8.2 Basic Parameters for Video Analysis

The content analysis can be divided into three processing steps:

- determination of basic (physical) parameters from the digital data stream,
- computing of semantic features on a higher level of abstraction,
- editing and concatenation of the various algorithms to form user-friendly applications.

In the following, we will discuss the above topics for video (both still images and sequences of images) and later for audio.

Analysis of Still Images

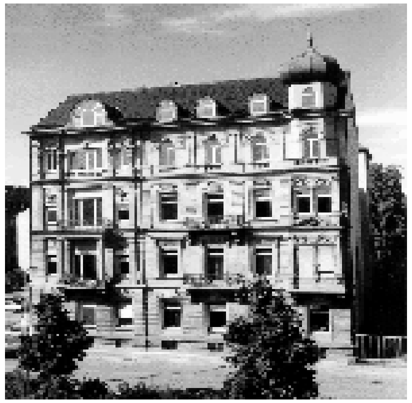
Color Histograms

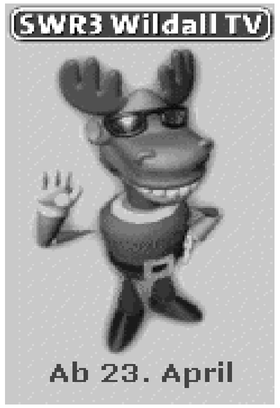
The easiest and most important characterization of a still image is its color histogram. It describes the occurrence of the different colors (resp. gray-scale values) in the image.

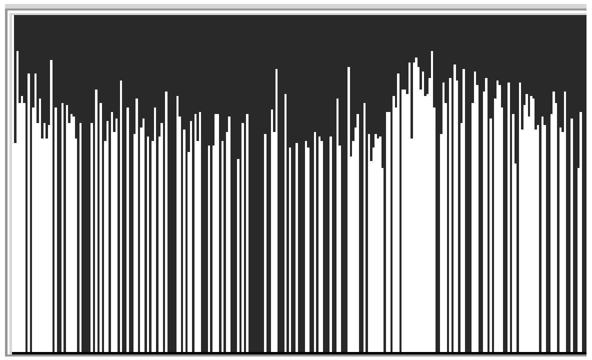
The histogram of a color image has three dimensions (i.e., RGB or YUV), while it is one-dimensional for gray-scale images.

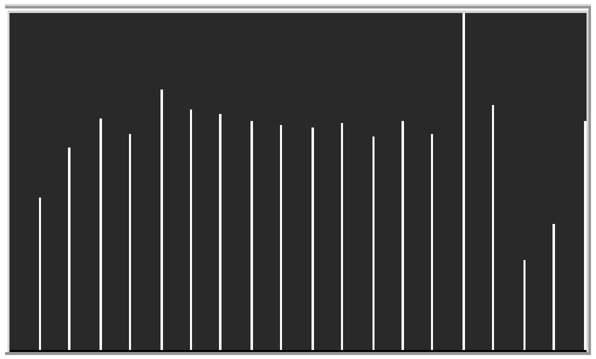
The color histogram of an image is often used as a simple search filter in image data bases when the objective is to search for images similar to a given query image.

Sample Gray-Scale Histograms







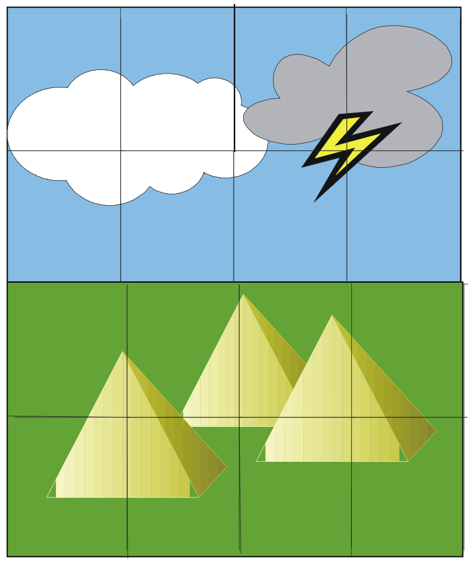


Tesselation

A major drawback of color histograms is that they do not discriminate the location and grouping of colors in the image. Examples:

- an image containing much sky might have a similar color histogram as an image containing much water,
- an image of a sunset might have a similar histogram as an image showing Californian poppies.

A first improvement is the **tesselation** of an image into a mosaic of rectangles of equal size. One then demands that similar colors appear at the same locations.



Color Coherence Vectors

A **color coherence vector** (CCV) contains two entries per color value:

- the percentage of pixels contained in a region of size above average, and
- the percentage of pixels contained in a region of size below average.

$$\text{CCV} = <(\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)>$$

Color coherence vectors facilitate the description of similarities between images. They do not identify the exact position of an object, but they allow to discriminate between the appearance of a specific color in few large regions or in many small regions.

Edge Detection

Another interesting low-level parameter for the analysis of an image is the occurrence of edges. They delimit the various objects contained in an image. **Edge detection** is an important field of research in image processing.

There are many approaches to detect edges in an image. Two of them are:

- line tracing, i.e., a given part of an edge is traced along the border of an object, and
- region growing, based on the color values of the pixels.

In practice, region growing is usually less sensitive to noise.

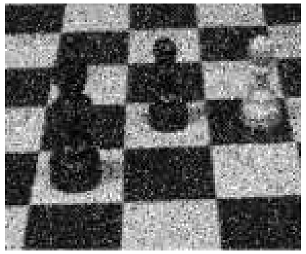
Algorithm “Edge Tracing“ (heuristic approach)

1. Start with a pixel located on the edge that is to be followed.
2. For all end points of edges found so far:
   - 2.1 Investigate the 1-pixel-neighbourhood of the end point
   - 2.2 If a neighbouring pixel has a difference in color value of Δ_c or less, then add this pixel to the line until no more progress can be made.

Two problems immediately emerge:

- What is a best value for Δ_c?
- What happens if the real edge is interrupted by noise? Should the search area be enlarged? Should we interpolate? Across how many pixels?

original image


noisy image

Algorithm “Edge Tracing“ (graph search approach)

Heuristic approach

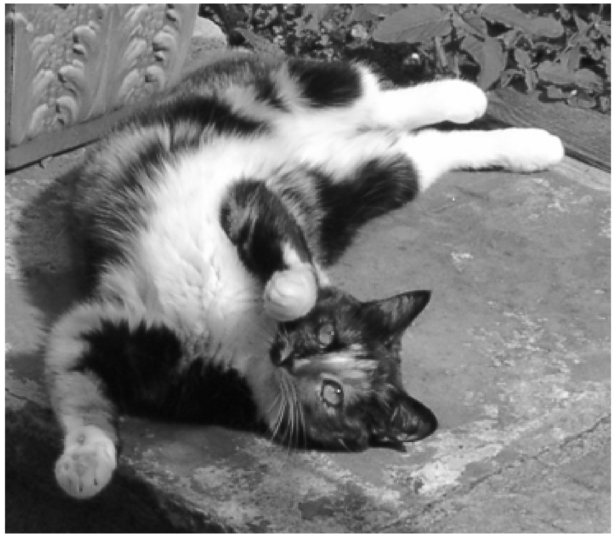
Advantage:	Only the starting point of an edge has to be known
Disadvantage:	What happens if the edge is interrupted / what happens in the case of more than one possibility for the continuation?

Graph search approach

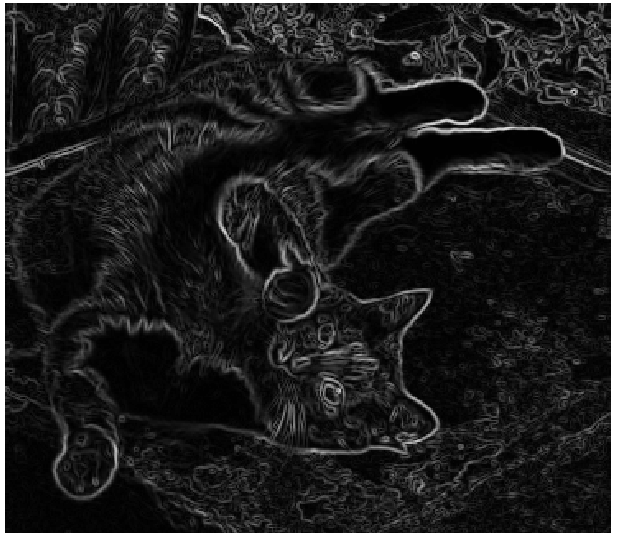
Precondition:	We need a starting point A and an end point B of the edge.
Idea:	Consider all possible paths, pixel by pixel, between A and B. Each path is assigned a cost value. Choose the path with minimal cost as the approximated edge.

Example for the Graph Searching Algorithm

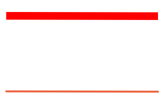
Photo



gradient image (1. derivative)

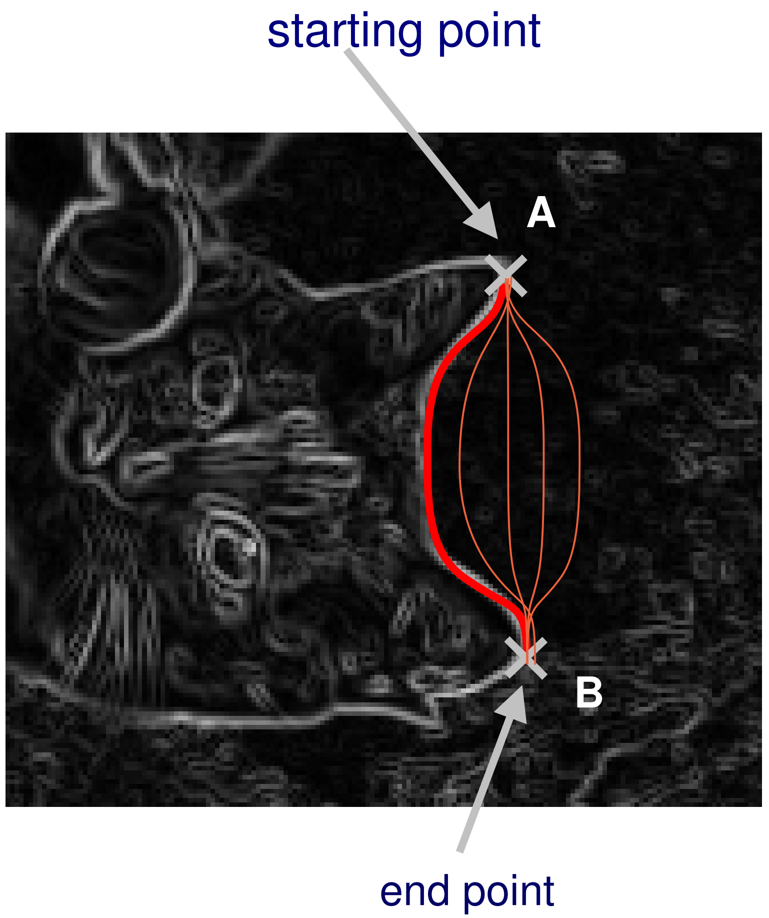


Edge Tracing with the Graph Searching Algorithm



Problem 1: How do we define the cost function such that the stronges edge gets the best score?

Problem 2: How do we find the edge with the minimum cost out of a *very* large set of edges?



A Cost Function for Edge Tracing

T = trajectory between A and B

c(T) = cost of trajectory T

(x, y) = image location (x, y)

g(x,y) = gradient at location (x,y)

g_{max} = highest possible gradient

$$c(T) = \sum_{\forall (x,y) \in T} g_{\max} - g(x,y)$$

$$T_{\min} = \arg\min_{T} c(T)$$

Imagine the gradient image as a mountain landscape. Trajectories travelling along the ridge of the mountain get low total cost values.

Yet to be solved: Find the trajectory T_{min} with minimal c(T) out of a very large number of trajectories.

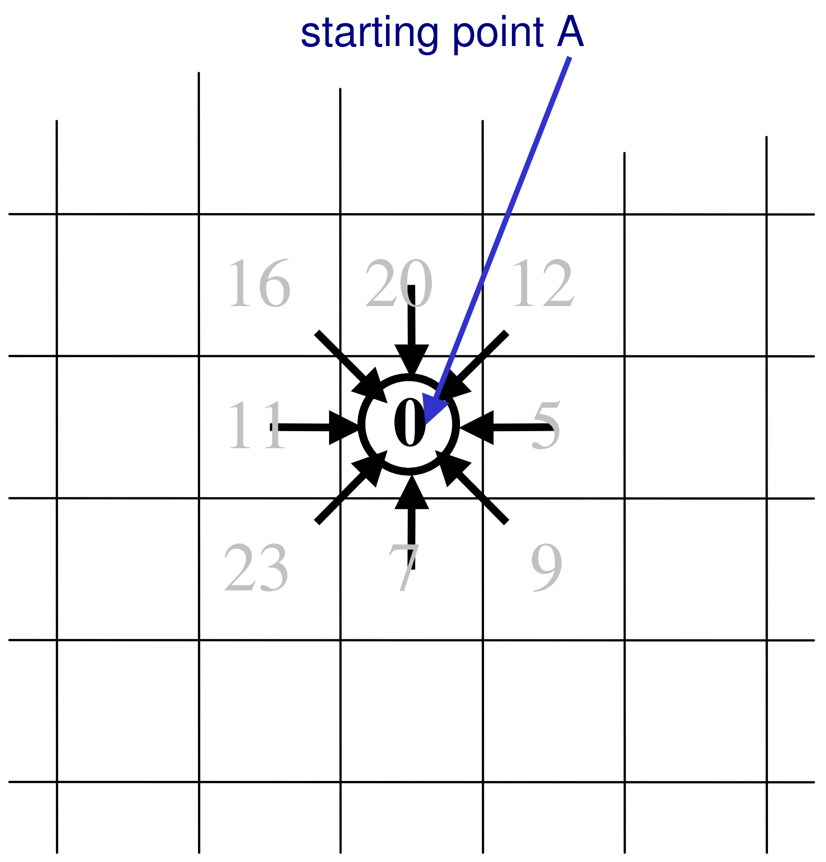
Finding the Least-Cost Trajectory Efficiently (1)

For finding the trajectory T_{min} we use a simple breadth search starting at A and spreading into all directions. We are done when we reach B.

The starting point gets the cost of zero. It is at the same time put into a priority queue.

(1) Remove the current point P with the least cost value from the queue

(2) Put all neighbors of P into the queue. Their assigned costs are their own costs plus the cost of P



Finding the Least-Cost Trajectory Efficiently (2)

Algorithm Find-Least-Cost-Trajectory

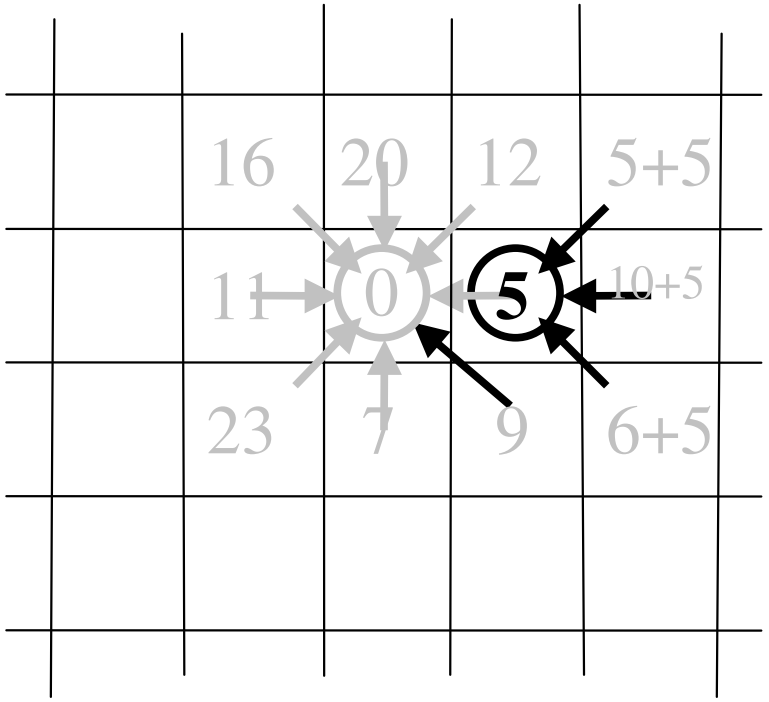
(1) Remove the current point P with the minimum cost value from the queue

(2) Put all neighbors of P into the queue. Their assigned cost are their own costs plus the cost of P.

(3) Store a reference pointing back to P

(4) If you see a pixel which is already in the queue, remove the old instance if the new one has lower cost (and do not forget to update the reference)

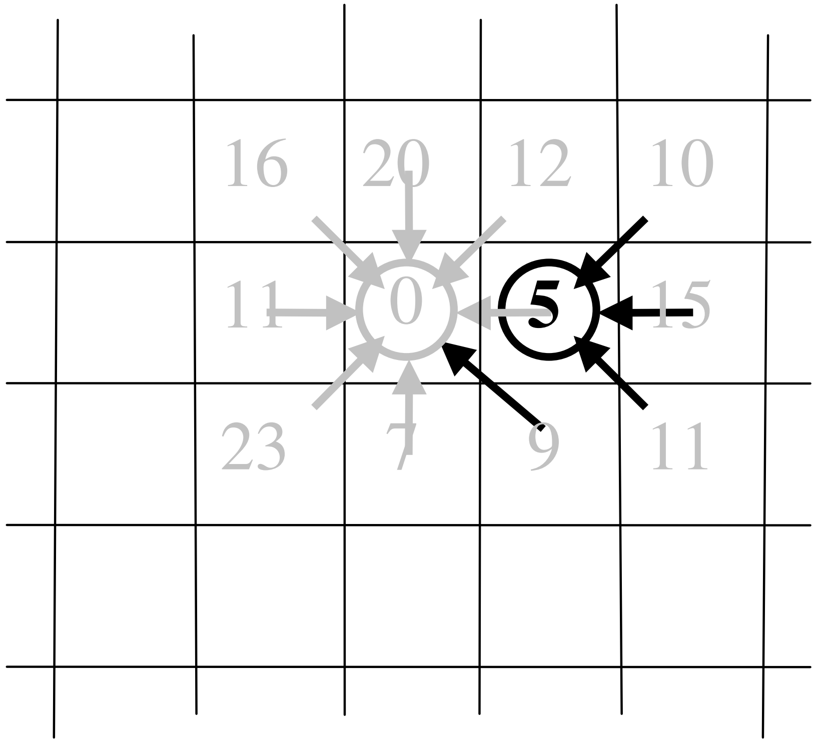
(5) Goto (1) unless the queue is empty or B is reached



Properties of Find-Least-Cost-Trajectory

Properties of Find-Least-Cost-Trajectory

- The priority queue ensures the path is the shortest
- If we don't stop at B but process all pixels of the image we can use the graph for finding every edge back to A (see next slide)
- Once the graph is build we get T_{min} simply by tracking the tree from a node or leaf back to the root A. A trajectory of n pixels is derived efficiently in O(n).

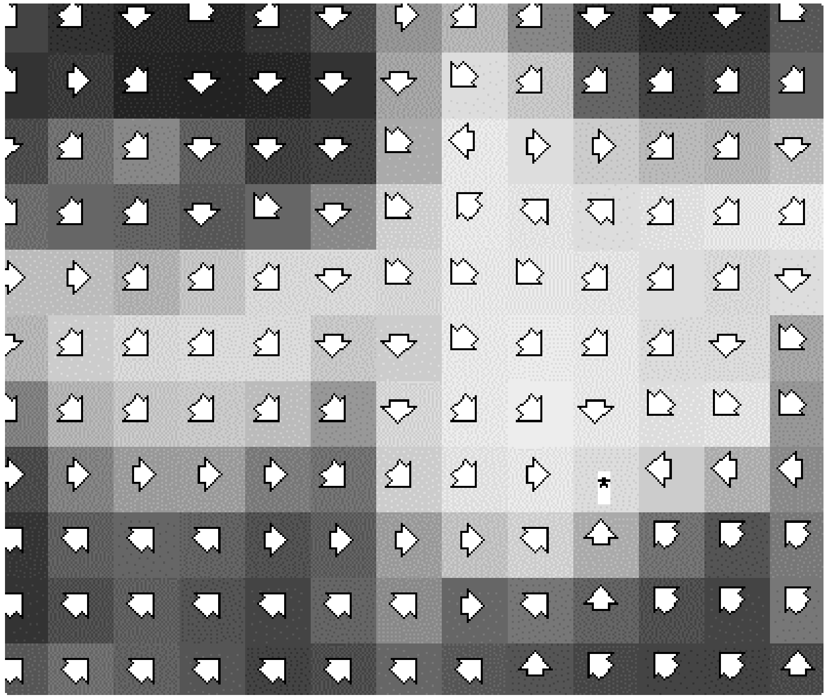


Example of Find-Least-Cost-Trajectory

Example: subset of a graph embedded into the image for visualization.

Each square corresponds to a pixel. The starting point A ist marked with an "*".

- Sooner or later every trajectory ends in A
- Trajectories tend to snuggle to a local edge first before converging to A (why?)



Algorithm “Region Growing”

1. The initial set of regions is empty.
2. Find an arbitrary pixel that is not yet contained in a region. This pixel defines the current region.
3. Repeat for all pixels within the current region:
   - 3.1 Investigate the 1-pixel-neighborhood of the pixel
   - 3.2 If a neighboring pixel has a color difference of Δ_c or less, then add it to the region until the current region stops growing.
4. If there are still pixels not belonging to any region, proceed with step 2.

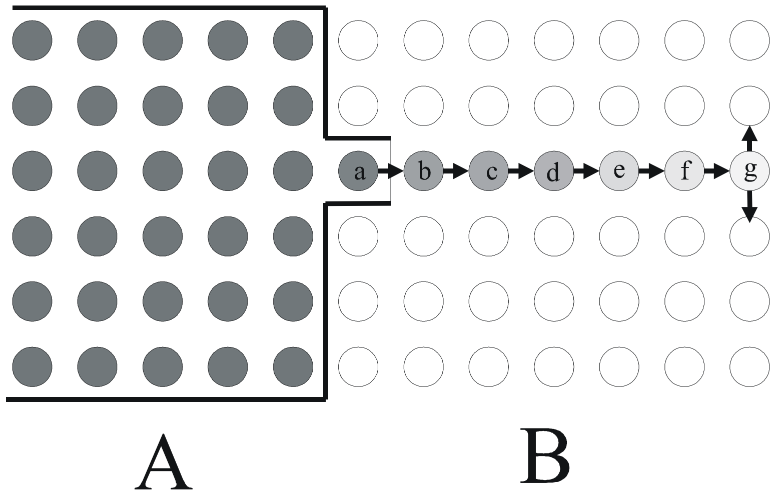
The parameter Δ_c is the **threshold of homogeneity** of the regions.

Problems with “Region Growing“

It is possible that a sequence of pixels located at the border of the current region and fulfilling

$$| c_{i+1} - c_i \prec \Delta_c$$

spoils the edge.

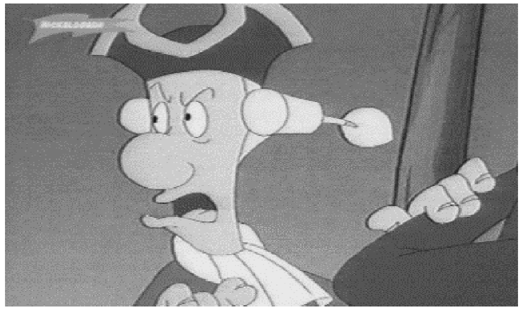


Note: There exists a dual algorithm to region growing, called **region splitting,** as well as a combination of both to the so-called **split-and-merge** algorithm. In practice, the latter often works best.

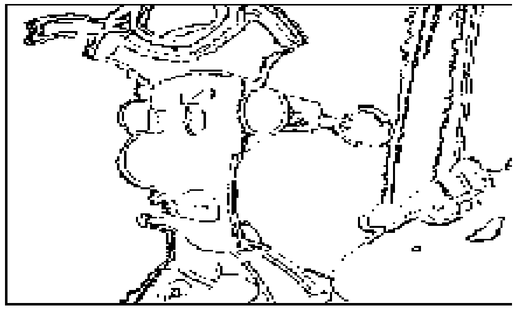
The threshold of homogeneity is critical!

A good setting of the threshold of homogeneity Δ_c is crucial for the algorithm to work. A wrong parameter value leads to over- or under-segmented images:

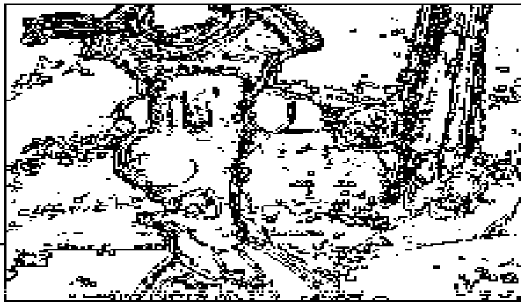
original image



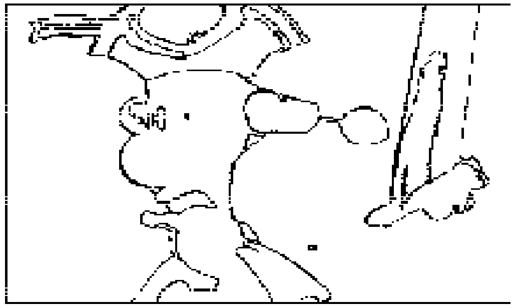
good segmentation



over-segmentation

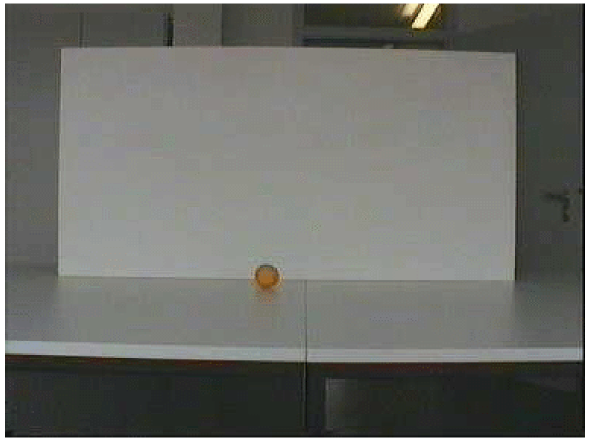


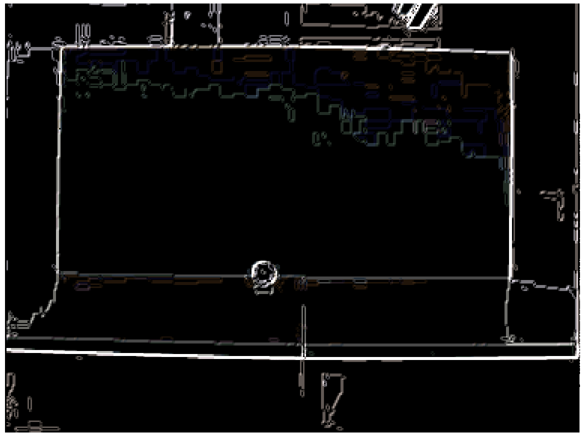
under-segmentation



Segmentation Example

A colored ball rolls in front of a gray background. It can be segmented by means of edge detection. The image below has been obtained with the algorithm **region growing.**


original image


segmented image

Object Segmentation

Edge detection is the basis for many **object segmentation** algorithms. Object segmentation tries to identify semantic objects within an image.

Unfortunately, the edges, as derived by the basic algorithms described so far, hardly ever correspond to the real borderlines of a semantic object! Problems are due to:

- objects that consist of several regions of different colors (e.g., a person with a white shirt and blue pants),
- partial occlusion of an object,
- objects at image boundaries that are only partially visible,
- objects that change their shape over time (e.g., people),
- etc, etc.

We conclude that **object segmentation is one of the hardest problems in image analysis and computer vision**.

Analysis of Frame Sequences

The analysis of frame (or image) sequences can help to obtain a better understanding of video content.

The **movement of an object** can carry semantics. For example, a zigzag movement could be characteristic for a downhill skier.

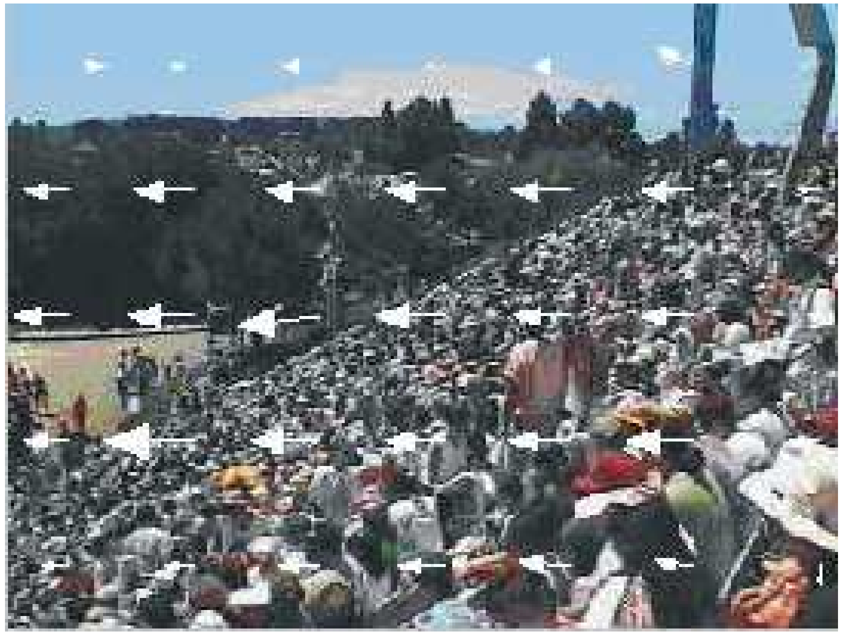
In many cases, motion detection can facilitate object segmentation. The human visual system is highly based on motion for object segementation; for example, a bird in a tree is detected only when it moves. This is also true for the visual system of many animals.

A **camera operation** (e.g., panning, zooming) can be distinguished from the movement of an object by the fact that **all** image pixels are affected in a deterministic and computable manner. An example is the lateral motion of all pixels during panning. Thus, camera operations can often be detected automatically.

Motion Vectors

Modern compression algorithms for video often compute **motion vectors** for pixel blocks. Typical examples are MPEG-1, MPEG-2, H.261 and H.263. The motion of macro blocks does not help much for object segmentation but it allows the detection of camera operations. An advantage is that the required data is readily available for our analysis, without separate (and costly) computation.

Example:

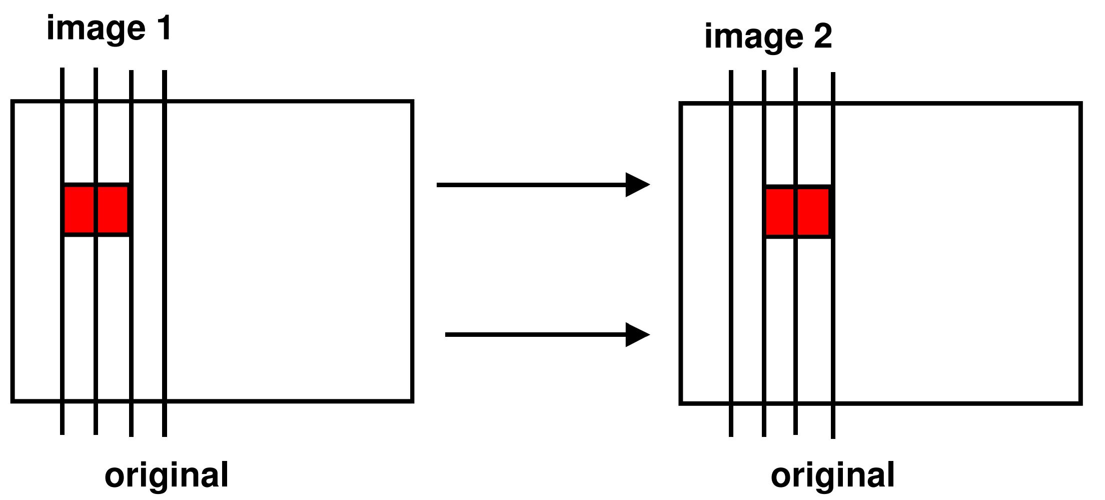


Block-based Motion Vectors

If a monochrome, two-dimensional object moves across the image, motion detection works well only at:

- the edges of the object, and
- only in the direction of the movement.

Example:

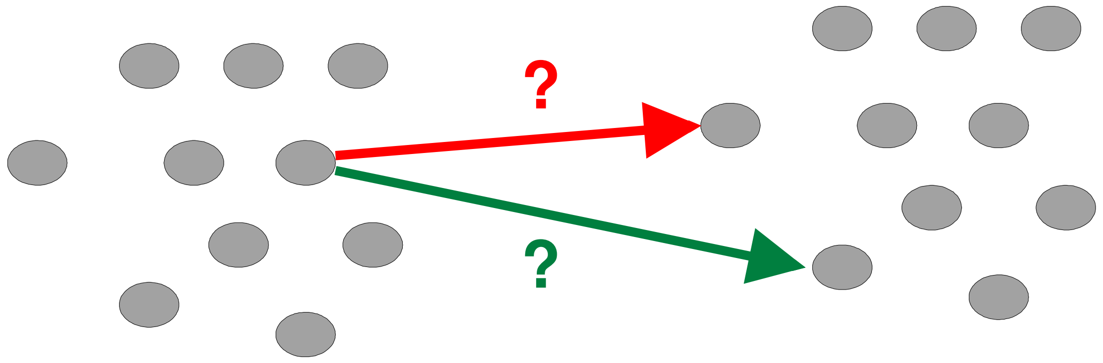


Optical Flow

The movement of objects in the real world is reflected by color changes in a frame sequence. For simplification, one often restricts the analysis to gray-scale images. The notion of **optical flow** describes the movement of gray-scale patterns in a frame sequence.

In a first step, a motion vector is assigned to each pixel. A continuous **vector field** of all these motion vectors is then computed which represents the optic flow. Both steps make assumptions (e.g., a constant intensity of light), and both are error-prone. A large number of algorithms for the computation of optical flow exists in the literature.

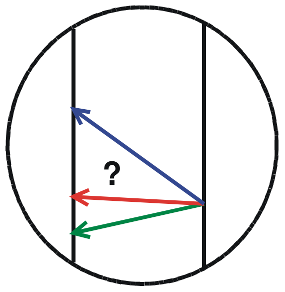
One major difficulty is the determination of the movement of one particular pixel between frame *i* and frame *i*+1 (“correspondence problem “), as shown below.



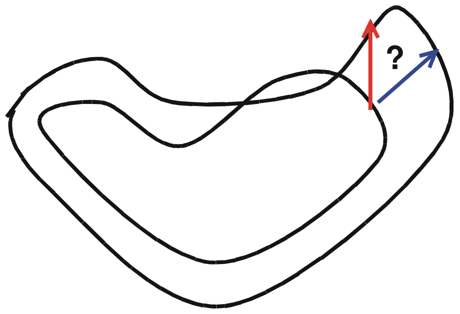
Problems in the Computation of Optical Flow

Unfortunately, many other problems also complicate the computation of optical flow.

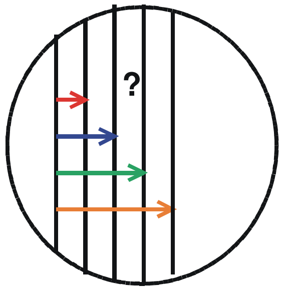
Examples:



aperture problem



deformable objects



periodic structures

We conclude that the optical flow is difficult to compute, and is often not a reliable indicator for the motion of an object.

Edge Change Ratio

Once the edges contained in an image are computed (see the section on "edge detection" above), the **edge change ratio** between two successive frames *i* and *i*+1 can be determined. We first compute the pixels of frame *i* that are located on edges; their number is denoted by s_i. Similarly, we compute the pixels of frame *i*+1 that are located on edges; their number is denoted by s_{i+1}. Now we compute the number of pixels that are located on edges in frame *i* but not in frame *i*+1 (i.e., the vanishing edges E_{out}) and the edge pixels in frame *i*+1 that were not on edges in frame *i* (i.e., new edge pixels E_{in}).

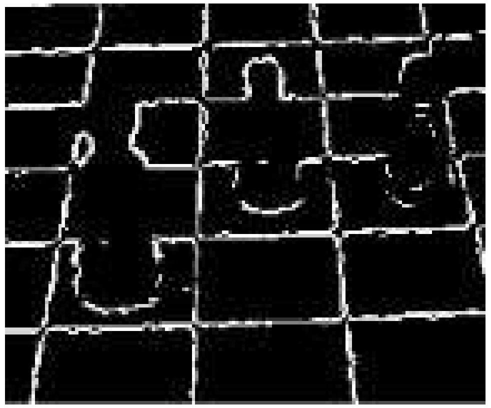
The **edge change ratio (ECR)** between frame *i* and frame *i*+1 is then defined as:

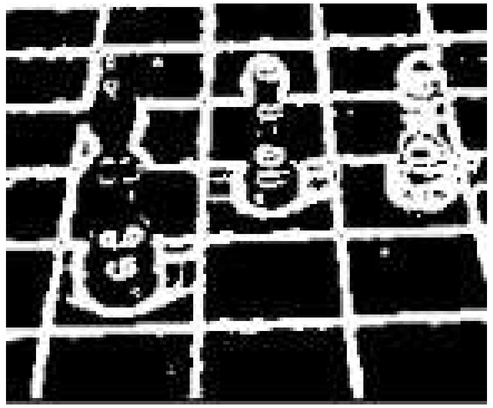
$$ECR_i = \max\left(\frac{E_{in}}{s_{i+1}}, \frac{E_{out}}{s_i}\right)$$

For example, the ECR can be used as a simple measure of motion intensity.

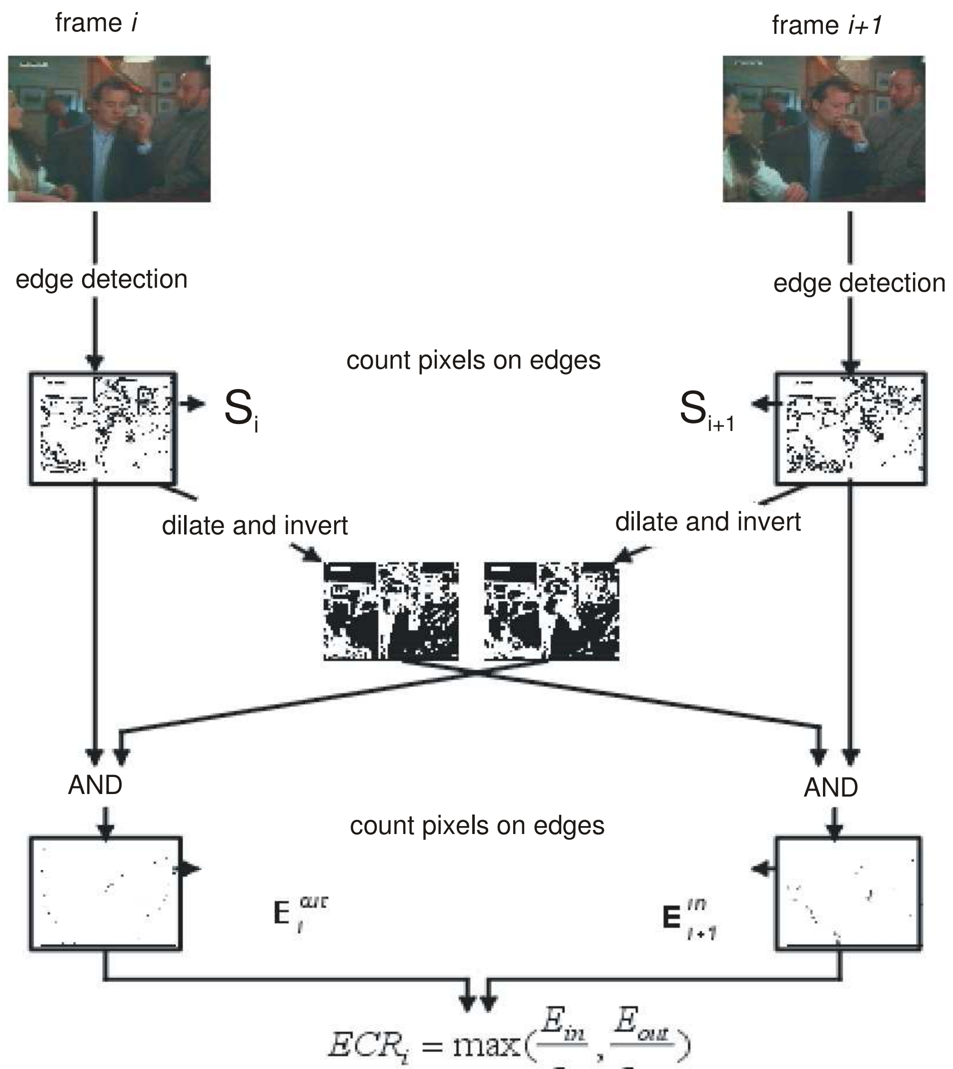
ECR: Improvement by Widening

In order to increase the robustness of this measure with regard to noise and jitter, the edges are often artificially widened.





Algorithm "Compute ECR"



$$ECR_i = \max\left(\frac{E_{in}}{S_i}, \frac{E_{out}}{S_{i+1}}\right)$$