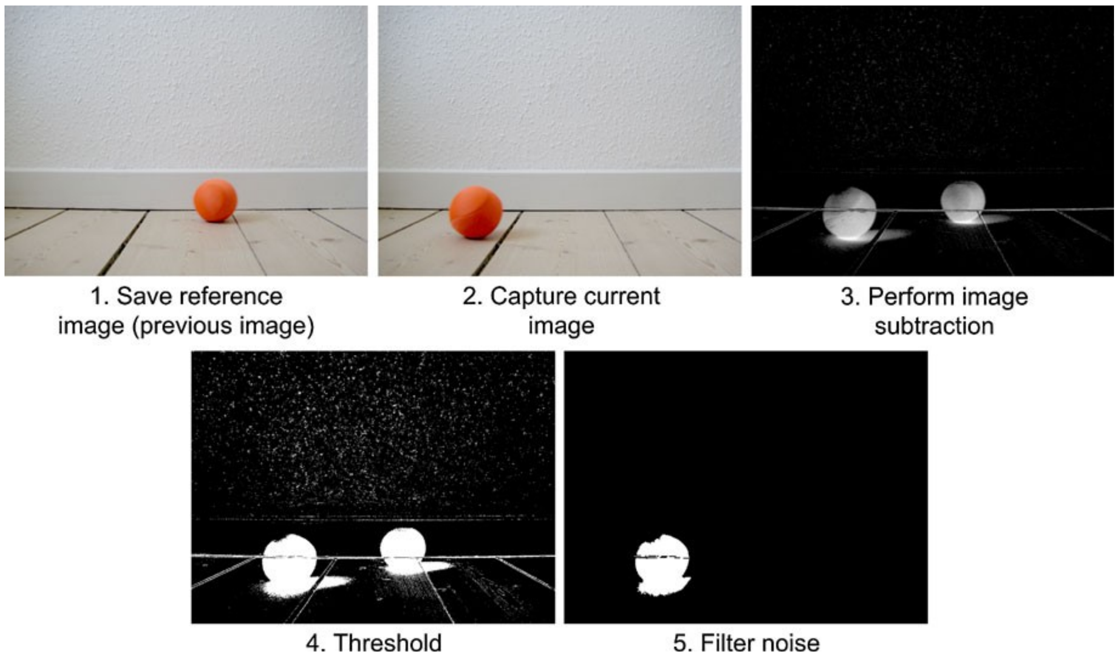
1. Save reference image (previous image)
2. Capture current image
3. Perform image subtraction
4. Threshold
5. Filter noise

Fig. 8.4 The five steps of segmenting video data through image differencing

Step Four

Step four of the algorithm is simply a matter of binarizing the difference image $Abs(g(x, y))$ by comparing each pixel with a threshold value, T:

$$\text{Binary image} = \begin{cases} 0, & \text{if } Abs(g(x, y)) < T; \\ 255, & \text{otherwise} \end{cases} \tag{8.3}$$

Step Five

The threshold in step four will, like any other threshold operation, produce noise due to an imperfect camera sensor, small fluctuations in the lighting, the object being similar to the background, etc. The noise will be in the form of missing pixels inside the silhouette of the object (false negatives, see Appendix C) and silhouette-pixels outside the actual silhouette (false positives). See Figs. 8.3 or 8.4 for examples.

The noise will in general have a negative influence on the quality of the results and *step five* therefore removes the noise (if possible) using some kind of filtering. Small isolated silhouette-pixels outside the actual silhouette can often be removed using either a median filter or a morphologic opening operation. The holes inside the silhouette can often be removed using a morphologic closing operation. Which method to apply obviously depends on the concrete application.

In the following, image differencing and background subtraction are explained in more detail.

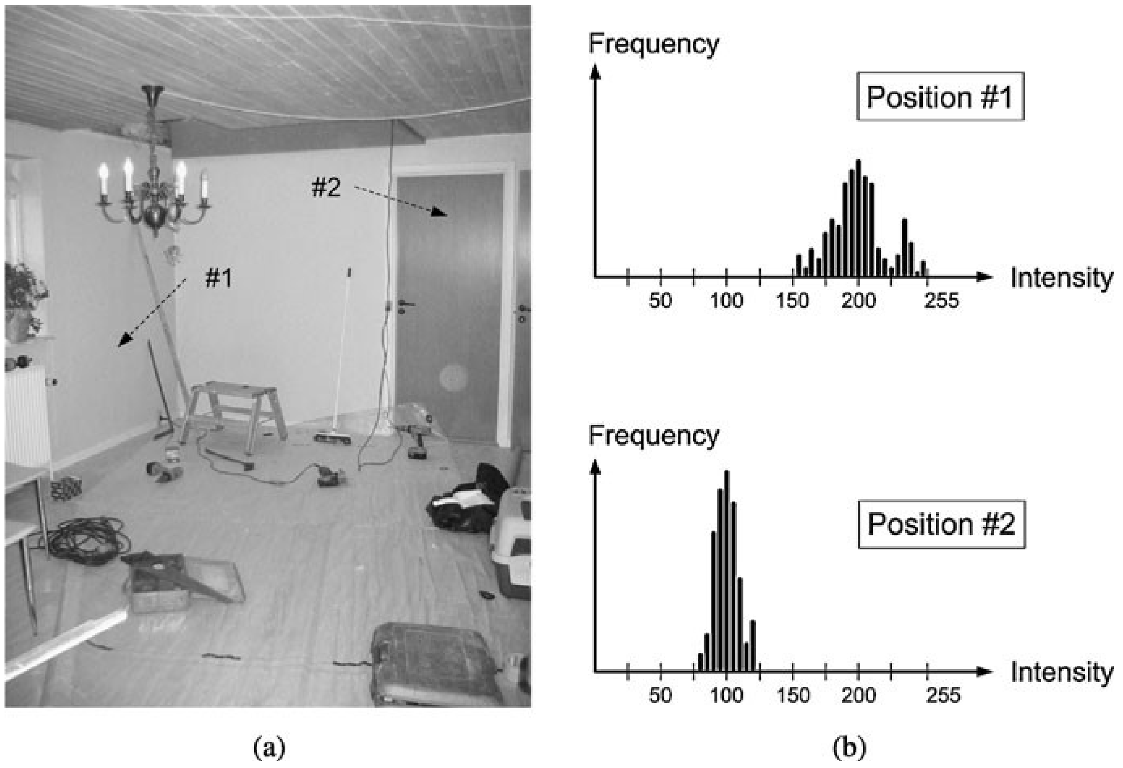

Fig. 8.5 (**a**) A static background image. The *two arrows* indicate the position of two pixels. (**b**) Histograms of the pixel values at the two positions. The data come from a sequence of images

8.3 Background Subtraction

Background subtraction is a simple and yet efficient method of extracting an object in a scene. This is especially true if the background can be designed to be uniform. In indoor and controlled setups this is indeed realistic, but for more complicated scenarios, other methods might be necessary. Even in the case of a controlled setup two issues must be considered:

1. Is the background really constant?
2. How to define the threshold value, which is used to binarize the difference image?

When you point a video camera at a static scene, for example a wall, the images seem the same. Very often, however, they are not. The primary reason being that artificial lighting seldom produces a constant illumination. Furthermore, if sunlight enters the scene, then this will also contribute to the non-constant illumination due to the randomness associated with the incoming light rays. The effect of this is illustrated in Fig. 8.5. To the left an image from a static scene is shown. To the right two histograms are shown. The first histogram is based on the pixel values at position #1 for a few seconds and similar for the second histogram. If the images are actually the same, the histograms would contain only one non-zero bin. As can be seen this is not the case and in general no such thing as a static background exists.

Say that the pixel at position #2 in the first image of the video sequence has a value of 80 (not very likely according to the histogram, but nevertheless possible). If the first image is used as the reference image, then typical background images (around 100 according to the histogram) will result in a difference around 20. Depending on the threshold value, this could actually be interpreted as an object in the scene, since it seems different from the reference image. This is obviously not desirable and each pixel in the reference image should therefore be calculated as the *mean* of the first N images, see Appendix C. The reference image at this particular position will then be around 100, which is much more appropriate according to the histogram. So to make the background subtraction more robust the first few seconds of processing should therefore be spent on calculating a good reference image.

Sometimes the background changes during processing. For example due to the changing position of the sun during the day or due to changes in the illumination sources, e.g., they are accidentally moved. In such situations a new reference image should be calculated. But how do we detect that this has happened? One way is, of course, if we can see that the performance of the system degrades. An automatic way is to gradually change the value of each pixel in the reference image in the following way:

$$r_{\text{new}}(x, y) = \alpha \cdot r_{\text{old}}(x, y) + (1 - \alpha) \cdot f(x, y) \tag{8.4}$$

where $r(x, y)$ is the reference image, $f(x, y)$ is the current image, and α is a weighting factor that defines how fast the reference image is updated. The value of α depends on the application, but a typical value is $\alpha = 0.95$.

8.3.1 Defining the Threshold Value

As for any other threshold operation, defining the actual threshold value is a trade-off between false positives and false negatives, see Appendix C, which is application-dependent.

It is important to notice that Eq. 8.3 is actually based on the assumption that the histograms for different pixel positions are similar and only differ in their mean values. That is, it is assumed that the variation in the histograms is similar. In order to understand the implications of this assumption let us have a closer look at the bottom histogram in Fig. 8.5 together with Eq. 8.3. Say we define the threshold value to 25. This means that an object in an image needs to have a value below 75 or above 125 in order to be segmented as an object pixel and not a background pixel. This seems fine. But then have a look at the top histogram in Fig. 8.5. Clearly this histogram has a larger variation and applying a threshold of 25 will result in incorrect segmentation of pixel values in the intervals: $[150, 175]$ and $[225, 255]$.

In many situations different histograms will occur simply because the different parts of the scene are exposed to different illumination conditions, which yields histograms with different variations. For example, you could have some parts of the background which move slightly (due to a draft for instance) and this will create a larger variation. So to sum up the above, the problem is that each position in the image is a associated with the same *global threshold* value.

The solution to this potential problem is to have a unique threshold value for each pixel position! Finding these manually, is not realistic simply due to the number of pixels and the threshold values are therefore found automatically by the use of the standard deviation for each pixel position, see Appendix C. So when the mean of each pixel is calculated, so is the standard deviation. Equation 8.3 is therefore reformulated as

$$\text{Binary image} = \begin{cases} 0, & \text{if } Abs(g(x,y)) < \beta \cdot \sigma(x,y); \\ 255, & \text{otherwise} \end{cases} \tag{8.5}$$

where β is a scaling factor and $\sigma(x,y)$ is the standard deviation at the position (x,y). Since β is the same for every position, we have no more parameters to define than above, but now the thresholding is done with respect to the actual data, hence a *local threshold*.

8.4 Image Differencing

If the assumption of a static background is violated significantly then background subtraction will produce incorrect results. In such situations we can apply image differencing to detect changes in a scene. As stated above, image differencing operates as background subtraction, except for the fact that the reference image is now a previous image.

Image differencing is simple and can efficiently measure changes in the image. Unfortunately the method has two problems. The first is a lack of detecting new objects which are not in motion. Say a new object enters a scene. As long as the object moves, image differencing detects this in the image subtracting process, but if the object stops moving, the reference image will be equal to the current image and hence nothing is detected. This is a clear weakness compared to background subtraction, which is indifferent to whether the new object is moving or not, as long as the appearance of the object is different from the background.

The other problem associated with image differencing is the notion of *ghost objects* illustrated in Fig. 8.6. The figure contains artificial images from a sequence where an object is moving horizontally through a scene. To make it simple, the object is a square with uniform gray-scale value. What can be seen is that the image differencing produces two segments (smaller objects). One originates from the current object and the other one from the object in the reference image—where the object was. This latter segment type is denoted a ghost object, since no object is present. A ghost object can also be seen in Fig. 8.4.

If the goal is only to obtain the coarse motion in the image, then this does not matter. If, however, we are only interested in the position of the object in the current image, then we need to remove ghost objects. One approach for doing so is if we know the moving direction of the object. We can then infer which is the object and which is the ghost. Another approach is if we know that the object is always brighter than the background. Then the pixels belonging to the ghost will have negative values after the image subtraction.

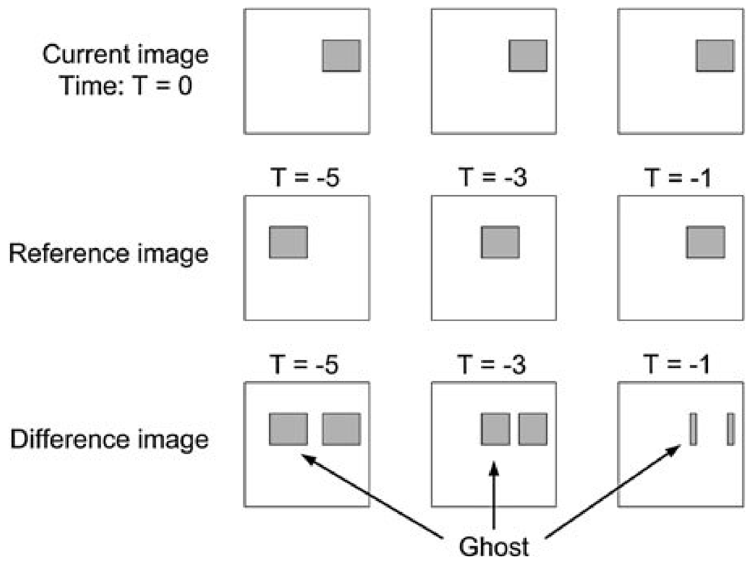

Fig. 8.6 Image differencing. The effect of changing the reference image

It should also be noticed that when the object is overlapping in the reference and current image, then we only detect a part of the object, as seen in Fig. 8.6. If we know the size and speed of the object we can calculate how long time there should be between the reference image and the current image to avoid overlap. Or in other words, the reference image need not be the previous image $T = -1$, it can also be for example $T = -5$, see Fig. 8.6.

8.5 Further Information

Video compression has for a long time been a cornerstone in video acquisition and allowed for transmission and storage of video data. Video compression is a research field in its own right and contains many more aspects than those basics presented in this chapter. The interested reader is referred to a textbook dedicated to exactly this field, see for example [17]. As the hardware and software has matured it has become possible to capture, transmit and store larger and larger amounts of video data. But even with today's fast computers, clever transmission systems, and huge storage facilities, the handling of video data can still be too difficult and a reduced framerate/resolution/quality is necessary. To appreciate this fact just imagine the amount of video data captured, transmitted and stored in a surveillance setup with for example 100 cameras.

Background subtraction can be a powerful allied when it comes to segmenting objects in a scene. The method, however, has some build-in limitations that are exposed especially when processing video of outdoor scenes. First of all, the method requires the background to be empty when learning the background model for each pixel. This can be a challenge in a natural scene where moving objects may always be present. One solution is to median filter all training samples for each pixel. This will eliminated pixels where an object is moving through the scene and the resulting model of the pixel will be a true background pixel. An extension is to first order all training pixels (as done in the median filter) and then calculate the average of the pixels closest to the median. This will provide both a mean and variance per pixel. Such approaches assume that each pixel is covered by objects less than half the time in the training period.

Another problem that is especially apparent when processing outdoor video is the fact that a pixel may cover more than one background. Say we have a background pixel from a gray road. Imagine now that the wind sometime blows so a leaf covers the same pixel. This will result in two very different backgrounds for this pixel; a greenish color and a grayish color. If we find the mean for this pixel we will end up with something in between green and gray with a huge variance. This will render a poor segmentation of this pixel during background subtraction. A better approach is therefore to define two different background models for this pixel; one for the leaf and one for the road, see [12, 18] for specific examples and [9] for a general discussion.

Yet another problem in outdoor video is shadows due to strong sunlight. Such shadow pixels can easily appear different from the learnt background model and hence be incorrectly classified as object pixels. Different approaches can be followed in order to avoid such misclassifications. First of all, a background pixel in shadow tends to have the same color as when not in shadow—just darker. A more detailed version of this idea is based on the notion that when a pixel is in shadow it often means that it is not exposed to direct sunlight, but rather illuminated by the sky. And since the sky tends to me more bluish, the color of a background pixel in shadow can be expected to be more blueish too. Secondly, one can group neighboring object pixels together and analyze the layout of the edges within that region. If that layout is similar to the layout of the edges in the background model, then the region is likely to be a shadow and not an object. For more information please refer to [6, 15].

8.6 Exercises

Exercise 1: Explain the following concepts: framerate, compression, background subtraction, local vs. global thresholding, image differencing, ghost object.

Exercise 2: What is the compression factor of the following sequence of pixels if we apply entropy coding? 14, 14, 14, 7, 14, 14, 14, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 4, 4, 4, 4.

Exercise 3: A camera has a framerate of 125 Hz. How many images does the camera capture per minute?

Exercise 4: A camera captures a new image every 125 ms. What is the framerate of the camera?

Exercise 5: A function is defined as $y = abs(x - 1)$. Draw this function for $x \in [-10, 10]$.

Exercise 6: The reference image $r(x, y)$ in background subtraction is updated gradually with a weight (α) of 0.9. At one point in time a pixel at position (50,50) in the reference image has the value 100, that is, $r(50, 50) = 100$. In the next five images we have: $f(50, 50) = 10$, $f(50, 50) = 12$, $f(50, 50) = 12$, $f(50, 50) = 14$, $f(50, 50) = 15$. What is $r(50, 50)$ after these five frames?