HALCON Version 6.1.4

MVTec Software GmbH

HALCON Application Guide

Application Guide for HALCON, Version 6.1.4.

Edition 1	June 2002	(HALCON 6.1)
Edition 1a	May 2003	(HALCON 6.1.2)

More information about HALCON can be found at:

`http://www.mvtec.com/halcon/`

Contents

Application Note

How to Use Shape-Based Matching to Find and Localize Objects

Provided Functionality

▷ Finding objects starting based on a single model image

▷ Localizing objects with subpixel accuracy

Typical Applications

▷ Object recognition and localization

▷ Intermediate machine vision steps, e.g., alignment of ROIs

▷ Completeness check

▷ Parts inspection

Involved Operators

create_shape_model, create_scaled_shape_model
inspect_shape_model, get_shape_model_params
set_shape_model_origin, get_shape_model_origin
find_shape_model, find_shape_models
find_scaled_shape_model, find_scaled_shape_models
write_shape_model, read_shape_model
clear_shape_model, clear_all_shape_models

Overview

HALCON's operators for *shape-based matching* enable you to find and localize objects based on a single model image, i.e., from a *model*. This method is robust to noise, clutter, occlusion, and arbitrary non-linear illumination changes. Objects are localized with subpixel accuracy in 2D, i.e., found even if they are rotated or scaled.

The process of shape-based matching (see section 1 for a quick overview) is divided into two distinct phases: In a first phase, you specify and create the model. This model can be stored in a file to be reused in different applications. Detailed information about this phase can be found in section 2. In the second phase, the model is used to find and localize an object. Section 3 describes how to optimize the outcome of this phase by restricting the search space.

Shape-based matching is a powerful tool for various machine vision tasks, ranging from intermediate image processing, e.g., to place ROIs automatically or to align them to a moving part, to complex tasks, e.g., recognize and localize a part in a robot vision application. Examples can be found in section 4.

Unless specified otherwise, the example programs can be found in the subdirectory `shape_matching` of the directory `%HALCONROOT%\examples\application_guide`.

Contents

1 A First Example

In this section we give a quick overview of the matching process. To follow the example actively, start the HDevelop program `hdevelop\first_example_shape_matching.dev`, which locates the print on an IC; the steps described below start after the initialization of the application (press $\boxed{\text{F5}}$ once to reach this point).

Step 1: Select the object in the model image

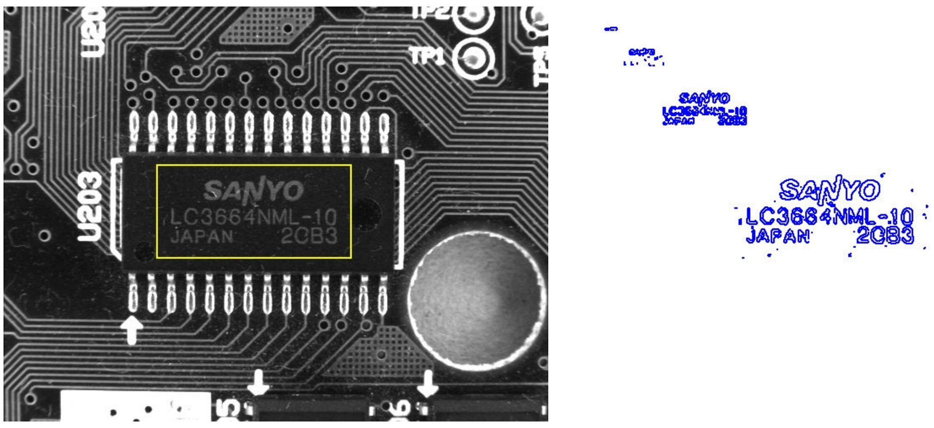
```
Row1 := 188
Column1 := 182
Row2 := 298
Column2 := 412
gen_rectangle1 (ROI, Row1, Column1, Row2, Column2)
reduce_domain (ModelImage, ROI, ImageROI)
```

After grabbing the so-called *model image*, i.e., a representative image of the object to find, the first task is to create a region containing the object. In the example program, a rectangular region is created using the operator `gen_rectangle1`; alternatively, you can draw the region interactively using, e.g., `draw_rectangle1` or use a region that results from a previous segmentation process. Then, an image containing just the selected region is created using the operator `reduce_domain`. The result is shown in figure 1.

Step 2: Create the model

```
inspect_shape_model (ImageROI, ShapeModelImages, ShapeModelRegions, 8, 30)
create_shape_model (ImageROI, NumLevels, 0, rad(360), 0, 'none',
                    'use_polarity', 30, 10, ModelID)
```

With the operator `create_shape_model`, the so-called *model* is created, i.e., the internal data structure describing the searched object. Before this, we recommend to apply the operator `inspect_shape_model`, which helps you to find suitable parameters for the model creation. `inspect_shape_model` shows the effect of two parameters: the *number of pyramid levels* on which the model is created, and the minimum *contrast* that object points must have to be included in the model. As a result, the operator `inspect_shape_model` returns the model points



① ②

Figure 1: ① specifying the object; ② the internal model (4 pyramid levels).

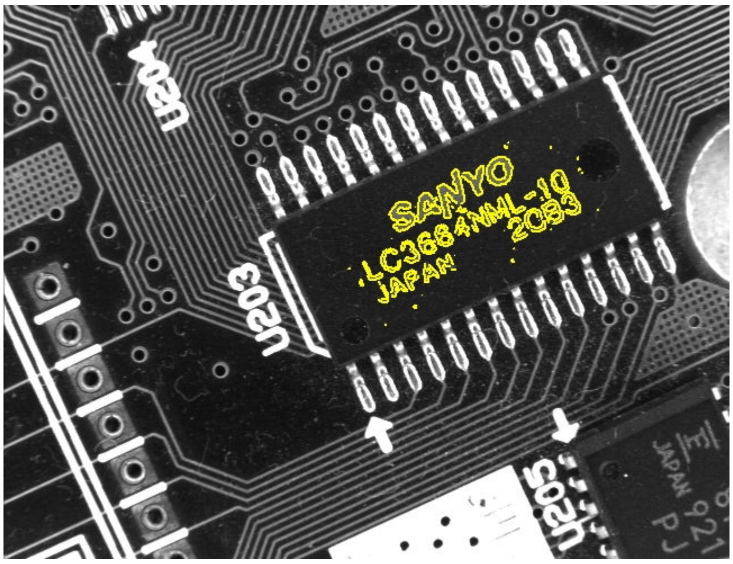


Figure 2: Finding the object in other images.

on the selected pyramid levels as shown in figure 1; thus, you can check whether the model contains the relevant information to describe the object of interest.

When actually creating the model with the operator `create_shape_model`, you can specify additional parameters besides `NumLevels` and `Contrast`: First of all, you can restrict the range of angles the object can assume (parameters `AngleStart` and `AngleExtent`) and the angle steps at which the model is created (`AngleStep`). With the help of the parameter `Optimization` you can reduce the number of model points; this is useful in the case of very large models. The parameter `Metric` lets you specify whether the *polarity* of the model points must be observed. Finally, you can specify the minimum contrast object points must have in the *search images* to be compared with the model (`MinContrast`). The creation of the model is described in detail in section 2.

As a result, the operator `create_shape_model` returns a *handle* for the newly created model (`ModelID`), which can then be used to specify the model, e.g., in calls to the operator `find_shape_model`. Note that if you use HALCON's COM or C++ interface and call the operator via the classes `HShapeModelX` or `HShapeModel`, no handle is returned because the instance of the class itself acts as your handle.

If not only the orientation but also the scale of the searched object is allowed to vary, you must use the operator `create_scaled_shape_model` to create the model; then, you can describe the allowed range of scaling with three parameters similar to the range of angles.

Step 3: Find the object again

```
for i := 1 to 20 by 1
  grab_image (SearchImage, FGHandle)
  find_shape_model (SearchImage, ModelID, 0, rad(360), 0.8, 1, 0.5,
                    'interpolation', 0, 0.9, RowCheck, ColumnCheck,
                    AngleCheck, Score)
endfor
```

To find the object again in a *search image*, all you need to do is call the operator `find_shape_model`; figure 2 shows the result for one of the example images. Besides the already mentioned ModelID, find_shape_model provides further parameters to optimize the search process: The parameters AngleStart, AngleExtent, and NumLevels, which you already specified when creating the model, allow you to use more restrictive values in the search process; by using the value 0 for NumLevels, the value specified when creating the model is used. With the parameter MinScore you can specify how many of the model points must be found; a value of 0.5 means that half of the model must be found. Furthermore, you can specify how many instances of the object are expected in the image (NumMatches) and how much two instances of the object may overlap in the image (MaxOverlap). To compute the position of the found object with subpixel accuracy the parameter SubPixel should be set to a value different from 'none'. Finally, the parameter Greediness describes the used search heuristics, ranging from "safe but slow" (value 0) to "fast but unsafe" (value 1). How to optimize the search process is described in detail in section 3.

The operator find_shape_model returns the position and orientation of the found object instances in the parameters Row, Column, and Angle, and their coresponding Score, i.e., how much of the model was found.

If you use the operator find_scaled_shape_model (after creating the model using create_scaled_shape_model), the scale of the found object is returned Scale.

2 Creating a Suitable Model

A prerequisite for a successful matching process is, of course, a suitable model for the object you want to find. A model is suitable if it describes the *significant* parts of the object, i.e., those parts that characterize it and allow to discriminate it clearly from other objects or from the background. On the other hand, the model should not contain clutter, i.e., points not belonging to the object (see, e.g., figure 4).

2.1 A Closer Look at the Region of Interest

When creating the model, the first step is to select a *region of interest* (ROI), i.e., the part of the image which serves as the model. In HALCON, a *region* defines an area in an image or, more generally, a set of points. A region can have an arbitrary shape; its points do not even need to be connected. Thus, the region of the model can have an arbitrary shape as well.

The sections below describe how to create simple and more complex regions. The following code fragment shows the typical next steps after creating an ROI:

```
reduce_domain (ModelImage, ROI, ImageROI)
create_shape_model (ImageROI, 0, 0, rad(360), 0, 'none', 'use_polarity',
                    30, 10, ModelID)
```

Note that the region of interest used when creating a shape model influences the matching results: Its center of gravity is used as the *reference point* of the model (see section 4 for more information).

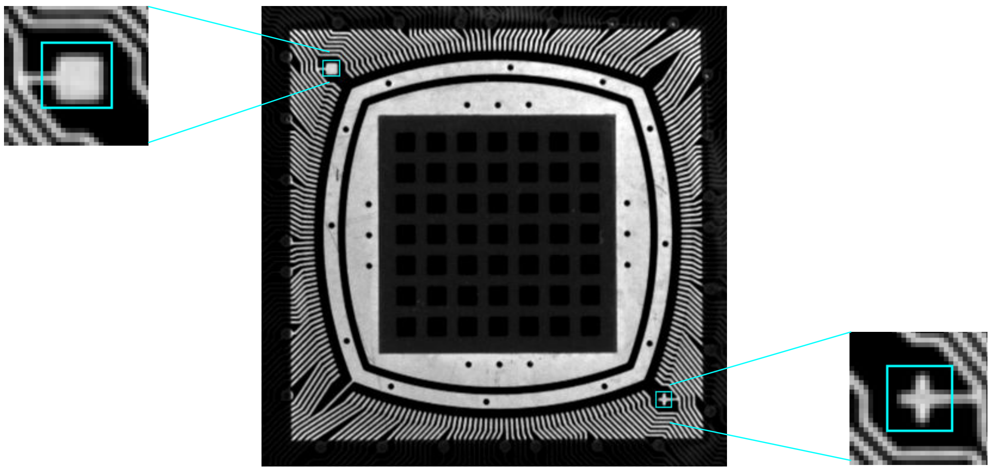


Figure 3: Creating an ROI from two regions.

2.1.1 How to Create a Region

HALCON offers multiple operators to create regions, ranging from standard shapes like rectangles (`gen_rectangle2`) or ellipses (`gen_ellipse`) to free-form shapes (e.g., `gen_region_polygon_filled`). These operators can be found in the HDevelop menu `Operators` ▷ `Regions` ▷ `Creation`.

However, to use these operators you need the "parameters" of the shape you want to create, e.g., the position, size and, orientation of a rectangle or the position and radius of a circle. Therefore, they are typically combined with the operators in the HDevelop menu `Operators` ▷ `Graphics` ▷ `Drawing`, which let you draw a shape on the displayed image and then return the shape parameters:

```
draw_rectangle1 (WindowHandle, ROIRow1, ROIColumn1, ROIRow2, ROIColumn2)
gen_rectangle1 (ROI, ROIRow1, ROIColumn1, ROIRow2, ROIColumn2)
```

2.1.2 How to Combine and Mask Regions

You can create more complex regions by adding or subtracting standard regions using the operators `union2` and `difference`. For example, to create an ROI containing the square and the cross in figure 3, the following code fragment was used:

```
draw_rectangle1 (WindowHandle, ROI1Row1, ROI1Column1, ROI1Row2,
                 ROI1Column2)
gen_rectangle1 (ROI1, ROI1Row1, ROI1Column1, ROI1Row2, ROI1Column2)
draw_rectangle1 (WindowHandle, ROI2Row1, ROI2Column1, ROI2Row2,
                 ROI2Column2)
gen_rectangle1 (ROI2, ROI2Row1, ROI2Column1, ROI2Row2, ROI2Column2)
union2 (ROI1, ROI2, ROI)
```

Similarly, you can subtract regions using the operator `difference`. This method is useful to "mask" those parts of a region containing clutter, i.e., high-contrast points that are not part of the object. In figure 4, e.g., the task is to find the three capacitors. When using a single circular ROI, the created model contains many clutter points, which are caused by reflections on the

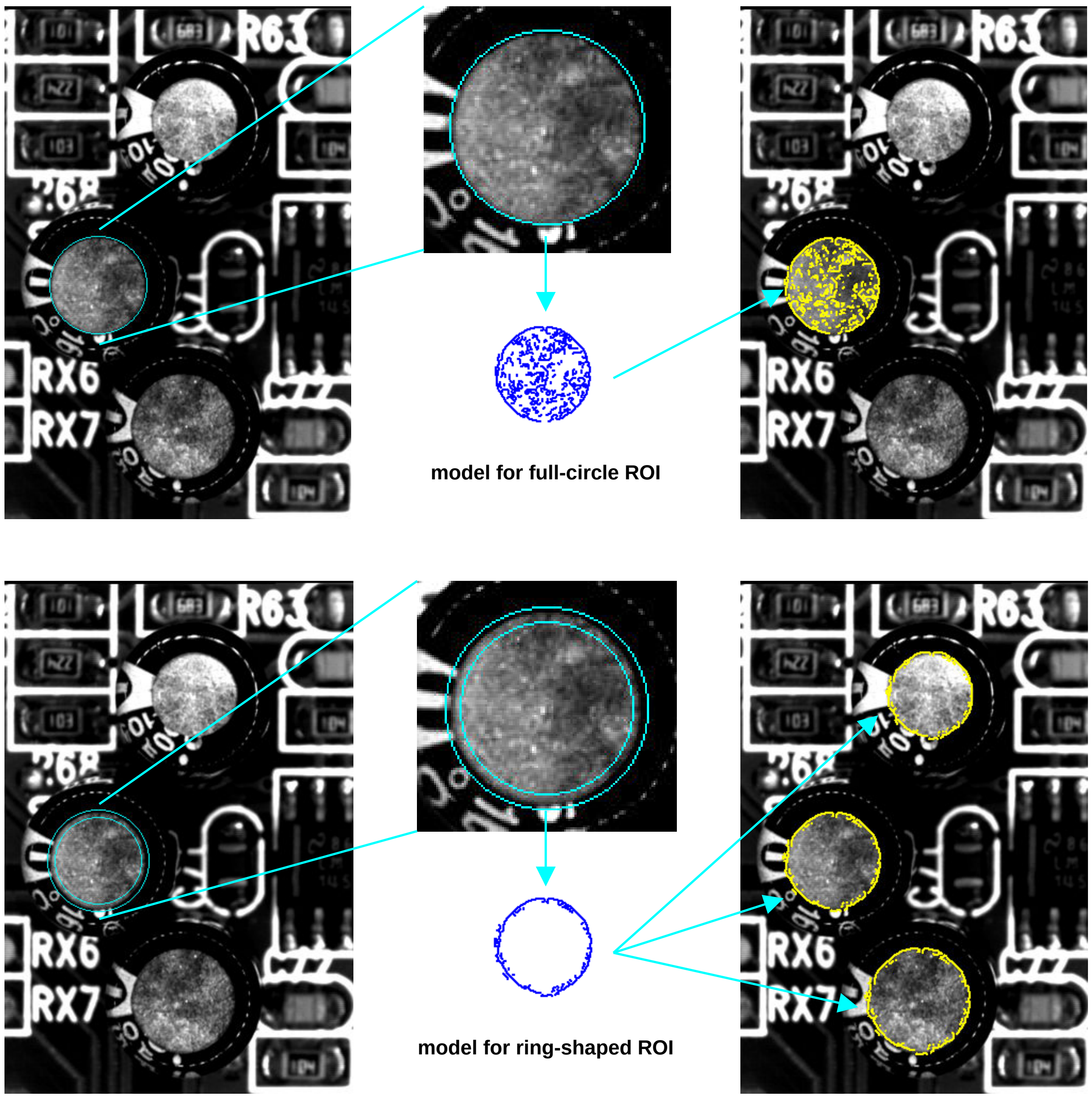


Figure 4: Masking the part of a region containing clutter.

metallic surface. Thus, the other two capacitors are not found. The solution to this problem is to use a ring-shaped ROI, which can be created by the following lines of code:

```
draw_circle (WindowHandle, ROI1Row, ROI1Column, ROI1Radius)
gen_circle (ROI1, ROI1Row, ROI1Column, ROI1Radius)
gen_circle (ROI2, ROI1Row, ROI1Column, ROI1Radius-8)
difference (ROI1, ROI2, ROI)
```

Note that the ROI should not be too “thin”, otherwise it vanishes at higher pyramid levels! As a rule of thumb, an ROI should be $2^{NumLevels-1}$ pixels wide; in the example, the width of 8 pixels therefore allows to use 4 pyramid levels.

☞ For this task even better results can be obtained by **using a synthetic model image**. This is described in section 2.3.

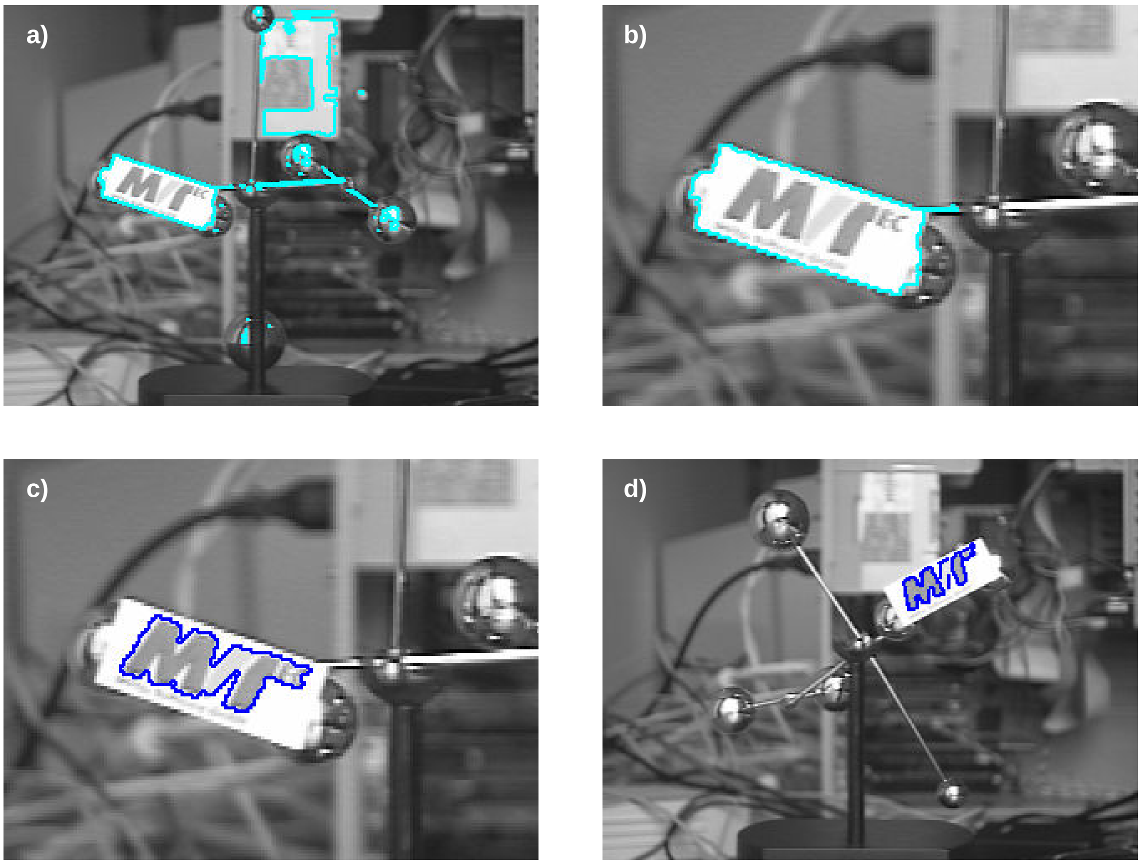


Figure 5: Using image processing to create an ROI: a) extract bright regions; b) select the card; c) the logo forms the ROI; d) result of the matching.

2.1.3 Using Image Processing to Create and Modify Regions

In the previous sections, regions were created explicitely by specifying their shape parameters. Especially for complex ROIs this method can be inconvenient and time-consuming. In the following, we therefore show you how to extract and modify regions using image processing operators.

Example 1: Determining the ROI Using Blob Analysis

To follow the example actively, start the HDevelop program `hdevelop\create_roi_via_vision.dev`, which locates the MVTec logo on a pendulum (see figure 5); we start after the initialization of the application (press $\boxed{\texttt{F5}}$ once). The main idea is to “zoom in” on the desired region in multiple steps: First, find the bright region corresponding to the card, then extract the dark characters on it.

Step 1: Extract the bright regions

```
threshold (ModelImage, BrightRegions, 200, 255)
connection (BrightRegions, ConnectedRegions)
fill_up (ConnectedRegions, FilledRegions)
```

First, all bright regions are extracted using a simple thresholding operation (`threshold`); the operator `connection` forms connected components. The extracted regions are then filled up via fill_up; thus, the region corresponding to the card also encompasses the dark characters (see figure 5a).

Step 2: Select the region of the card

```
select_shape (FilledRegions, Card, 'area', 'and', 1800, 1900)
```

The region corresponding to the card can be selected from the list of regions with the operator select_shape. In HDevelop, you can determine suitable features and values using the dialog Visualization ▷ Region Info; just click into a region, and the dialog immediately displays its feature values. Figure 5b shows the result of the operator.

Step 3: Use the card as an ROI for the next steps

```
reduce_domain (ModelImage, Card, ImageCard)
```

Now, we can restrict the next image processing steps to the region of the card using the operator reduce_domain. This iterative focusing has an important advantage: In the restricted region of the card, the logo characters are much easier to extract than in the full image.

Step 4: Extract the logo

```
threshold (ImageCard, DarkRegions, 0, 230)
connection (DarkRegions, ConnectedRegions)
select_shape (ConnectedRegions, Characters, 'area', 'and', 150, 450)
union1 (Characters, CharacterRegion)
```

The logo characters are extracted similarly to the card itself; as a last step, the separate character regions are combined using the operator union1.

Step 5: Enlarge the region using morphology

```
dilation_circle (CharacterRegion, ROI, 1.5)
reduce_domain (ModelImage, ROI, ImageROI)
create_shape_model (ImageROI, 0, 0, rad(360), 0, 'none', 'use_polarity',
                    30, 10, ModelID)
```

Finally, the region corresponding to the logo is enlarged slightly using the operator dilation_circle. Figure 5c shows the resulting ROI, which is then used to create the shape model.

Example 2: Further Processing the Result of inspect_shape_model

You can also combine the interactive ROI specification with image processing. A useful method in the presence of clutter in the model image is to create a first model region interactively and then process this region to obtain an improved ROI. Figure 6 shows an example; the task is to locate the arrows. To follow the example actively, start the HDevelop program hdevelop\process_shape_model.dev; we start after the initialization of the application (press F5 once).

Step 1: Select the arrow

```
gen_rectangle1 (ROI, 361, 131, 406, 171)
```

First, an initial ROI is created around the arrow, without trying to exclude clutter (see figure 6a).

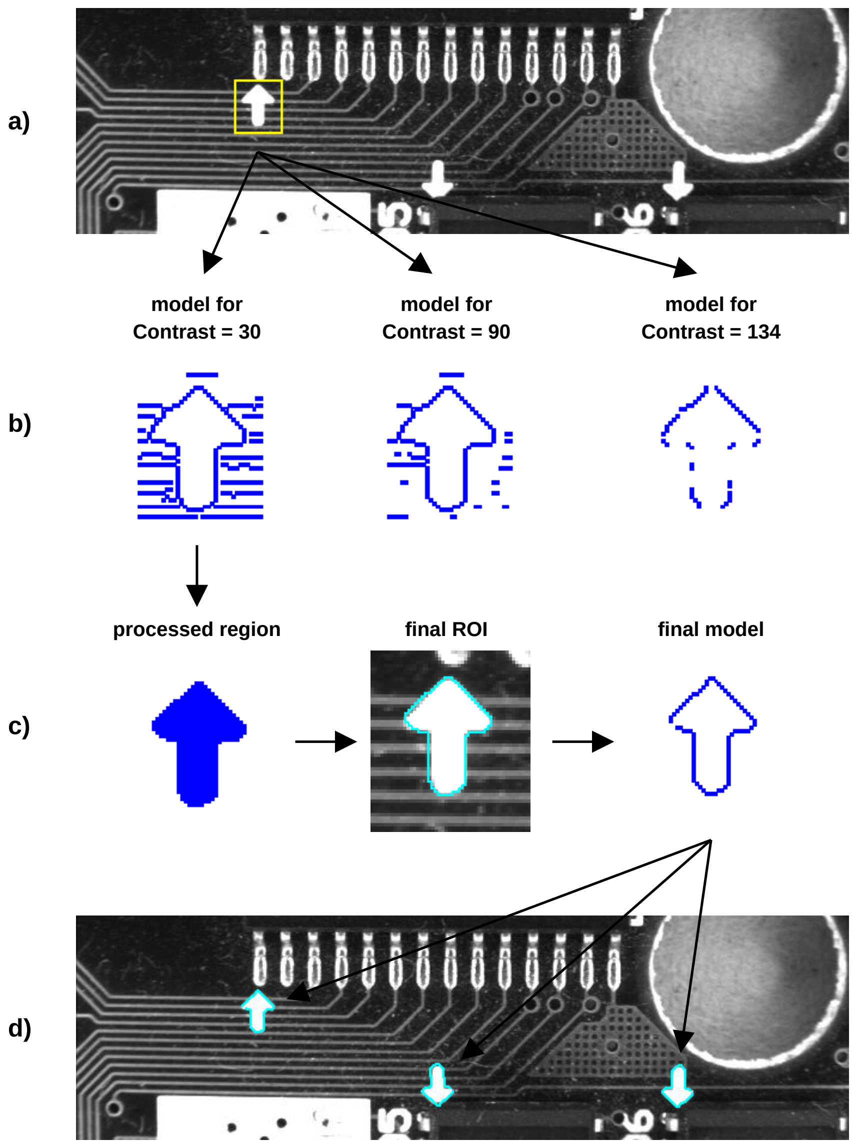


Figure 6: Processing the result of `inspect_shape_model`: a) interactive ROI; b) models for different values of `Contrast`; c) processed model region and corresponding ROI and model; d) result of the search.

Step 2: Create a first model region

```
reduce_domain (ModelImage, ROI, ImageROI)
inspect_shape_model (ImageROI, ShapeModelImage, ShapeModelRegion, 1, 30)
```

Figure 6b shows the shape model regions that would be created for different values of the parameter `Contrast`. As you can see, you cannot remove the clutter without losing characteristic points of the arrow itself.

Step 3: Process the model region

```
fill_up (ShapeModelRegion, FilledModelRegion)
opening_circle (FilledModelRegion, ROI, 3.5)
```

You can solve this problem by exploiting the fact that the operator `inspect_shape_model` returns the shape model region; thus, you can process it like any other region. The main idea to get rid of the clutter is to use morphological operator opening_circle, which eliminates small regions. Before this, the operator fill_up must be called to fill the inner part of the arrow, because only the boundary points are part of the (original) model region. Figure 6c shows the resulting region.

Step 4: Create the final model

```
reduce_domain (ModelImage, ROI, ImageROI)
create_shape_model (ImageROI, 3, 0, rad(360), 0, 'none', 'use_polarity',
                    30, 10, ModelID)
```

The processed region is then used to create the model; figure 6c shows the corresponding ROI and the final model region. Now, all arrows are located successfully.

2.1.4 How the ROI Influences the Search

Note that the ROI used when creating the model also influences the results of the subsequent matching: The center point of the ROI acts as the so-called *point of reference* of the model for the estimated position, rotation, and scale. You can query the reference point using the operator get_shape_model_origin and modify it using set_shape_model_origin; please refer to sections 3.4 and 4.3 for additional information.

The point of reference also influences the search itself: An object is only found if the point of reference lies within the image, or more exactly, within the *domain* of the image (see also section 3.1.1). Please note that this test is always performed for the original point of reference, i.e., the center point of the ROI, even if you modified the reference point using set_shape_model_origin.

2.2 Which Information is Stored in the Model?

As the name *shape-based pattern matching* suggests, objects are represented and recognized by their *shape*. There exist multiple ways to determine or describe the shape of an object. Here, the shape is extracted by selecting all those points whose *contrast* exceeds a certain threshold; typically, the points correspond to the contours of the object (see, e.g., figure 1). Section 2.2.1 takes a closer look at the corresponding parameters.

To speed up the matching process, a so-called *image pyramid* is created, consisting of the original, full-sized image and a set of downsampled images. The model is then created and searched on the different pyramid levels (see section 2.2.2 for details).

If the object is allowed to appear rotated or scaled, the corresponding information is used already when creating the model. This also speeds up the matching process, at the cost of higher memory requirements for the created model. Section 2.2.3 and section 2.2.4 describe the corresponding parameters.

In the following, all parameters belong to the operator create_shape_model if not stated otherwise.

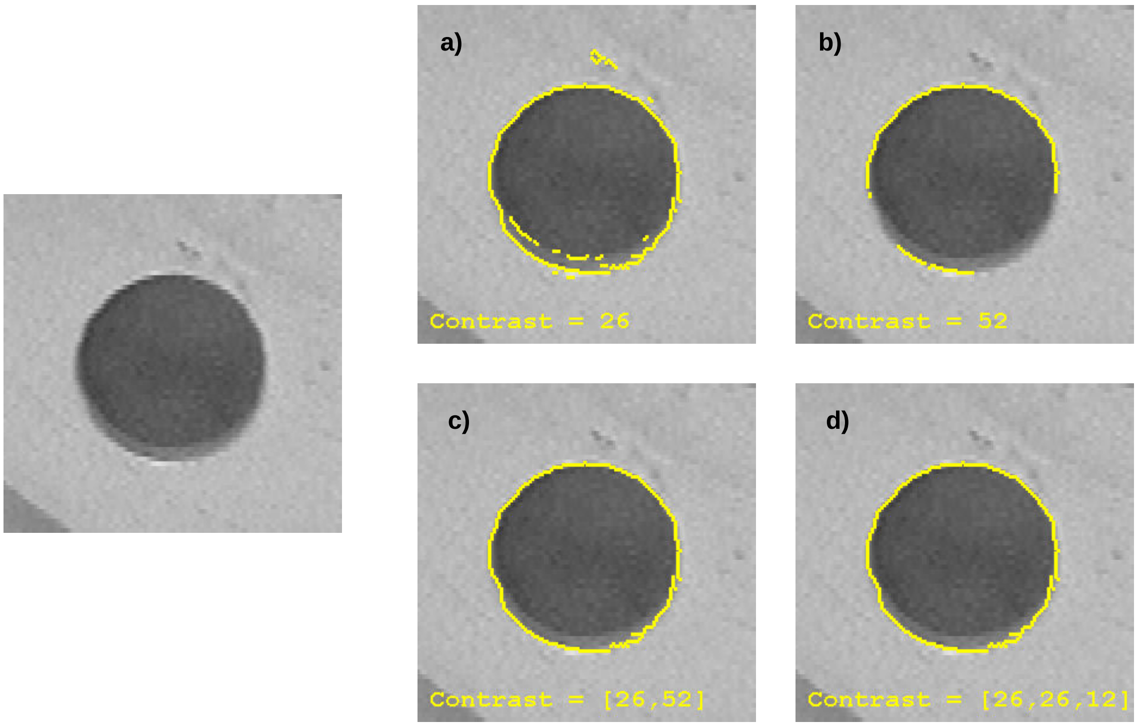


Figure 7: Selecting significant pixels via Contrast: a) complete object but with clutter; b) no clutter but incomplete object; c) hysteresis threshold; d) minimum contour size.

2.2.1 Which Pixels are Part of the Model?

For the model those pixels are selected whose *contrast*, i.e., gray value difference to neighboring pixels, exceeds a threshold specified by the parameter Contrast when calling create_shape_model. In order to obtain a suitable model the contrast should be chosen in such a way that the *significant* pixels of the object are included, i.e., those pixels that characterize it and allow to discriminate it clearly from other objects or from the background. Obviously, the model should not contain clutter, i.e., pixels that do not belong to the object.

In some cases it is impossible to find a single value for Contrast that removes the clutter but not also parts of the object. Figure 7 shows an example; the task is to create a model for the outer rim of a drill-hole: If the complete rim is selected, the model also contains clutter (figure 7a); if the clutter is removed, parts of the rim are missing (figure 7b).

To solve such problems, the parameter Contrast provides two additional methods: hysteresis thresholding and selection of contour parts based on their size. Both methods are used by specifying a *tuple* of values for Contrast instead of a single value.

Hysteresis thresholding (see also the operator hysteresis_threshold) uses two thresholds, a lower and an upper threshold. For the model, first pixels that have a contrast higher than the upper threshold are selected; then, pixels that have a contrast higher than the lower threshold and that are connected to a high-contrast pixel, either directly or via another pixel with contrast above the lower threshold, are added. This method enables you to select contour parts whose contrast varies from pixel to pixel. Returning to the example of the drill-hole: As you can see in figure 7c, with a hysteresis threshold you can create a model for the complete rim without clutter. The following line of code shows how to specify the two thresholds in a tuple:

```
inspect_shape_model (ImageROI, ModelImages, ModelRegions, 1, [26,52])
```

The second method to remove clutter is to specify a minimum size, i.e., number of pixels, for the contour components. Figure 7d shows the result for the example task. The minimum size must be specified in the third element of the tuple; if you don't want to use a hysteresis threshold, set the first two elements to the same value:

```
inspect_shape_model (ImageROI, ModelImages, ModelRegions, 1, [26,26,12])
```

Alternative methods to remove clutter are to modify the ROI as described in section 2.1 or create a synthetic model (see section 2.3).

2.2.2 How Subsampling is Used to Speed Up the Search

To speed up the matching process, a so-called *image pyramid* is created, both for the model image and for the search images. The pyramid consists of the original, full-sized image and a set of downsampled images. For example, if the the original image (first pyramid level) is of the size 600x400, the second level image is of the size 300x200, the third level 150x100, and so on. The object is then searched first on the highest pyramid level, i.e., in the smallest image. The results of this fast search are then used to limit the search in the next pyramid image, whose results are used on the next lower level until the lowest level is reached. Using this iterative method, the search is both fast and accurate. Figure 8 depicts 4 levels of an example image pyramid together with the corresponding model regions.

You can specify how many pyramid levels are used via the parameter `NumLevels`. We recommend to choose the highest pyramid level at which the model contains at least 10-15 pixels and in which the shape of the model still resembles the shape of the object. You can inspect the model image pyramid using the operator `inspect_shape_model`, e.g., as shown in the HDevelop program `hdevelop\first_example_shape_matching.dev`:

```
inspect_shape_model (ImageROI, ShapeModelImages, ShapeModelRegions, 8, 30)
area_center (ShapeModelRegions, AreaModelRegions, RowModelRegions,
             ColumnModelRegions)
HeightPyramid := |ShapeModelRegions|
for i := 1 to HeightPyramid by 1
  if (AreaModelRegions[i-1] >= 15)
    NumLevels := i
  endif
endfor
create_shape_model (ImageROI, NumLevels, 0, rad(360), 0, 'none',
                    'use_polarity', 30, 10, ModelID)
```

After the call to the operator, the model regions on the selected pyramid levels are displayed in HDevelop's Graphics Window; you can have a closer look at them using the online zooming (menu entry `Visualization` ▷ `Online Zooming`). The code lines following the operator call loop through the pyramid and determine the highest level on which the model contains at least 15 points. This value is then used in the call to the operator `create_shape_model`.

A much easier method is to **let HALCON select a suitable value itself** by specifying the value `0` for `NumLevels`. You can then query the used value via the operator `get_shape_model_params`.

The operator `inspect_shape_model` returns the pyramid images in form of an image tuple (array); the individual images can be accessed like the model regions with the operator `select_obj`. Please note that **object tuples start with the index 1, whereas control parame-**

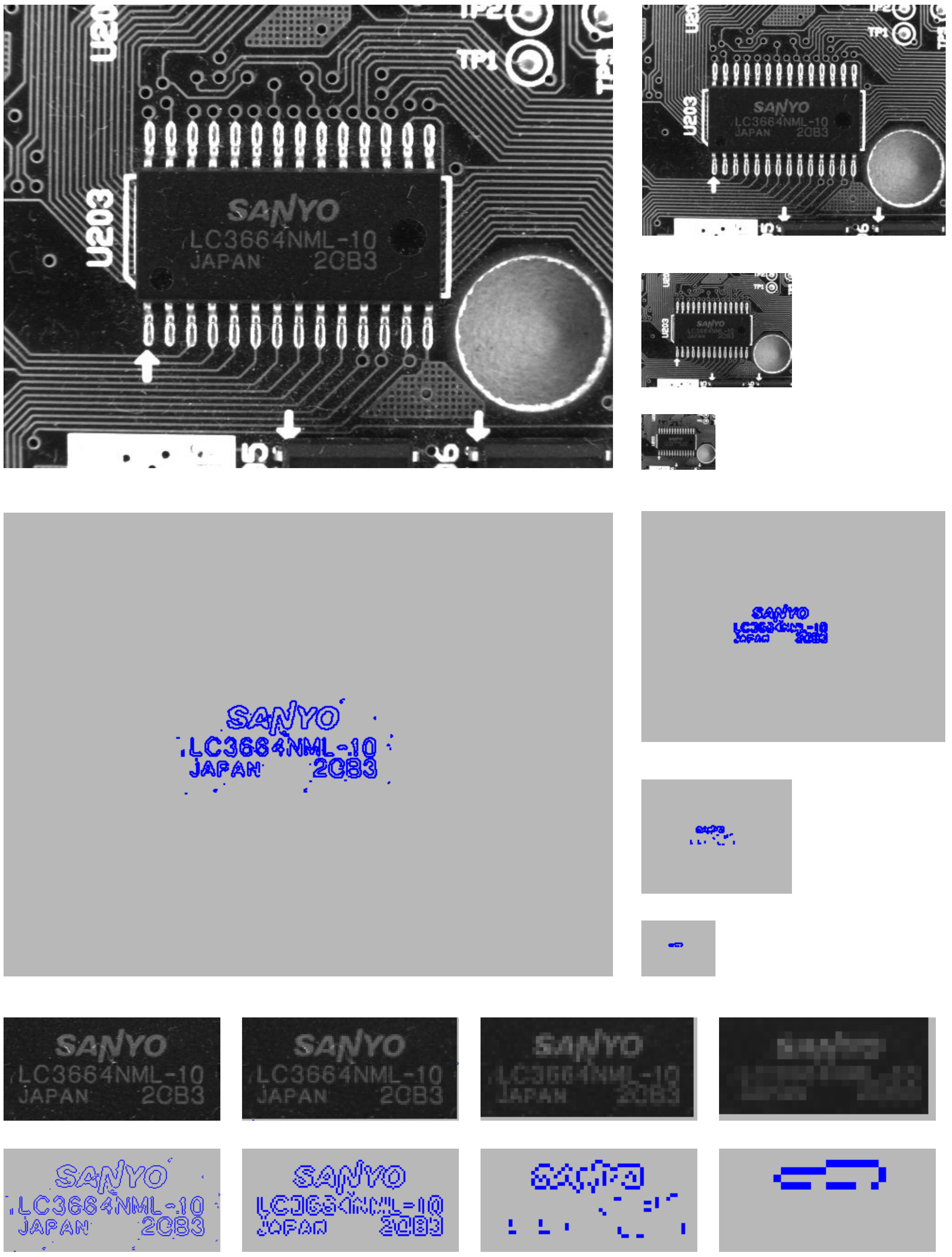


Figure 8: The image and the model region at four pyramid levels (original size and zoomed to equal size).

ter tuples start with the index 0!

You can enforce a further reduction of model points via the parameter Optimization. This may be useful to speed up the matching in the case of particularly large models. Please note that regardless of your selection all points passing the contrast criterion are displayed, i.e., you cannot check which points are part of the model.

2.2.3 Allowing a Range of Orientation

If the object's rotation may vary in the search images you can specify the allowed range in the parameter `AngleExtent` and the starting angle of this range in the parameter `AngleStart` (unit: rad). Note that the range of rotation is defined relative to the model image, i.e., a starting angle of `0` corresponds to the orientation the object has in the model image. Therefore, to allow rotations up to +/-5°, e.g., you should set the starting angle to `-rad(5)` and the angle extent to `rad(10)`.

We recommend to limit the allowed range of rotation as much as possible in order to speed up the search process and to minimize the required memory. Note that you can further limit the allowed range when calling the operator `find_shape_model` (see section 3.1.2). If you want to reuse a model for different tasks requiring a different range of angles and if memory is not an issue, you can therefore use a large range when creating the model and a smaller range for the search.

If the object is (almost) symmetric you should limit the allowed range. Otherwise, the search process will find multiple, almost equally good matches on the same object at different angles; which match (at which angle) is returned as the best can therefore "jump" from image to image. The suitable range of rotation depends on the symmetry: For a cross-shaped or square object the allowed extent must be less than 90°, for a rectangular object less than 180°, and for a circular object 0°.

To speed up the matching process, the model is precomputed for different angles within the allowed range, at steps specified with the parameter `AngleStep`. If you select the value `0`, HALCON automatically chooses an optimal step size ϕ_{opt} to obtain the highest possible accuracy by determining the smallest rotation that is still discernible in the image. The underlying algorithm is explained in figure 9: The rotated version of the cross-shaped object is clearly discernible from the original if the point that lies farthest from the center of the object is moved by at least 2 pixels. Therefore, the corresponding angle ϕ_{opt} is calculated as follows:

$$d^2 = l^2 + l^2 - 2 \cdot l \cdot l \cdot \cos\phi \quad \Rightarrow \quad \phi_{opt} = \arccos\left(1 - \frac{d^2}{2 \cdot l^2}\right) = \arccos\left(1 - \frac{2}{l^2}\right)$$

with l being the maximum distance between the center and the object boundary and $d = 2$ pixels.

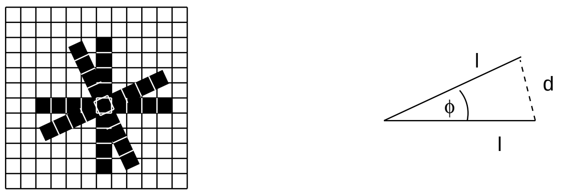


Figure 9: Determining the minimum angle step size from the extent of the model.

The automatically determined angle step size ϕ_{opt} is suitable for most applications; therefore, **we recommend to select the value 0**. You can query the used value after the creation via the operator get_shape_model_params. By selecting a higher value you can speed up the search process, however, at the cost of a decreased accuracy of the estimated orientation. Note that for very high values the matching may fail altogether!

The value chosen for AngleStep should not deviate too much from the optimal value ($\frac{1}{3}\phi_{opt} \leq \phi \leq 3\phi_{opt}$). Note that choosing a very small step size does not result in an increased angle accuracy!

2.2.4 Allowing a Range of Scale

Similarly to the range of orientation, you can specify an allowed range of scale with the parameters ScaleMin, ScaleMax, and ScaleStep of the operator create_scaled_shape_model.

Again, we recommend to limit the allowed range of scale as much as possible in order to speed up the search process and to minimize the required memory. Note that you can further limit the allowed range when calling the operator find_scaled_shape_model (see section 3.1.2).

Note that if you are searching for the object on a large range of scales you should **create the model based on a large scale** because HALCON cannot "guess" model points when precomputing model instances at scales larger than the original one. On the other hand, NumLevels should be chosen such that the highest level contains enough model points also for the smallest scale.

If you select the value 0 for the parameter ScaleStep, HALCON automatically chooses a suitable step size to obtain the highest possible accuracy by determining the smallest scale change that is still discernible in the image. Similarly to the angle step size (see figure 9), a scaled object is clearly discernible from the original if the point that lies farthest from the center of the object is moved by at least 2 pixels. Therefore, the corresponding scale change Δs_{opt} is calculated as follows:

$$\Delta s = \frac{d}{l} \quad \Rightarrow \quad \Delta s_{opt} = \frac{2}{l}$$

with l being the maximum distance between the center and the object boundary and $d = 2$ pixels.

The automatically determined scale step size is suitable for most applications; therefore, **we recommend to select the value 0**. You can query the used value after the creation via the operator get_shape_model_params. By selecting a higher value you can speed up the search process, however, at the cost of a decreased accuracy of the estimated scale. Note that for very high values the matching may fail altogether!

The value chosen for ScaleStep should not deviate too much from the optimal value ($\frac{1}{3}\Delta s_{opt} \leq \Delta s \leq 3\Delta s_{opt}$). Note that choosing a very small step size does not result in an increased scale accuracy!

2.2.5 Which Pixels are Compared with the Model?

For efficiency reasons the model contains information that influences the search process: With the parameter MinContrast you can specify which contrast a point in a search image must at least have in order to be compared with the model. The main use of this parameter is to exclude noise, i.e., gray value fluctuations, from the matching process. You can determine the noise by examining the gray values with the HDevelop dialog `Visualization` ▷ `Pixel Info`; then, set the minimum contrast to a value larger than the noise.

The parameter `Metric` lets you specify whether the *polarity*, i.e., the direction of the contrast must be observed. If you choose the value `'use_polarity'` the polarity is observed, i.e., the points in the search image must show the same direction of the contrast as the corresponding points in the model. If, for example, the model is a bright object on a dark background, the object is found in the search images only if it is also brighter than the background.

You can choose to ignore the polarity globally by selecting the value `'ignore_global_polarity'`. In this mode, an object is recognized also if the direction of its contrast reverses, e.g., if your object can appear both as a dark shape on a light background and vice versa. This flexibility, however, is obtained at the cost of a slightly lower recognition speed.

If you select the value `'ignore_local_polarity'`, the object is found even if the contrast changes locally. This mode can be useful, e.g., if the object consists of a part with a medium gray value, within which either darker of brighter sub-objects lie. Please note however, that the recognition speed may decrease dramatically in this mode, especially if you allowed a large range of rotation (see section 2.2.3).

2.3 Synthetic Model Images

Depending on the application it may be difficult to create a suitable model because there is no “good” model image containing a perfect, easy to extract instance of the object. An example of such a case was already shown in section 2.1.2: The task of locating the capacitors seems to be simple at first, as they are prominent bright circles on a dark background. But because of the clutter inside and outside the circle even the model resulting from the ring-shaped ROI is faulty: Besides containing clutter points also parts of the circle are missing.

In such cases, it may be better to use a *synthetic model image*. How to create such an image to locate the capacitors is explained below. To follow the example actively, start the HDevelop program `hdevelop\synthetic_circle.dev`; we start after the initialization of the application (press $\boxed{\text{F5}}$ once).

Step 1: Create an XLD contour

```
RadiusCircle := 43
SizeSynthImage := 2*RadiusCircle + 10
gen_ellipse_contour_xld (Circle, SizeSynthImage / 2, SizeSynthImage / 2, 0,
                         RadiusCircle, RadiusCircle, 0, 6.28318,
                         'positive', 1.5)
```

First, we create a circular region using the operator `gen_ellipse_contour_xld` (see figure 10a). You can determine a suitable radius by inspecting the image with the HDevelop dialog `Visualization` ▷ `Online Zooming`. Note that the synthetic image should be larger than the region because pixels around the region are used when creating the image pyramid.

Step 2: Create an image and insert the XLD contour

```
gen_image_const (EmptyImage, 'byte', SizeSynthImage, SizeSynthImage)
paint_xld (Circle, EmptyImage, SyntheticModelImage, 128)
```

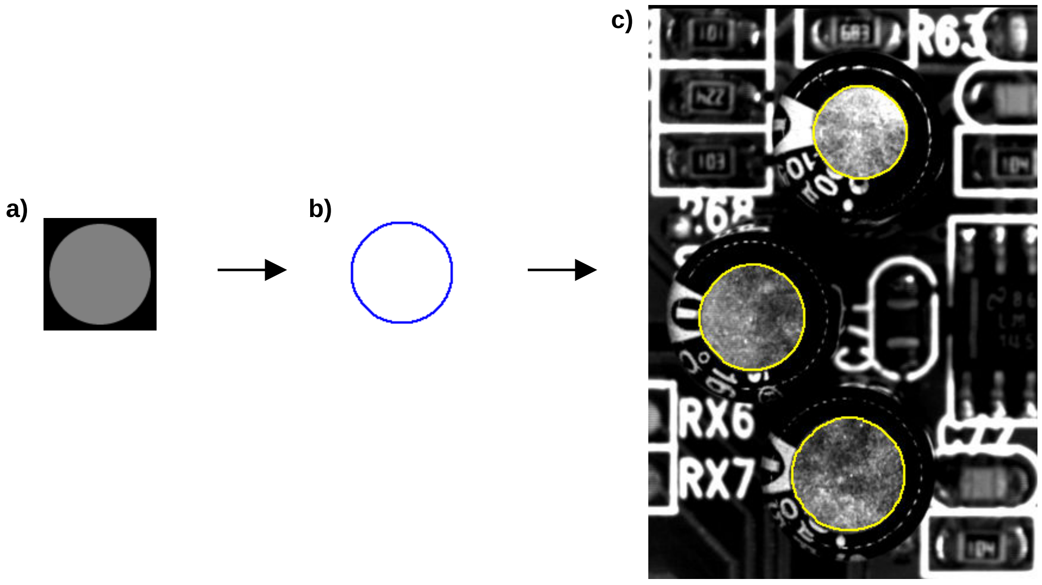


Figure 10: Locating the capacitors using a synthetic model: a) paint region into synthetic image; b) corresponding model; c) result of the search.

Then, we create an empty image using the operator gen_image_const and insert the XLD contour with the operator paint_xld. In figure 10a the resulting image is depicted.

Step 3: Create the model

```
create_scaled_shape_model (SyntheticModelImage, 0, 0, 0, 0.01, 0.8, 1.2, 0,
                           'none', 'use_polarity', 30, 10, ModelID)
```

Now, the model is created from the synthetic image. Figure 10d shows the corresponding model region, figure 10e the search results.

Note how the image itself, i.e., its domain, acts as the ROI in this example.

3 Optimizing the Search Process

The actual matching is performed by the operators find_shape_model, find_scaled_shape_model, find_shape_models, or find_scaled_shape_models. In the following, we show how to select suitable parameters for these operators to adapt and optimize it for your matching task.

3.1 Restricting the Search Space

An important concept in the context of finding objects is that of the so-called *search space*. Quite literally, this term specifies where to search for the object. However, this space encompasses not only the 2 dimensions of the image, but also other parameters like the possible range of scales and orientations or the question of how much of the object must be visible. The more you can restrict the search space, the faster the search will be.

3.1.1 Searching in a Region of Interest

The obvious way to restrict the search space is to apply the operator find_shape_model to a region of interest only instead of the whole image as shown in figure 11. This can be realized in a few lines of code:

Step 1: Create a region of interest

```
Row1 := 141
Column1 := 163
Row2 := 360
Column2 := 477
gen_rectangle1 (SearchROI, Row1, Column1, Row2, Column2)
```

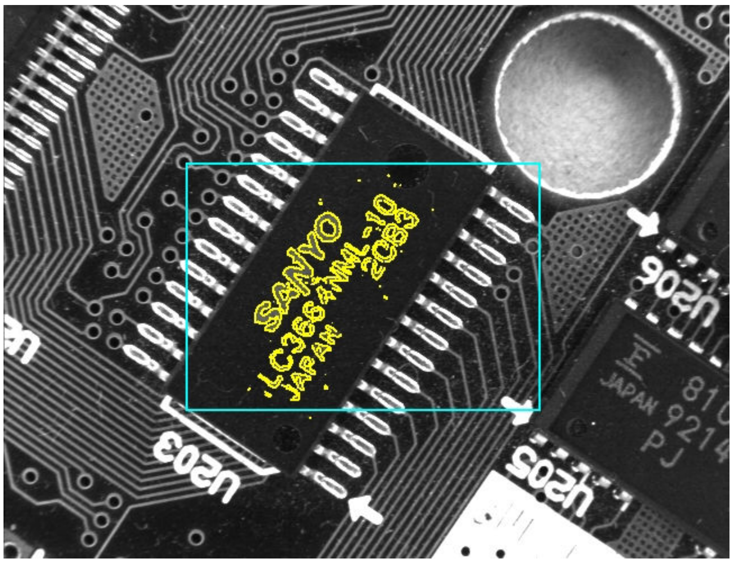


Figure 11: Searching in a region of interest.

First, you create a region, e.g., with the operator gen_rectangle1 (see section 2.1.1 for more ways to create regions).

Step 2: Restrict the search to the region of interest

```
for i := 1 to 20 by 1
  grab_image (SearchImage, FGHandle)
  reduce_domain (SearchImage, SearchROI, SearchImageROI)
  find_shape_model (SearchImageROI, ModelID, 0, rad(360), 0.8, 1, 0.5,
                    'interpolation', 0, 0.9, RowCheck, ColumnCheck,
                    AngleCheck, Score)
endfor
```

The region of interest is then applied to each search image using the operator reduce_domain. In this example, the searching speed is almost doubled using this method.

Note that by restricting the search to a region of interest you actually restrict the position of the *point of reference* of the model, i.e., the center of gravity of the model ROI (see section 2.1.4). This means that the size of the search ROI corresponds to the extent of the allowed movement; for example, if your object can move $\pm$ 10 pixels vertically and $\pm$ 15 pixels horizontally you can restrict the search to an ROI of the size 20×30. In order to assure a correct boundary treatment on higher pyramid levels, we recommend to enlarge the ROI by $2^{NumLevels-1}$ pixels; to continue the example, if you specified NumLevels = 4, you can restrict the search to an ROI of the size 36×46.

Please note that even if you modify the point of reference using set_shape_model_origin, the original one, i.e., the center point of the model ROI, is used during the search. Thus, you must always specify the search ROI relative to the original reference point.

3.1.2 Restricting the Range of Orientation and Scale

When creating the model with the operator create_shape_model (or create_scaled_shape_model), you already specified the allowed range of orientation and scale section 2.2.3 and section 2.2.4. When calling the operator find_shape_model (or find_scaled_shape_model) you can futher limit these ranges with the parameters AngleStart, AngleExtent, ScaleMin, and ScaleMax. This is useful if you can restrict these ranges by other information, which can, e.g., be obtained by suitable image processing operations.

Another reason for using a larger range when creating the model may be that you want to reuse the model for other matching tasks.

3.1.3 Visibility

With the parameter MinScore you can specify how much of the object — more precisely: of the model — must be visible. A typical use of this mechanism is to allow a certain degree of occlusion as demonstrated in figure 12: The security ring is found if MinScore is set to 0.7.

Let's take a closer look at the term "visibility": When comparing a part of a search image with the model, the matching process calculates the so-called *score*, which is a measure of how many model points could be matched to points in the search image (ranging from 0 to 1). A model point may be "invisible" and thus not matched because of multiple reasons:

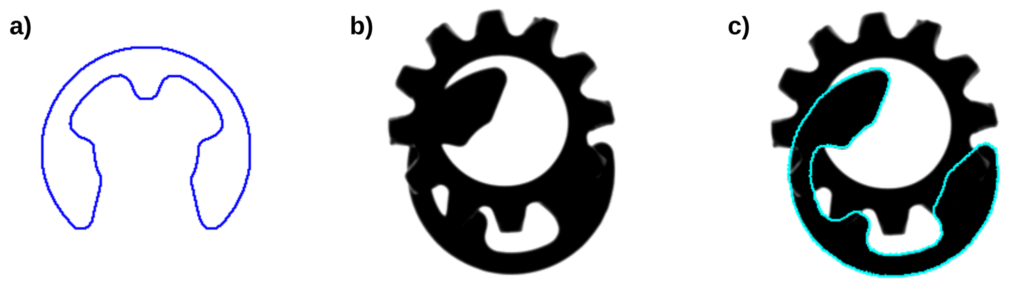


Figure 12: Searching for partly occluded objects: a) model of the security ring; b) search result for MinScore = 0.8; c) search result for MinScore = 0.7.

- Parts of the object's contour are occluded, e.g., as in figure 12.

  ! Please note that **an object must not be clipped at the image border**; this case is not treated as an occlusion! More precisely, the smallest rectangle surrounding the model must not be clipped.

- Parts of the contour have a contrast lower than specified in the parameter MinContrast when creating the model (see section 2.2.5).

- The polarity of the contrast changes globally or locally (see section 2.2.5).

- If the object is deformed, parts of the contour may be visible but appear at an incorrect position and therefore do not fit the model anymore. Note that this effect also occurs if camera observes the scene under an oblique angle; section 5.1 shows how to handle this case.

Besides these obvious reasons, which have their root in the search image, there are some not so obvious reasons caused by the matching process itself:

- As described in section 2.2.3, HALCON precomputes the model for intermediate angles within the allowed range of orientation. During the search, a candidate match is then compared to all precomputed model instances. If you select a value for the parameter AngleStep that is significantly larger than the automatically selected minimum value, the effect depicted in figure 13 can occur: If the object lies between two precomputed angles, points lying far from the center are not matched to a model point, and therefore the score decreases.

  Of course, the same line of reasoning applies to the parameter ScaleStep (see section 2.2.4).

- Another stumbling block lies in the use of an image pyramid which was introduced in section 2.2.2: When comparing a candidate match with the model, the specified minimum score must be reached on each pyramid level. However, on different levels the score may vary, with only the score on the lowest level being returned in the parameter Score; this sometimes leads to the apparently paradox situation that MinScore must be set significantly lower than the resulting Score.

☞ **Recommendation:** The higher MinScore, the faster the search!

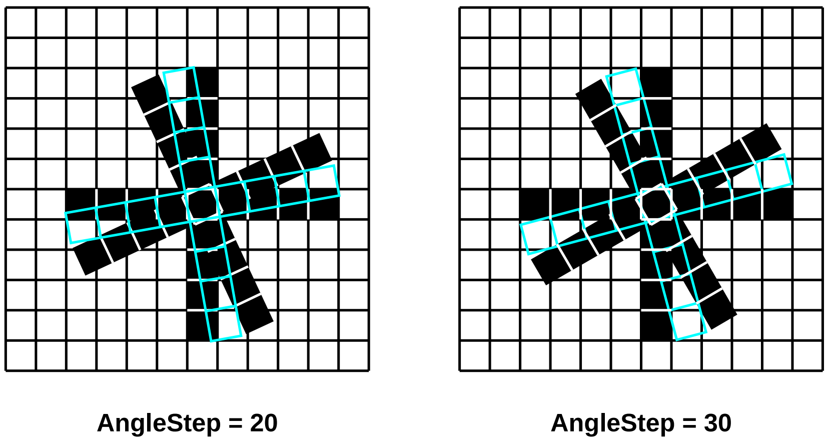


Figure 13: The effect of a large `AngleStep` on the matching.

3.1.4 Thoroughness vs. Speed

With the parameter `Greediness` you can influence the search algorithm itself and thereby trade thoroughness against speed. If you select the value `0`, the search is thorough, i.e., if the object is present (and within the allowed search space and reaching the minimum score), it will be found. In this mode, however, even very unlikely match candidates are also examined thoroughly, thereby slowing down the matching process considerably.

The main idea behind the “greedy” search algorithm is to break off the comparison of a candidate with the model when it seems unlikely that the minimum score will be reached. In other words, the goal is not to waste time on hopeless candidates. This greediness, however, can have unwelcome consequences: In some cases a perfectly visible object is not found because the comparison “starts out on a wrong foot” and is therefore classified as a hopeless candidate and broken off.

You can adjust the `Greediness` of the search, i.e., how early the comparison is broken off, by selecting values between `0` (no break off: thorough but slow) and `1` (earliest break off: fast but unsafe). Note that the parameters `Greediness` and `MinScore` interact, i.e., you may have to specify a lower minimum score in order to use a greedier search. Generally, you can reach a higher speed with a high greediness and a sufficiently lowered minimum score.

3.2 Searching for Multiple Instances of the Object

All you have to do to search for more than one instance of the object is to set the parameter `NumMatches` accordingly. The operator `find_shape_model` (or `find_scaled_shape_model`) then returns the matching results as tuples in the parameters `Row`, `Column`, `Angle`, `Scale`, and `Score`. If you select the value `0`, all matches are returned.

Note that a search for multiple objects is only slightly slower than a search for a single object.

A second parameter, `MaxOverlap`, lets you specify how much two matches may overlap (as a fraction). In figure 14b, e.g., the two security rings overlap by a factor of approximately 0.2. In order to speed up the matching as far as possible, however, the overlap is calculated not for the models themselves but for their smallest surrounding rectangle. This must be kept in mind when specifying the maximum overlap; in most cases, therefore a larger value is needed (e.g., compare figure 14b and figure 14d).

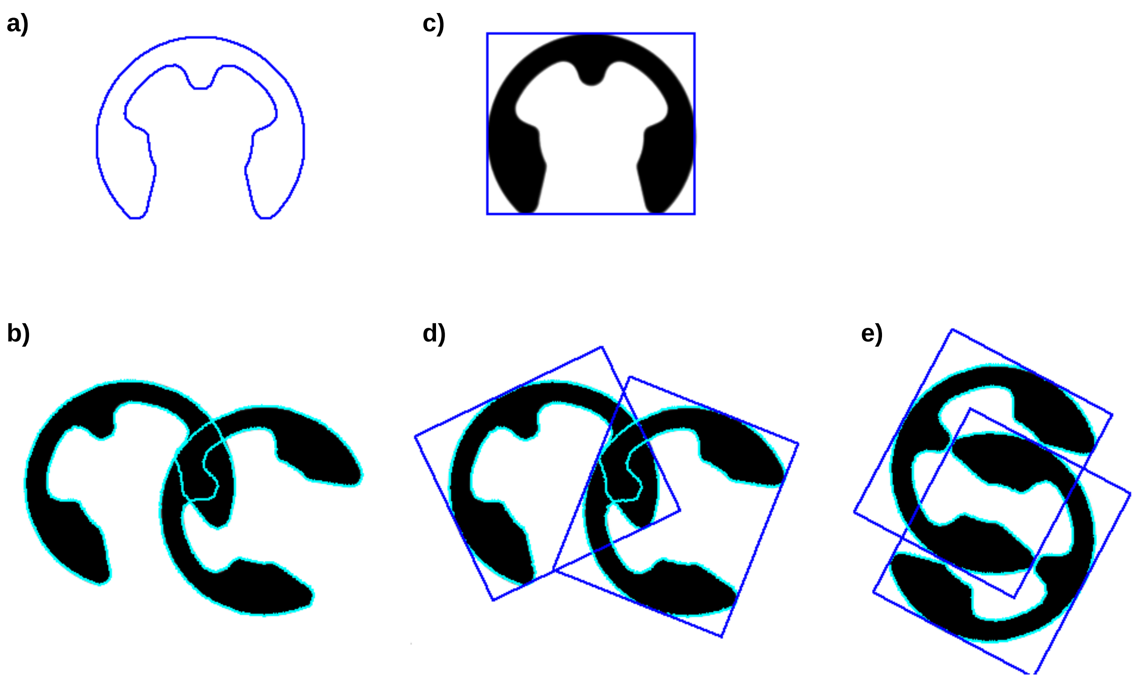


Figure 14: A closer look at overlapping matches: a) model of the security ring; b) model overlap; c) smallest rectangle surrounding the model; d) rectangle overlap; e) pathological case.

Figure 14e shows a “pathological” case: Even though the rings themselves do not overlap, their surrounding rectangles do to a large degree. Unfortunately, this effect cannot be prevented.

3.3 Searching for Multiple Models Simultaneously

If you are searching for instances of multiple models in a single image, you can of course call the operator `find_shape_model` (or `find_scaled_shape_model`) multiple times. A much faster alternative is to use the operators `find_shape_models` or `find_scaled_shape_models` instead. These operators expect similar parameters, with the following differences:

- With the parameter `ModelIDs` you can specify a *tuple* of model IDs instead of a single one. As when searching for multiple instances (see section 3.2), the matching result parameters `Row` etc. return tuples of values.
- The output parameter `Model` shows to which model each found instance belongs. Note that the parameter does not return the model IDs themselves but the index of the model ID in the tuple `ModelIDs` (starting with 0).
- The search is always performed in a single image. However, you can restrict the search to a certain region for each model individually by passing an image tuple (see below for an example).
- You can either use the same search parameters for each model by specifying single values for `AngleStart` etc., or pass a tuple containing individual values for each model.
- You can also search for multiple instances of multiple models. If you search for a certain number of objects independent of their type (model ID), specify this (single) value in the

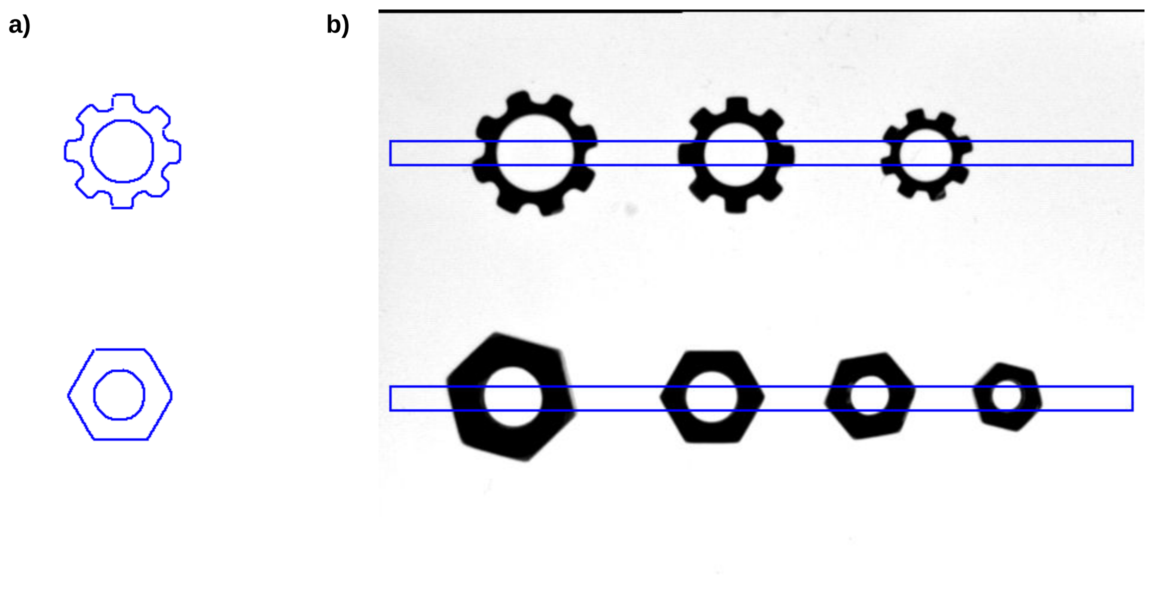


Figure 15: Searching for multiple models : a) models of ring and nut; b) search ROIs for the two models.

parameter NumMatches. By passing a tuple of values, you can specify for each model individually how many instances are to be found. In this tuple, you can mix concrete values with the value 0; the tuple [3,0], e.g., specifies to return the best 3 instances of the first model and all instances of the second model.

Similarly, if you specify a single value for MaxOverlap, the operators check whether a found instance is overlapped by any of the other instances independent of their type. By specifying a tuple of values, each instance is only checked against all other instances of the same type.

The example HDevelop program hdevelop\multiple_models.dev uses the operator find_scaled_shape_models to search simultaneously for the rings and nuts depicted in figure 15.

Step 1: Create the models

```
create_scaled_shape_model (ImageROIRing, 0, -rad(22.5), rad(45), 0, 0.8,
                           1.2, 0, 'none', 'use_polarity', 60, 10,
                           ModelIDRing)
create_scaled_shape_model (ImageROINut, 0, -rad(30), rad(60), 0, 0.6, 1.4,
                           0, 'none', 'use_polarity', 60, 10, ModelIDNut)
ModelIDs := [ModelIDRing, ModelIDNut]
```

First, two models are created, one for the rings and one for the nuts. The two model IDs are then concatenated into a tuple using the operator assign.

Step 2: Specify individual search ROIs

```
gen_rectangle1 (SearchROIRing, 110, 10, 130, Width - 10)
gen_rectangle1 (SearchROINut, 315, 10, 335, Width - 10)
SearchROIs := [SearchROIRing,SearchROINut]
add_channels (SearchROIs, SearchImage, SearchImageReduced)
```

In the example, the rings and nuts appear in non-overlapping parts of the search image; therefore, it is possible to restrict the search space for each model individually. As explained in section 3.1.1, a search ROI corresponds to the extent of the allowed movement; thus, narrow horizontal ROIs can be used in the example (see figure 15b).

The two ROIs are concatenated into a region array (tuple) using the operator `concat_obj` and then "added" to the search image using the operator `add_channels`. The result of this operator is an array of two images, both having the same image matrix; the domain of the first image is restricted to the first ROI, the domain of the second image to the second ROI.

Step 3: Find all instances of the two models

```
find_scaled_shape_models (SearchImageReduced, ModelIDs, [-rad(22.5),
                          -rad(30)], [rad(45), rad(60)], [0.8, 0.6], [1.2,
                          1.4], 0.8, 0, 0, 'interpolation', 0, 0.9,
                          RowCheck, ColumnCheck, AngleCheck, ScaleCheck,
                          Score, ModelIndex)
```

Now, the operator `find_scaled_shape_models` is applied to the created image array. Because the two models allow different ranges of rotation and scaling, tuples are specified for the corresponding parameters. In contrast, the other parameters are are valid for both models. Section 4.3.3 shows how to access the matching results.

3.4 A Closer Look at the Accuracy

During the matching process, candidate matches are compared with instances of the model at different positions, angles, and scales; for each instance, the resulting matching score is calculated. If you set the parameter `SubPixel` to `'none'`, the result parameters `Row`, `Column`, `Angle`, and `Scale` contain the corresponding values of the best match. In this case, the accuracy of the position is therefore 1 pixel, while the accuracy of the orientation and scale is equal to the values selected for the parameters `AngleStep` and `ScaleStep`, respectively, when creating the model (see section 2.2.3 and section 2.2.4).

If you set the parameter `SubPixel` to `'interpolation'`, HALCON examines the matching scores at the neighboring positions, angles, and scales around the best match and determines the maximum by interpolation. Using this method, the position is therefore estimated with subpixel accuracy ($\approx \frac{1}{20}$ pixel in typical applications). The accuracy of the estimated orientation and scale depends on the size of the object, like the optimal values for the parameters `AngleStep` and `ScaleStep` (see section 2.2.3 and section 2.2.4): The larger the size, the more accurately the orientation and scale can be determined. For example, if the maximum distance between the center and the boundary is 100 pixel, the orientation is typically determined with an accuracy of $\approx \frac{1}{10}^{\circ}$.

Recommendation: Because the interpolation is very fast, you can set `SubPixel` to `'interpolation'` in most applications.

When you choose the values `'least_squares'`, `'least_squares_high'`, or `'least_squares_very_high'`, a least-squares adjustment is used instead of an interpolation, resulting in a higher accuracy. However, this method requires additional computation time.

Please note that the **accuracy of the estimated position may decrease if you modify the**

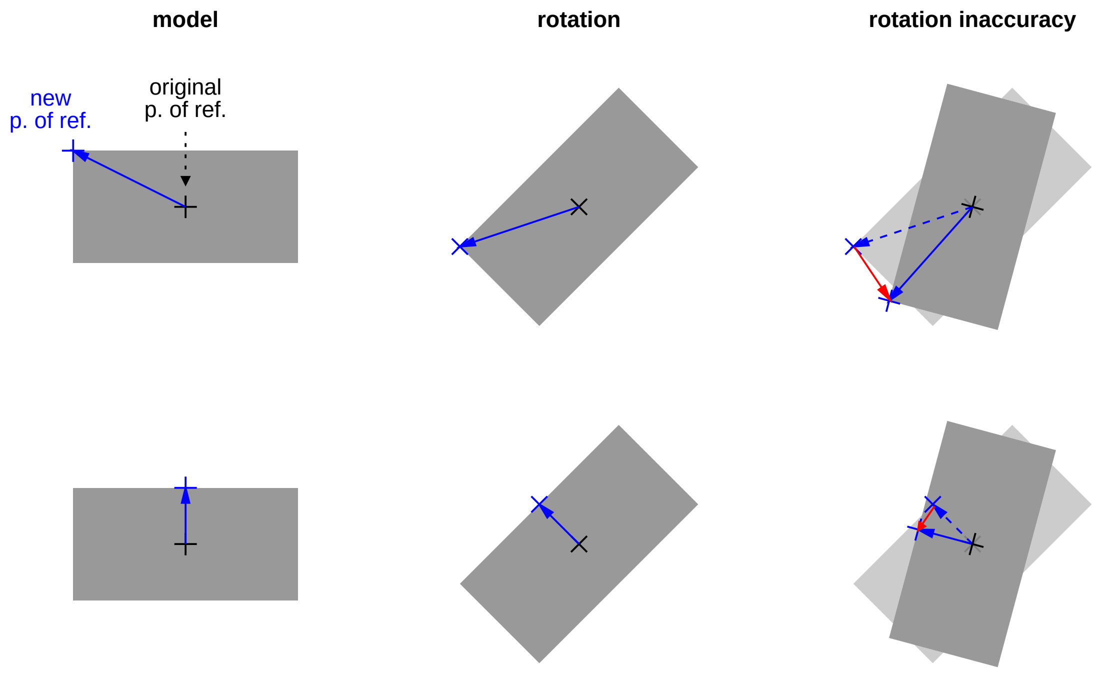


Figure 16: Effect of inaccuracy of the estimated orientation on a moved point of reference.

point of reference using `set_shape_model_origin`! This effect is visualized in figure 16: As you can see in the right-most column, an inaccuracy in the estimated orientation “moves” the modified point of reference, while the original point of reference is not affected. The resulting positional error depends on multiple factors, e.g., the offset of the reference point and the orientation of the found object. The main point to keep in mind is that the error increases linearly with the *distance* of the modified point of reference from the original one (compare the two rows in figure 16).

An inaccuracy in the estimated scale also results in an error in the estimated position, which again increases linearly with the distance between the modified and the original reference point.

For maximum accuracy in case the reference point is moved, the position should be determined using the least-squares adjustment. Note that the accuracy of the estimated orientation and scale is not influenced by modifying the reference point.

3.5 How to Optimize the Matching Speed

! In the following, we show how to optimize the matching process in two steps. Please note that in order to optimize the matching it is very important to have a **set of representative *test images* from your application** in which the object appears in all allowed variations regarding its position, orientation, occlusion, and illumination.

Step 1: Assure that all objects are found

Before tuning the parameters for speed, we recommend to find settings such that the matching succeeds in all test images, i.e., that all object instances are found. If this is not the case when using the default values, check whether one of the following situations applies:

? **Is the object clipped at the image border?**

Unfortunately, this failure cannot be prevented, i.e., you must assure that the object is not clipped (see section 3.1.3).

? **Is the search algorithm "too greedy"?**

As described in section 3.1.4, in some cases a perfectly visible object is not found if the `Greediness` is too high. Select the value `0` to force a thorough search.

? **Is the object partly occluded?**

If the object should be recognized in this state nevertheless, reduce the parameter `MinScore`.

? **Does the matching fail on the highest pyramid level?**

As described in section 3.1.3, in some cases the minimum score is not reached on the highest pyramid level even though the score on the lowest level is much higher. Test this by reducing `NumLevels` in the call to `find_shape_model`. Alternatively, reduce the `MinScore`.

? **Does the object have a low contrast?**

If the object should be recognized in this state nevertheless, reduce the parameter `MinContrast` (operator `create_shape_model`!).

? **Is the polarity of the contrast inverted globally or locally?**

If the object should be recognized in this state nevertheless, use the appropriate value for the parameter `Metric` when creating the model (see section 2.2.5). If only a small part of the object is affected, it may be better to reduce the `MinScore` instead.

? **Does the object overlap another instance of the object?**

If the object should be recognized in this state nevertheless, increase the parameter `MaxOverlap` (see section 3.2).

? **Are multiple matches found on the same object?**

If the object is almost symmetric, restrict the allowed range of rotation as described in section 2.2.3 or decrease the parameter `MaxOverlap` (see section 3.2).

Step 2: Tune the parameters regarding speed

The speed of the matching process depends both on the model and on the search parameters. To make matters more difficult, the search parameters depend on the chosen model parameters. We recommend the following procedure:

- Increase the `MinScore` as far as possible, i.e., as long as the matching succeeds.
- Now, increase the `Greediness` until the matching fails. Try reducing the `MinScore`; if this does not help restore the previous values.
- If possible, use a larger value for `NumLevels` when creating the model.
- Restrict the allowed range of rotation and scale as far as possible as described in section 2.2.3 and section 2.2.4. Alternatively, adjust the corresponding parameters when calling `find_shape_model` or `find_scaled_shape_model`.
- Restrict the search to a region of interest as described in section 3.1.1.

The following methods are more "risky", i.e., the matching may fail if you choose unsuitable parameter values.

- Increase the `MinContrast` as long as the matching succeeds.
- If you a searching for a particularly large object, it sometimes helps to select a higher point reduction with the parameter `Optimization` (see section 2.2.2).
- Increase the `AngleStep` (and the `ScaleStep`) as long as the matching succeeds.

4 Using the Results of Matching

As results, the operators find_shape_model, find_scaled_shape_model etc. return

- the position of the match in the parameters Row and Column,
- its orientation in the parameter Angle,
- the scaling factor in the parameter Scale, and
- the matching score in the parameter Score.

The matching score, which is a measure of the similarity between the model and the matched object, can be used "as it is", since it is an absolute value.

In contrast, the results regarding the position, orientation, and scale are worth a closer look as they are determined relative to the created model. Before this, we introduce HALCON's powerful operators for the so-called *affine transformations*, which, when used together with the shape-based matching, enable you to easily realize applications like image rectification or the alignment of ROIs with a few lines of code.

4.1 Introducing Affine Transformations

"Affine transformation" is a technical term in mathematics describing a certain group of transformations. Figure 17 shows the types that occur in the context of the shape-based matching: An object can be *translated* (moved) along the two axes, *rotated*, and *scaled*. In figure 17d, all three transformations were applied in a sequence.

Note that for the rotation and the scaling there exists a special point, called *fixed point* or *point of reference*. The transformation is performed around this point. In figure 17b, e.g., the IC is rotated around its center, in figure 17e around its upper right corner. The point is called fixed point because it remains unchanged by the transformation.

The transformation can be thought of as a mathematical instruction that defines how to calculate the coordinates of object points after the transformation. Fortunately, you need not worry about the mathematical part; HALCON provides a set of operators that let you specify and apply tranformations in a simple way.

4.2 Creating and Applying Affine Transformations With HALCON

HALCON allows to transform not only regions, but also images and XLD contours by providing the operators affine_trans_region, affine_trans_image, and affine_trans_contour_xld. The transformation in figure 17d corresponds to the line

```
affine_trans_region (IC, TransformedIC, ScalingRotationTranslation,
                     'false')
```

The parameter ScalingRotationTranslation is a so-called *homogeneous transformation matrix* that describes the desired transformation. You can create this matrix by adding simple transformations step by step. First, an identity matrix is created:

```
hom_mat2d_identity (EmptyTransformation)
```

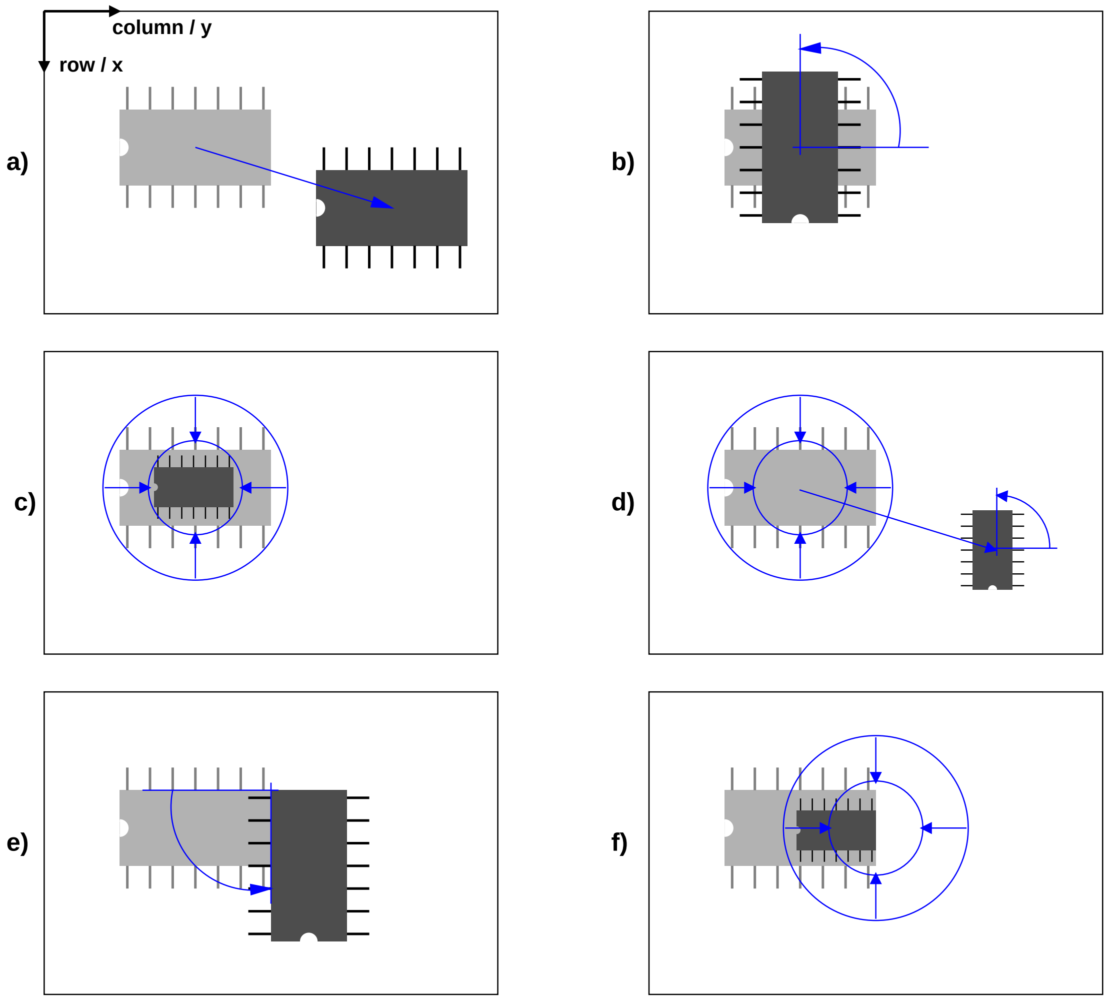


Figure 17: Typical affine transformations: a) translation along two axes; b) rotation around the IC center; c) scaling around the IC center; d) combining a, b, and c; e) rotation around the upper right corner; f) scaling around the right IC center.

Then, the scaling around the center of the IC is added:

```
hom_mat2d_scale (EmptyTransformation, 0.5, 0.5, RowCenterIC,
                 ColumnCenterIC, Scaling)
```

Similarly, the rotation and the translation are added:

```
hom_mat2d_rotate (Scaling, rad(90), RowCenterIC, ColumnCenterIC,
                  ScalingRotation)
hom_mat2d_translate (ScalingRotation, 100, 200, ScalingRotationTranslation)
```

Please note that in these operators the coordinate axes are labeled with `x` and `y` instead of `Row` and `Column`! Figure 17a clarifies the relation.

Tranformation matrices can also be constructed by a sort of “reverse engineering”. In other words, if the result of the transformation is known for some points of the object, you can determine the corresponding transformation matrix. If, e.g., the position of the IC center and its orientation after the transformation is known, you can get the corresponding matrix via the operator `vector_angle_to_rigid`.

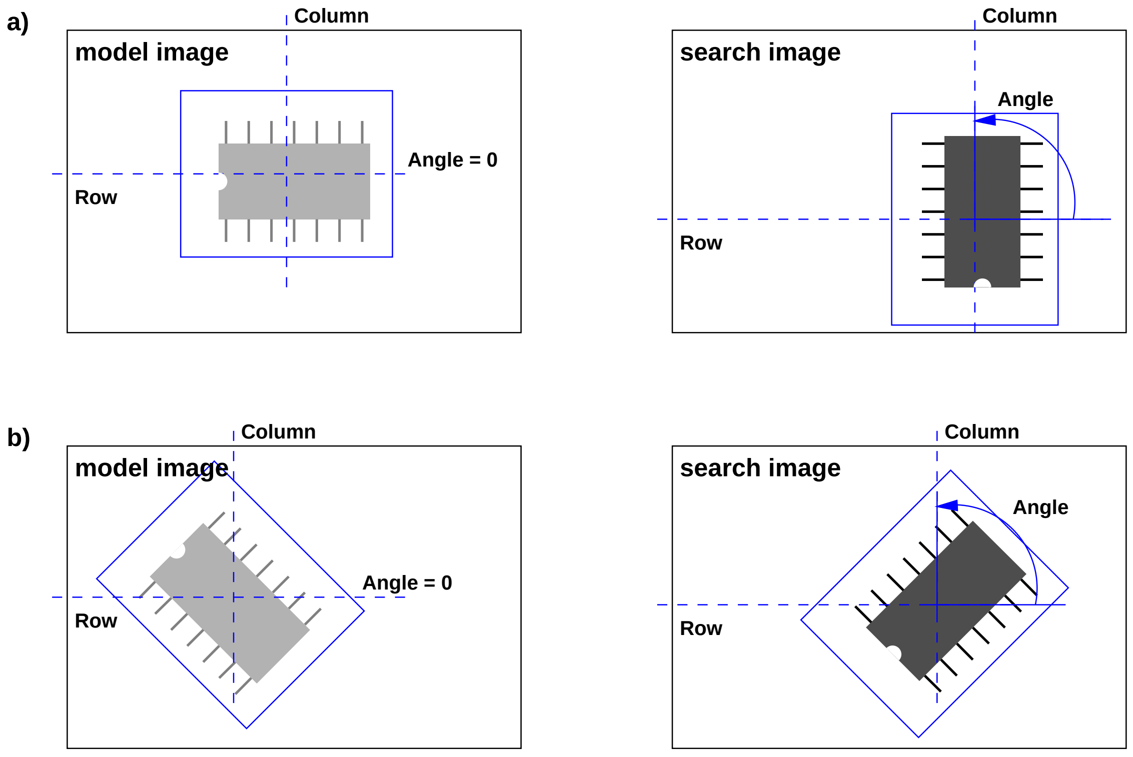


Figure 18: The position and orientation of a match: a) The center of the ROI acts as the default point of reference; b) In the model image, the orientation is always 0.

```
vector_angle_to_rigid (RowCenterIC, ColumnCenterIC, 0,
                       TransformedRowCenterIC, TransformedColumnCenterIC,
                       rad(90), RotationTranslation)
```

and then use this matrix to compute the transformed region:

```
affine_trans_region (IC, TransformedIC, RotationTranslation, 'false')
```

4.3 Using the Estimated Position and Orientation

! There are two things to keep in mind about the position and orientation returned in the parameters `Row`, `Column`, and `Angle`: First, by default **the center of the ROI acts as the point of reference** for both transformations, i.e., the rotation is performed around this point, and the returned position denotes the position of the ROI center in the search image. This is depicted in figure 18a with the example of an ROI whose center does not coincide with the center of the IC.

Secondly, in the model image the object is taken as not rotated, i.e., its angle is `0`, even if it seems to be rotated, e.g., as in figure 18b.

After creating a model, you can change its point of reference with the operator `set_shape_model_origin`. Note that this operator expects not the absolute position of the new reference point as parameters, but its *distance* to the default reference point! An example can be found section 4.3.4; please note that by modifying the point of reference, the accuracy of the estimated position may decrease (see section 3.4).

4.3.1 Displaying the Matches

Especially during the development of a matching application it is useful to display the matching results overlaid on the search image. This can be realized in a few steps (see, e.g., the HDevelop program `hdevelop\first_example_shape_matching.dev`):

Step 1: Determine the point of reference

```
gen_rectangle1 (ROI, Row1, Column1, Row2, Column2)
area_center (ROI, Area, CenterROIRow, CenterROIColumn)
```

You can determine the center of the ROI, i.e., the point of reference, with the operator `area_center`.

Step 2: Create an XLD contour containing the model

```
inspect_shape_model (ImageROI, ShapeModelImages, ShapeModelRegions, 8, 30)
ShapeModelRegion := ShapeModelRegions[1]
gen_contours_skeleton_xld (ShapeModelRegion, ShapeModel, 1, 'filter')
```

Below, we want to display the model at the extracted position and orientation. The corresponding region can be accessed via the operator `inspect_shape_model`. However, if you call the operator with `NumLevels` > 1 as in the example, an array (tuple) of regions is returned, with the desired region at the first position; you can select the region from the array via the operator `select_obj`. We recommend to transform this region into an XLD contour using the operator `gen_contours_skeleton_xld` because XLD contours can be transformed more precisely and quickly.

Step 3: Determine the affine transformation

```
  find_shape_model (SearchImage, ModelID, 0, rad(360), 0.8, 1, 0.5,
                    'interpolation', 0, 0.9, RowCheck, ColumnCheck,
                    AngleCheck, Score)
  if (|Score| = 1)
    vector_angle_to_rigid (CenterROIRow, CenterROIColumn, 0, RowCheck,
                           ColumnCheck, AngleCheck, MovementOfObject)
```

After the call of the operator `find_shape_model`, the results are checked; if the matching failed, empty tuples are returned in the parameters `Score` etc. For a successful match, the corresponding affine transformation can be constructed with the operator `vector_angle_to_rigid` from the movement of the center of the ROI (see section 4.2).

Step 4: Transform the XLD

```
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
                              MovementOfObject)
    dev_display (ModelAtNewPosition)
```

Now, you can apply the tranformation to the XLD version of the model using the operator `affine_trans_contour_xld` and display it; figure 2 shows the result.

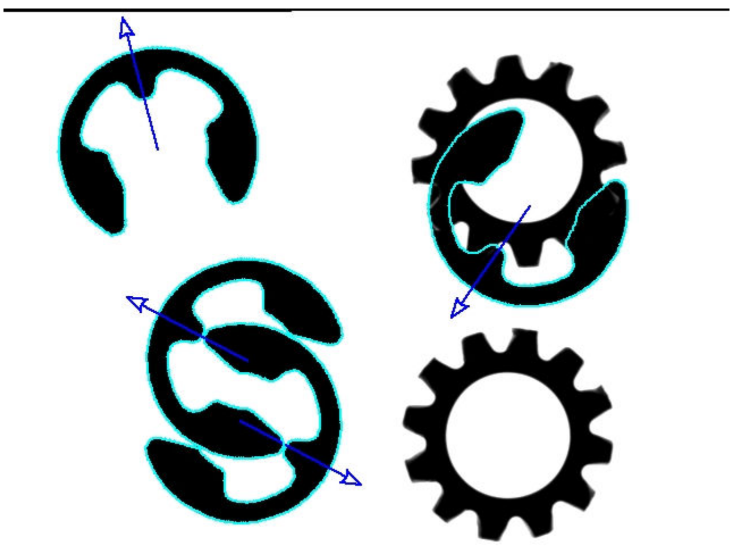


Figure 19: Displaying multiple matches; the used model is depicted in figure 12a.

4.3.2 Dealing with Multiple Matches

If multiple instances of the object are searched and found, the parameters Row, Column, Angle, and Score contain *tuples*. The HDevelop program hdevelop\multiple_objects.dev shows how to access these results in a loop:

Step 1: Determine the affine transformation

```
find_shape_model (SearchImage, ModelID, 0, rad(360), 0.75, 0, 0.55,
                  'interpolation', 0, 0.8, RowCheck, ColumnCheck,
                  AngleCheck, Score)
for j := 0 to |Score| - 1 by 1
  vector_angle_to_rigid (CenterROIRow, CenterROIColumn, 0,
                         RowCheck[j], ColumnCheck[j], AngleCheck[j],
                         MovementOfObject)
```

The transformation corresponding to the movement of the match is determined as in the previous section; the only difference is that the position of the point of reference is extracted from the tuple via the loop variable.

Step 2: Use the transformation

```
affine_trans_point_2d (MovementOfObject, CenterROIRow - 120 + 0.5,
                       CenterROIColumn + 0.5, RowArrowHead,
                       ColumnArrowHead)
disp_arrow (WindowHandle, RowCheck[j], ColumnCheck[j],
            RowArrowHead - 0.5, ColumnArrowHead - 0.5, 2)
```

In this example, the transformation is also used to display an arrow that visualizes the orientation (see figure 19).

! Note that the operator affine_trans_point_2d **and the HALCON regions (and XLDs) use different definitions of the position of a pixel**: For a region, a pixel is positioned at its middle, for affine_trans_point_2d at its upper left corner. Therefore, 0.5 must be added to the pixel coordinates before transforming them and subtracted again before creating the regions.

4.3.3 Dealing with Multiple Models

When searching for multiple models simultaneously as described in section 3.3, it is useful to store the information about the models, i.e., the reference point and the model region or XLD contour, in tuples. The following example code stems from the already partly described HDevelop program `hdevelop\multiple_models.dev`, which uses the operator `find_scaled_shape_models` to search simultaneously for the rings and nuts depicted in figure 15.

Step 1: Inspect the models

```
inspect_shape_model (ImageROIRing, PyramidImage, ModelRegionRing, 1, 30)
gen_contours_skeleton_xld (ModelRegionRing, ShapeModelRing, 1, 'filter')
area_center (ModelROIRing, Area, CenterROIRowRing, CenterROIColumnRing)
inspect_shape_model (ImageROINut, PyramidImage, ModelRegionNut, 1, 30)
gen_contours_skeleton_xld (ModelRegionNut, ShapeModelNut, 1, 'filter')
area_center (ModelROINut, Area, CenterROIRowNut, CenterROIColumnNut)
```

As in the previous sections, the XLD contours corresponding to the two models are created with the operators `inspect_shape_model` and `gen_contours_skeleton_xld`, the reference points are determined using `area_center`.

Step 2: Save the information about the models in tuples

```
NumContoursRing := |ShapeModelRing|
NumContoursNut := |ShapeModelNut|
ShapeModels := [ShapeModelRing,ShapeModelNut]
StartContoursInTuple := [1, NumContoursRing+1]
NumContoursInTuple := [NumContoursRing, NumContoursNut]
CenterROIRows := [CenterROIRowRing, CenterROIRowNut]
CenterROIColumns := [CenterROIColumnRing, CenterROIColumnNut]
```

To facilitate the access to the shape models later, the XLD contours and the reference points are saved in tuples in analogy to the model IDs (see section 3.3). However, when concatenating XLD contours with the operator `concat_obj`, one must keep in mind that XLD objects are already tuples as they may consist of multiple contours! To access the contours belonging to a certain model, you therefore need the number of contours of a model and the starting index in the concatenated tuple. The former is determined using the operator `count_obj`; the contours of the ring start with the index 1, the contours of the nut with the index 1 plus the number of contours of the ring.

Step 3: Access the found instances

```
find_scaled_shape_models (SearchImageReduced, ModelIDs, [-rad(22.5),
                          -rad(30)], [rad(45), rad(60)], [0.8, 0.6], [1.2,
                          1.4], 0.8, 0, 0, 'interpolation', 0, 0.9,
                          RowCheck, ColumnCheck, AngleCheck, ScaleCheck,
                          Score, ModelIndex)
for i := 0 to |Score| - 1 by 1
  Model := ModelIndex[i]
  vector_angle_to_rigid (CenterROIRows[Model], CenterROIColumns[Model],
                         0, RowCheck[i], ColumnCheck[i], AngleCheck[i],
                         MovementOfObject)
  hom_mat2d_scale (MovementOfObject, ScaleCheck[i], ScaleCheck[i],
                   RowCheck[i], ColumnCheck[i], MoveAndScalingOfObject)
  copy_obj (ShapeModels, ShapeModel, StartContoursInTuple[Model],
            NumContoursInTuple[Model])
  affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
                            MoveAndScalingOfObject)
  dev_display (ModelAtNewPosition)
endfor
```

As already described in section 4.3.2, in case of multiple matches the output parameters `Row` etc. contain tuples of values, which are typically accessed in a loop, using the loop variable as the index into the tuples. When searching for multiple models, a second index is involved: The output parameter `Model` indicates to which model a match belongs by storing the index of the corresponding model ID in the tuple of IDs specified in the parameter `ModelIDs`. This may sound confusing, but can be realized in an elegant way in the code: For each found instance, the model ID index is used to select the corresponding information from the tuples created above.

As already noted, the XLD representing the model can consist of multiple contours; therefore, you cannot access them directly using the operator `select_obj`. Instead, the contours belonging to the model are selected via the operator `copy_obj`, specifying the start index of the model in the concatenated tuple and the number of contours as parameters. Note that `copy_obj` does not copy the contours, but only the corresponding HALCON objects, which can be thought of as references to the contours.

4.3.4 Aligning Other ROIs

The results of the matching can be used to *align* ROIs for other image processing steps. i.e., to position them relative to the image part acting as the model. This method is very useful, e.g., if the object to be inspected is allowed to move or if multiple instances of the object are to be inspected at once as in the example application described below.

In the example application `hdevelop\align_measurements.dev` the task is to inspect razor blades by measuring the width and the distance of their “teeth”. Figure 20a shows the model ROI, figure 20b the corresponding model region.

The inspection task is realized with the following steps:

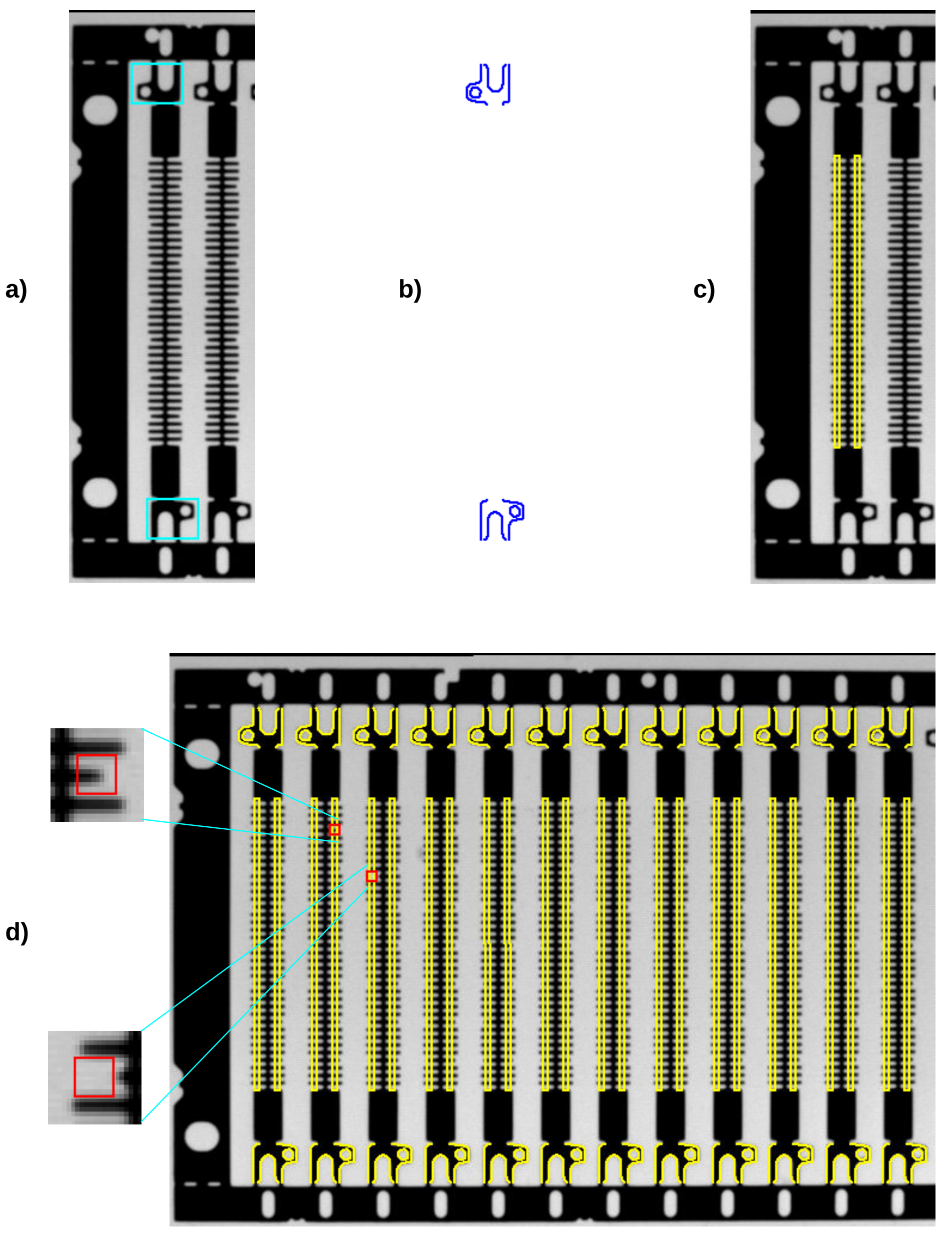


Figure 20: Aligning ROIs for inspecting parts of a razor: a) ROIs for the model; b) the model; c) measuring ROIs; d) inspection results with zoomed faults.

Step 1: Position the measurement ROIs for the model blade

```
Rect1Row := 244
Rect1Col := 73
DistColRect1Rect2 := 17
Rect2Row := Rect1Row
Rect2Col := Rect1Col + DistColRect1Rect2
RectPhi := rad(90)
RectLength1 := 122
RectLength2 := 2
```

First, two rectangular measurement ROIs are placed over the teeth of the razor blade acting as the model as shown in figure 20c.

Step 2: Move the reference point to the center of the first measure ROI

```
DistRect1CenterRow := Rect1Row - CenterROIRow
DistRect1CenterCol := Rect1Col - CenterROIColumn
set_shape_model_origin (ModelID, DistRect1CenterRow, DistRect1CenterCol)
```

Now, the reference point of the model is moved to the center of the first measure ROI using the operator `set_shape_model_origin`. As already mentioned, the operator expects not the absolute position of the new reference point, but its distance to the default reference point. Note that this step is only included to show how to use `set_shape_model_origin`; as described in section 3.4, the accuracy of the estimated position may decrease when using a modified point of reference.

Step 3: Find all razor blades

```
find_shape_model (SearchImage, ModelID, 0, 0, 0.8, 0, 0.5, 'interpolation',
                  0, 0.7, RowCheck, ColumnCheck, AngleCheck, Score)
```

Then, all instances of the model object are searched for in the image.

Step 4: Determine the affine transformation

```
for i := 0 to |Score|-1 by 1
  vector_angle_to_rigid (Rect1Row, Rect1Col, 0, RowCheck[i],
                         ColumnCheck[i], AngleCheck[i],
                         MovementOfObject)
```

For each razor blade, the transformation representing its position and orientation is calculated. Because the reference point was moved to the center of the first measure ROI, these coordinates are now used in the call to `vector_to_rigid`.

Step 5: Create measurement objects at the corresponding positions

```
  RectPhiCheck := RectPhi + AngleCheck[i]
  gen_measure_rectangle2 (RowCheck[i], ColumnCheck[i],
                          RectPhiCheck, RectLength1, RectLength2,
                          Width, Height, 'bilinear',
                          MeasureHandle1)
  affine_trans_point_2d (MovementOfObject, Rect2Row+0.5,
                         Rect2Col+0.5, Rect2RowTmp, Rect2ColTmp)
  Rect2RowCheck := Rect2RowTmp-0.5
  Rect2ColCheck := Rect2ColTmp-0.5
  gen_measure_rectangle2 (Rect2RowCheck, Rect2ColCheck,
                          RectPhiCheck, RectLength1, RectLength2,
                          Width, Height, 'bilinear',
                          MeasureHandle2)
```

Because the center of the first measure ROI serves as the reference point of the model, the returned position of the match can be used directly in the call to `gen_measure_rectangle2`. Unfortunately, there is only one point of reference. Therefore, the new position of the second measure ROI must be calculated explicitly using the operator `affine_trans_point_2d`. As remarked in section 4.3.2, the code adding and subtracting 0.5 to and from the point coordinates is necessary because the operator `affine_trans_point_2d` **and the HALCON regions (and XLDs) use different definitions of the position of a pixel**.

In the example application, the individual razor blades are only translated but not rotated relative to the model position. Instead of applying the full affine transformation to the measure ROIs and then creating new measure objects, one can therefore use the operator translate_measure to translate the measure objects themselves. The example program contains the corresponding code; you can switch between the two methods by modifying a variable at the top of the program.

Step 6: Measure the width and the distance of the "teeth"

```
measure_pairs (SearchImage, MeasureHandle1, 2, 25, 'negative',
               'all', RowEdge11, ColEdge11, Amp11, RowEdge21,
               ColEdge21, Amp21, Width1, Distance1)
measure_pairs (SearchImage, MeasureHandle2, 2, 25, 'negative',
               'all', RowEdge12, ColEdge12, Amp12, RowEdge22,
               ColEdge22, Amp22, Width2, Distance2)
```

Now, the actual measurements are performed using the operator measure_pairs.

Step 7: Inspect the measurements

```
NumberTeeth1 := |Width1|
if (NumberTeeth1 < 37)
  for j := 0 to NumberTeeth1 - 2 by 1
    if (Distance1[j] > 4.0)
      RowFault := round(0.5*(RowEdge11[j+1] + RowEdge21[j]))
      ColFault := round(0.5*(ColEdge11[j+1] + ColEdge21[j]))
      disp_rectangle2 (WindowHandle, RowFault, ColFault, 0,
                       4, 4)
```

Finally, the measurements are inspected. If a "tooth" is too short or missing completely, no edges are extracted at this point resulting in an incorrect number of extracted edge pairs. In this case, the faulty position can be determined by checking the distance of the teeth. Figure 20d shows the inspection results for the example.

Please note that the example program is not able to display the fault if it occurs at the first or the last tooth.

4.3.5 Rectifying the Search Results

In the previous section, the matching results were used to determine the so-called *forward transformation*, i.e., how objects are transformed from the model into the search image. Using this transformation, ROIs specified in the model image can be positioned correctly in the search image.

You can also determine the *inverse transformation* which transforms objects from the search image back into the model image. With this transformation, you can *rectify* the search image (or parts of it), i.e., transform it such that the matched object is positioned as it was in the model image. This method is useful if the following image processing step is not invariant against rotation, e.g., OCR or the variation model. Note that image rectification can also be useful *before* applying shape-based matching, e.g., if the camera observes the scene under an oblique angle; see section 5.1 for more information.

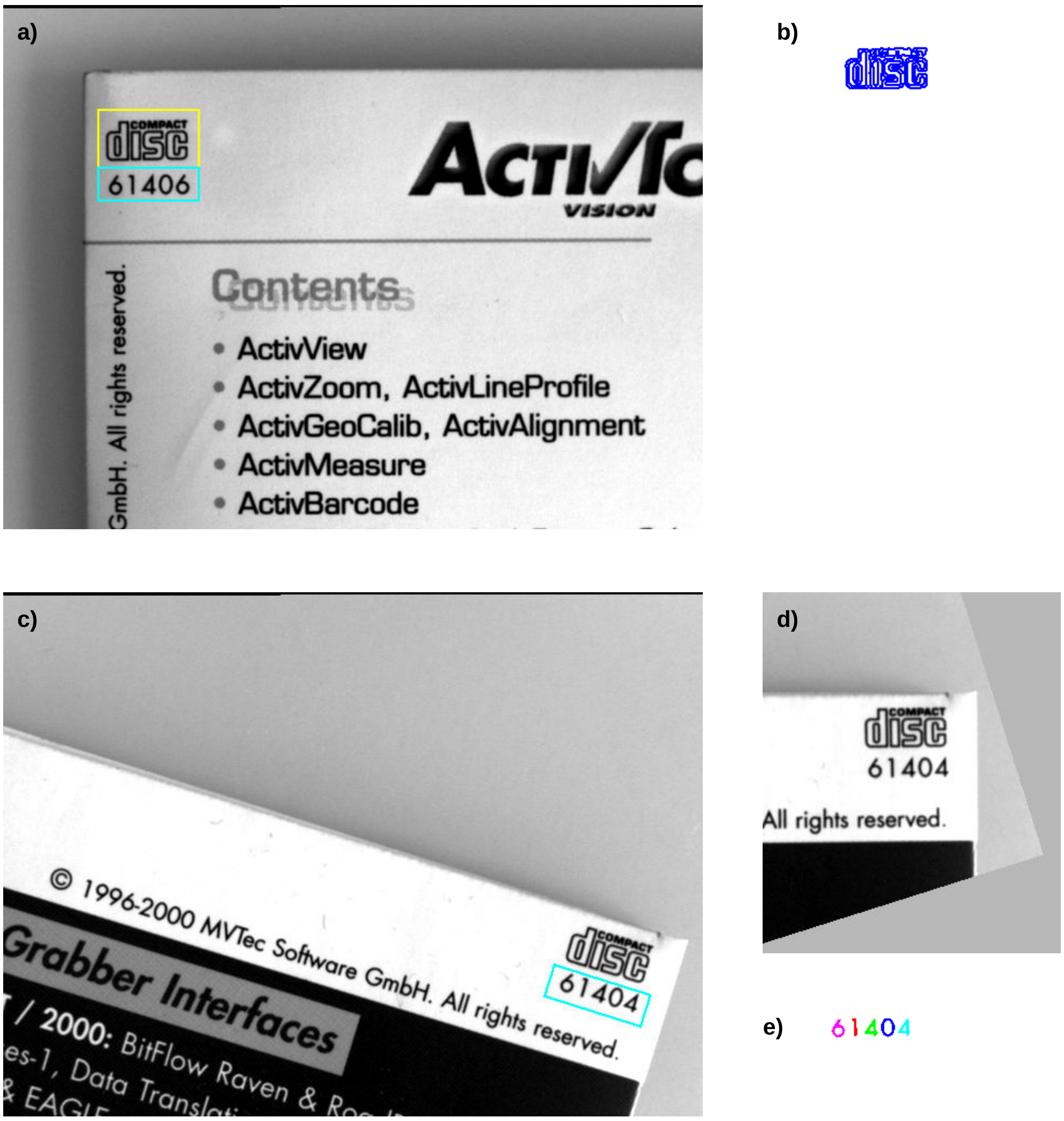


Figure 21: Rectifying the search results: a) ROIs for the model and for the number extraction; b) the model; c) number ROI at matched position; d) rectified search image (only relevant part shown); e) extracted numbers.

The inverse transformation can be determined and applied in a few steps, which are described below; in the corresponding example application of the HDevelop program `hdevelop\rectify_results.dev` the task is to extract the serial number on CD covers (see figure 21).

Step 1: Calculate the inverse transformation

```
hom_mat2d_invert (MovementOfObject, InverseMovementOfObject)
```

You can invert a transformation easily using the operator `hom_mat2d_invert`.

Step 2: Rectify the search image

```
affine_trans_image (SearchImage, RectifiedSearchImage,
                    InverseMovementOfObject, 'constant', 'false')
```

Now, you can apply the inverse transformation to the search image using the operator affine_trans_image. Figure 21d shows the resulting rectified image of a different CD; undefined pixels are marked in grey.

Step 3: Extract the numbers

```
reduce_domain (RectifiedSearchImage, NumberROI,
               RectifiedNumberROIImage)
threshold (RectifiedNumberROIImage, Numbers, 0, 128)
connection (Numbers, IndividualNumbers)
```

Now, the serial number is positioned correctly within the original ROI and can be extracted without problems. Figure 21e shows the result, which could then, e.g., be used as the input for OCR.

Unfortunately, the operator affine_trans_image transforms the full image even if you restrict its domain with the operator reduce_domain. In a time-critical application it may therefore be necessary to crop the search image before transforming it. The corresponding steps are visualized in figure 22.

Step 1: Crop the search image

```
smallest_rectangle1 (NumberROIAtNewPosition, Row1, Column1, Row2,
                     Column2)
crop_rectangle1 (SearchImage, CroppedNumberROIImage, Row1, Column1,
                 Row2, Column2)
```

First, the smallest axis-parallel rectangle surrounding the transformed number ROI is computed using the operator smallest_rectangle1, and the search image is cropped to this part. Figure 22b shows the resulting image overlaid on a grey rectangle to facilitate the comparison with the subsequent images.

Step 2: Create an extended affine transformation

```
hom_mat2d_translate (MovementOfObject, - Row1, - Column1,
                     MoveAndCrop)
hom_mat2d_invert (MoveAndCrop, InverseMoveAndCrop)
```

In fact, the cropping can be interpreted as an additional affine transformation: a translation by the negated coordinates of the upper left corner of the cropping rectangle (see figure 22a). We therefore "add" this transformation to the transformation describing the movement of the object using the operator hom_mat2d_translate, and then invert this extended transformation with the operator hom_mat2d_invert.

Step 3: Transform the cropped image

```
affine_trans_image (CroppedNumberROIImage, RectifiedROIImage,
                    InverseMoveAndCrop, 'constant', 'true')
reduce_domain (RectifiedROIImage, NumberROI,
               RectifiedNumberROIImage)
```

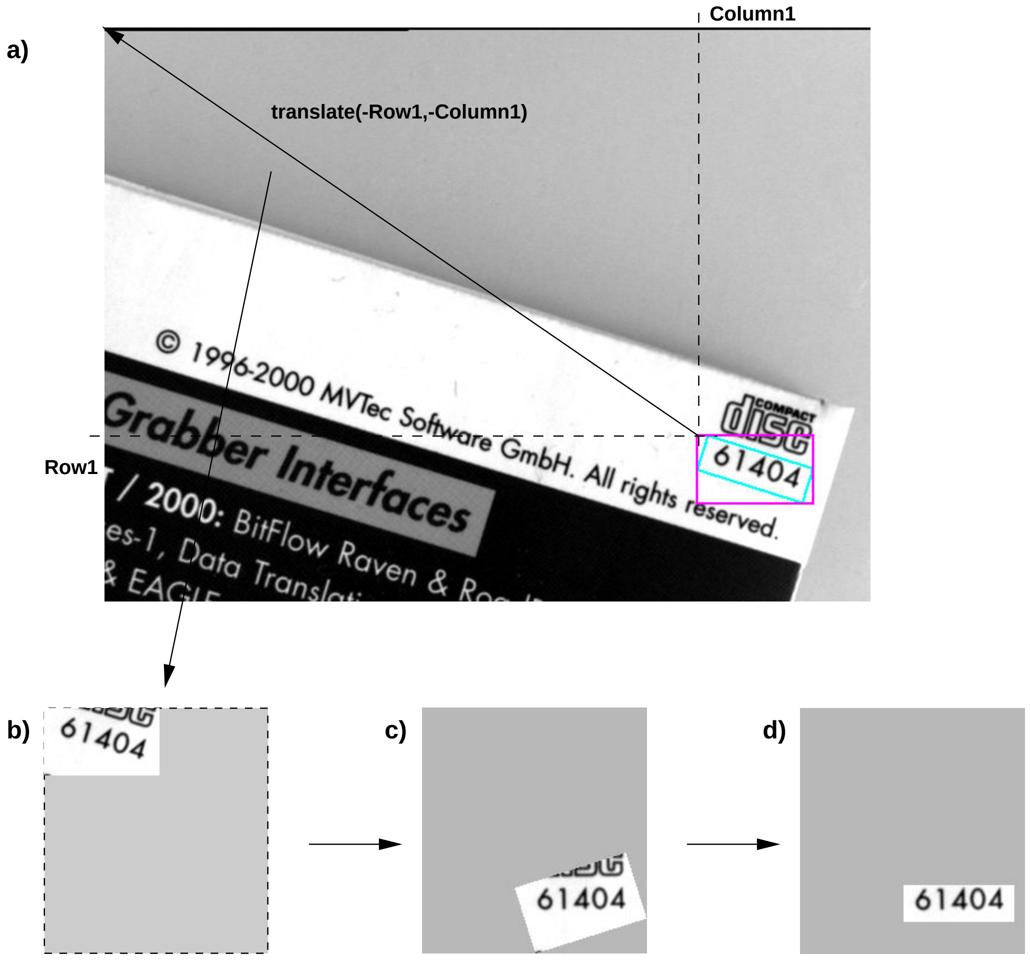


Figure 22: Rectifying only part of the search image: a) smallest image part containing the ROI; b) cropped search image; c) result of the rectification; d) rectified image reduced to the original number ROI.

Using the inverted extended transformation, the cropped image can easily be rectified with the operator affine_trans_image (figure 22c) and then be reduced to the original number ROI (figure 22d) in order to extract the numbers.

4.4 Using the Estimated Scale

Similarly to the rotation (compare section 4.3), the scaling is performed around the center of the ROI – if you didn't use set_shape_model_origin, that is. This is depicted in figure 23a at the example of an ROI whose center does not coincide with the center of the IC.

The estimated scale, which is returned in the parameter Scale, can be used similarly to the position and orientation. However, there is no convenience operator like vector_angle_to_rigid that creates an affine transformation including the scale; therefore, the scaling must be added separately. How to achieve this is explained below; in the corresponding example HDevelop program hdevelop\multiple_scales.dev, the task is to find nuts of varying sizes and to determine suitable points for grasping them (see figure 24).

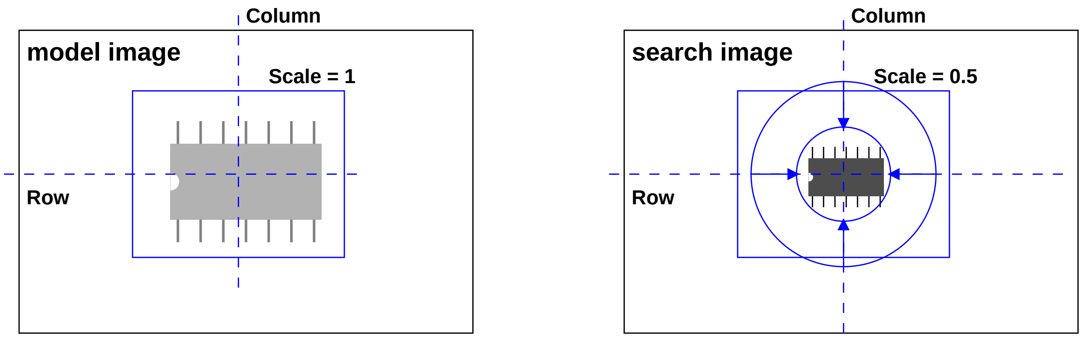


Figure 23: The center of the ROI acts as the point of reference for the scaling.

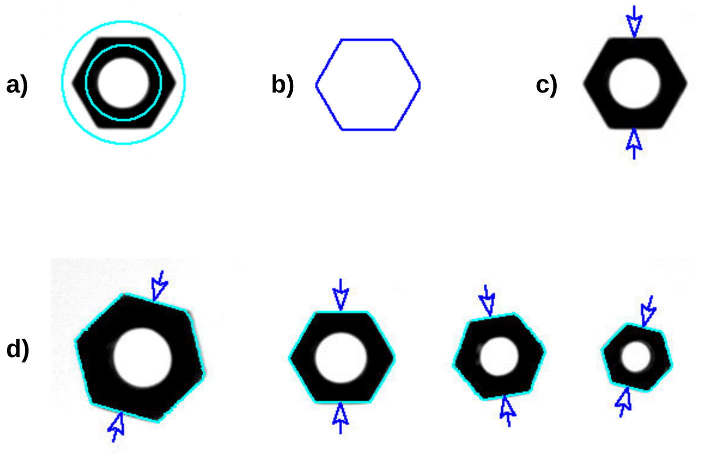


Figure 24: Determining grasping points on nuts of varying sizes: a) ring-shaped ROI; b) model; c) grasping points defined on the model nut; d) results of the matching.

Step 1: Specify grasping points

```
RowUpperPoint := 284
ColUpperPoint := 278
RowLowerPoint := 362
ColLowerPoint := 278
```

In the example program, the grasping points are specified directly in the model image; they are marked with arrows in figure 24c.

Step 2: Determine the complete transformation

```
find_scaled_shape_model (SearchImage, ModelID, -rad(30), rad(60), 0.6, 1.4,
                         0.9, 0, 0, 'interpolation', 0, 0.8, RowCheck,
                         ColumnCheck, AngleCheck, ScaleCheck, Score)
for i := 0 to |Score| - 1 by 1
  vector_angle_to_rigid (CenterROIRow, CenterROIColumn, 0, RowCheck[i],
                         ColumnCheck[i], AngleCheck[i], MovementOfObject)
  hom_mat2d_scale (MovementOfObject, ScaleCheck[i], ScaleCheck[i],
                   RowCheck[i], ColumnCheck[i], MoveAndScalingOfObject)
  affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
                            MoveAndScalingOfObject)
```

After the matching, first the translational and rotational part of the transformation is determined with the operator vector_angle_to_rigid as in the previous sections. Then, the scaling is added using the operator hom_mat2d_scale. Note that the position of the match, i.e., the transformed center of the ROI, is used as the point of reference; this becomes necessary because the scaling is performed "after" the translation and rotation. The resulting, complete transformation can be used as before to display the model at the position of the matches.

Step 3: Calculate the transformed grasping points

```
affine_trans_point_2d (MoveAndScalingOfObject, RowUpperPoint+0.5,
                       ColUpperPoint+0.5, TmpRowUpperPoint,
                       TmpColUpperPoint)
affine_trans_point_2d (MoveAndScalingOfObject, RowLowerPoint+0.5,
                       ColLowerPoint+0.5, TmpRowLowerPoint,
                       TmpColLowerPoint)
RowUpperPointCheck := TmpRowUpperPoint-0.5
ColUpperPointCheck := TmpColUpperPoint-0.5
RowLowerPointCheck := TmpRowLowerPoint-0.5
ColLowerPointCheck := TmpColLowerPoint-0.5
```

Of course, the affine transformation can also be applied to other points in the model image with the operator affine_trans_point_2d. In the example, this is used to calculate the position of the grasping points for all nuts; they are marked with arrows in figure 24d.

! As noted in section 4.3.2, the code adding and subtracting 0.5 to and from the point coordinates is necessary because the operator affine_trans_point_2d **and the HALCON regions (and XLDs) use different definitions of the position of a pixel**.

5 Miscellaneous

5.1 Adapting to a Changed Camera Orientation

As shown in the sections above, HALCON's shape-based matching allows to localize objects even if their position and orientation in the image or their scale changes. However, the shape-based matching fails if the camera observes the scene under an oblique angle, i.e., if it is not pointed perpendicularily at the plane in which the objects move, because an object then appears distorted due to perspective projection; even worse, the distortion changes with the position and orientation of the object.

In such a case we recommend to rectify images *before* applying the matching. This is a three-step process: First, you must *calibrate* the camera, i.e., determine its position and orientation and other parameters, using the operator camera_calibration. Secondly, the calibration data is used to create a mapping function via the operator gen_image_to_world_plane_map, which is then applied to images with the operator map_image. For more information please refer to the HDevelop example program pm_world_plane.dev, which can be found in the hdevelop\Applications\FA of the directory %HALCONROOT%\examples.

5.2 Reusing Models

If you want to reuse created models in other HALCON applications, all you need to do is to store the relevant information in files and then read it again. The following example code stems from the HDevelop program `hdevelop\reuse_model.dev`. First, a model is created and the corresponding XLD contour and the reference point are determined:

```
create_scaled_shape_model (ImageROI, 0, -rad(30), rad(60), 0, 0.6, 1.4, 0,
                           'none', 'use_polarity', 60, 10, ModelID)
inspect_shape_model (ImageROI, ShapeModelImage, ShapeModelRegion, 1, 30)
gen_contours_skeleton_xld (ShapeModelRegion, ShapeModel, 1, 'filter')
area_center (ModelROI, Area, CenterROIRow, CenterROIColumn)
```

Then, this information is stored in files using the operators `write_shape_model` (for the model), `write_contour_xld_arc_info` (for the XLD contour), and `write_tuple` (for the reference point, whose coordinates have been concatenated into a tuple first):

```
write_shape_model (ModelID, ModelFile)
write_contour_xld_arc_info (ShapeModel, XLDFile)
ReferencePoint := [CenterROIRow, CenterROIColumn]
write_tuple (ReferencePoint, RefPointFile)
```

In the example program, all shape models are cleared to represent the start of another application.

The model, the XLD contour, and the reference point are now read from the files using the operators `read_shape_model`, `read_contour_xld_arc_info`, and `read_tuple`, respectively. Furthermore, the parameters used to create the model are accessed with the operator `get_shape_model_params`:

```
read_shape_model (ModelFile, ReusedModelID)
read_contour_xld_arc_info (ReusedShapeModel, XLDFile)
read_tuple (RefPointFile, ReusedReferencePoint)
ReusedCenterROIRow := ReusedReferencePoint[0]
ReusedCenterROICol := ReusedReferencePoint[1]
get_shape_model_params (ReusedModelID, NumLevels, AngleStart, AngleExtent,
                        AngleStep, ScaleMin, ScaleMax, ScaleStep, Metric,
                        MinContrast)
```

Now, the model can be used as if it was created in the application itself:

```
find_scaled_shape_model (SearchImage, ReusedModelID, AngleStart,
                         AngleExtent, ScaleMin, ScaleMax, 0.9, 0, 0,
                         'interpolation', 0, 0.8, RowCheck, ColumnCheck,
                         AngleCheck, ScaleCheck, Score)
for i := 0 to |Score| - 1 by 1
  vector_angle_to_rigid (ReusedCenterROIRow, ReusedCenterROICol, 0,
                         RowCheck[i], ColumnCheck[i], AngleCheck[i],
                         MovementOfObject)
  hom_mat2d_scale (MovementOfObject, ScaleCheck[i], ScaleCheck[i],
                   RowCheck[i], ColumnCheck[i], MoveAndScalingOfObject)
  affine_trans_contour_xld (ReusedShapeModel, ModelAtNewPosition,
                            MoveAndScalingOfObject)
  dev_display (ModelAtNewPosition)
endfor
```

Application Note

The Art of Image Acquisition

Provided Functionality

- ▷ Connecting to simple and complex configurations of frame grabbers and cameras
- ▷ Acquiring images in various timing modes
- ▷ Configuring frame grabbers and cameras online

Involved Operators

open_framegrabber
info_framegrabber
grab_image, grab_image_async, grab_image_start
set_framegrabber_param, get_framegrabber_param
close_framegrabber, close_all_framegrabbers

gen_image1, gen_image3, gen_image1_extern

Overview

Obviously, the acquisition of images is a task to be solved in all machine vision applications. Unfortunately, this task mainly consists of interacting with special, non-standardized hardware in form of the frame grabber board. To let you concentrate on the actual machine vision problem, HALCON already provides interfaces performing this interaction for a large number of frame grabbers (see section 1).

Within your HALCON application, the task of image acquisition is thus reduced to a few lines of code, i.e., a few operator calls, as can be seen in section 2. What's more, this simplicity is not achieved at the cost of limiting the available functionality: Using HALCON, you can acquire images from various configurations of frame grabbers and cameras (see section 3) in different timing modes (see section 5).

Unless specified otherwise, the example programs can be found in the subdirectory `image_acquisition` of the directory `%HALCONROOT%\examples\application_guide`. Note that most programs are preconfigured to work with a certain HALCON frame grabber interface; in this case, the name of the program contains the name of the interface. To use the program with another frame grabber, please adapt the parts which open the connection to the frame grabber. More example programs for the different HALCON frame grabber interfaces can be found in the subdirectory `hdevelop\Image\Framegrabber` of the directory `%HALCONROOT%\examples`.

Please refer to the HALCON/C User's Manual and the HALCON/C++ User's Manual for information about how to compile and link the C and C++ example programs; among other things, they describe how to use the example UNIX makefiles which can be found in the subdirectories `c` and `cpp` of the directory `%HALCONROOT%\examples`. Under Windows, you can use Visual Studio workspaces containing the examples, which can be found in the subdirectory `i586-nt4` parallel to the source files.

Contents

1 The Philosophy Behind the HALCON Frame Grabber Interfaces

From the point of view of an user developing software for a machine vision application, the acquisition of images is only a prelude to the actual machine vision task. Of course it is important that images are acquired at the correct moment or rate, and that the camera and the frame grabber are configured suitably, but these tasks seem to be elementary, or at least independent of the used frame grabber.

The reality, however, looks different. Frame grabbers differ widely regarding the provided functionality, and even if their functionality is similar, the SDKs (*software development kit*) provided by the frame grabber manufacturers do not follow any standard. Therefore, if one decides to switch to a different frame grabber, this probably means to rewrite the image acquisition part of the application.

HALCON's answer to this problem are its *frame grabber interfaces* (HFGI) which are provided for currently more than 50 frame grabbers in form of *dynamically loadable libraries* (Windows NT/2000/XP: DLLs; UNIX: shared libraries). HALCON frame grabber interfaces bridge the gap between the individual frame grabbers and the HALCON library, which is independent of the used frame grabber, computer platform, and programming language (see figure 25). In other words, they

- provide a standardized interface to the HALCON user in form of 11 HALCON operators, and
- encapsulate details specific to the frame grabber, i.e., the interaction with the frame grabber SDK provided by the manufacturer.

Therefore, if you decide to switch to a different frame grabber, all you need to do is to install the corresponding driver and SDK provided by the manufacturer and to use different parameter values when calling the HALCON operators; the operators themselves stay the same.

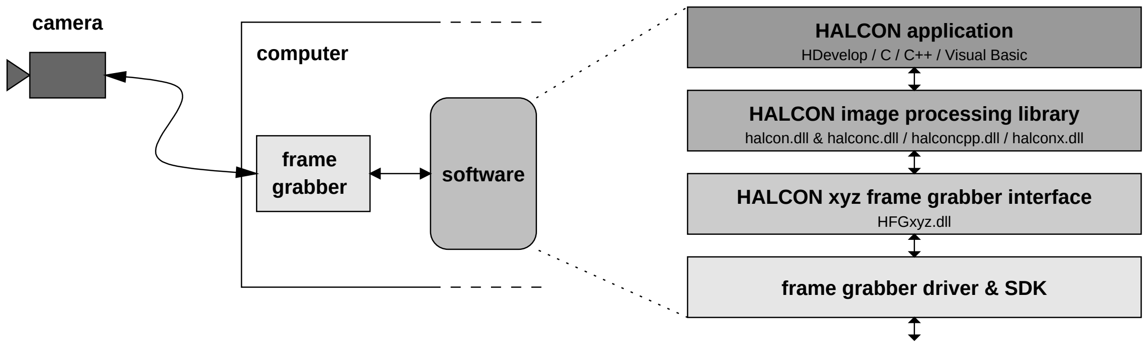

camera
computer
frame grabber
software
HALCON application
HDevelop / C / C++ / Visual Basic
HALCON image processing library
halcon.dll & halconc.dll / halconcpp.dll / halconx.dll
HALCON xyz frame grabber interface
HFGxyz.dll
frame grabber driver & SDK

Figure 25: From the camera to a HALCON application.

In fact, the elementary tasks of image acquisition are covered by two HALCON operators:

- `open_framegrabber` connects to the frame grabber and sets general parameters, e.g., the type of the used camera or the port the camera is connected to, then
- `grab_image` (or `grab_image_async`, see section 5.1 for the difference) grabs images.

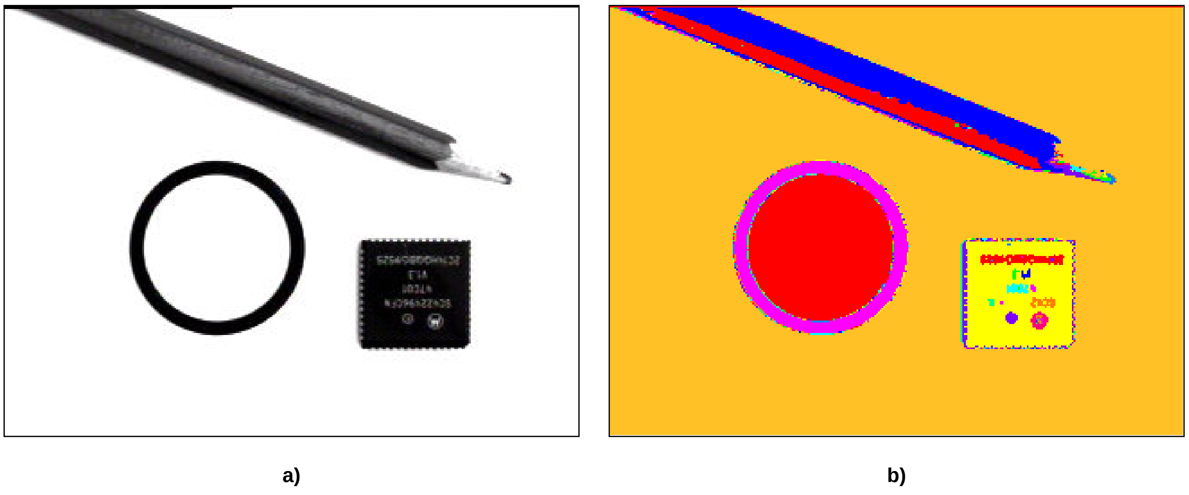


Figure 26: a) Acquired image; b) processed image (automatic segmentation).

If a frame grabber provides additional functionality, e.g., on-board modification of the image signal, special grabbing modes, or digital output lines, it is available via the operator `set_framegrabber_param` (see section 4).

Note, that for some frame grabbers not the full functionality is available within HALCON; please refer to the corresponding online documentation which can be found in the directory `%HALCONROOT%\doc\html\manuals` or via the HALCON folder in the Windows start menu (if you installed the documentation). The latest information can be found under `http://www.mvtec.com/halcon/framegrabber`.

If the frame grabber you want to use is not (yet) supported by HALCON, you can nevertheless use it together with HALCON. Please refer to section 6.1 for more details.

2 A First Example

In this section we start with a simple image acquisition task, which uses the frame grabber in its default configuration and the standard grabbing mode. The grabbed images are then segmented. To follow the example actively, start the HDevelop program `hdevelop\first_example_acquisition_ids.dev`; the steps described below start after the initialization of the application (press F5 once to reach this point). Note that the program is preconfigured for the HALCON frame grabber interface `IDS`; to use it with a different frame grabber, please adapt the parts which open the connection.

Step 1: Connect to the frame grabber

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'gray', -1,
                   'false', 'ntsc', 'default', -1, -1, FGHandle)
```

When opening the connection to your frame grabber using the operator `open_framegrabber`, the main parameter is the `Name` of the corresponding HALCON frame grabber interface. As a result, you obtain a so-called *handle* (`FGHandle`) which acts as your access to the frame grabber, e.g., in calls to the operator `grab_image`.

In the example, default values are used for most other parameters ('default' or -1); section 4.1 takes a closer look at this topic. How to connect to more complex frame grabber and camera configurations is described in section 3.

Step 2: Grab an image

```
grab_image (Image, FGHandle)
```

After successfully connecting to your frame grabber you can grab images by calling the operator `grab_image` with the corresponding handle `FGHandle`. More advanced modes of grabbing images are described in section 5.

Step 3: Grab and process images in a loop

```
while (Button # 1)
  grab_image (Image, FGHandle)
  auto_threshold (Image, Regions, 4)
  connection (Regions, ConnectedRegions)
  get_mposition (WindowHandleButton, Row, Column, Button)
endwhile
```

In the example, the grabbed images are then automatically segmented using the operator `auto_threshold` (see figure 26). This is done in a loop which can be exited by clicking into a window with the left mouse button.

3 Connecting to Your Frame Grabber

In this section, we show how to connect to different configurations of frame grabber(s) and camera(s), ranging from the simple case of one camera connected to one frame grabber board to more complex ones, e.g., multiple synchronized cameras connected to one or more boards.

3.1 Opening a Connection to a Specified Configuration

With the operator `open_framegrabber` you open a connection to a frame grabber, or to be more exact, via a frame grabber to a camera. This connection is described by four parameters (see figure 27): First, you select a frame grabber (family) with the parameter `Name`. If multiple boards are allowed, you can select one with the parameter `Device`; depending on the frame grabber interface, this parameter can contain a string describing the board or simply a number (in form of a string!).

Typically, the camera can be connected to the frame grabber at different ports, whose number can be selected via the parameter `Port` (in rare cases `LineIn`). The parameter `CameraType` describes the connected camera: For analog cameras, this parameter usually specifies the used signal norm, e.g., 'ntsc'; more complex frame grabber interfaces use this parameter to select a camera configuration file.

As a result, `open_framegrabber` returns a *handle* for the opened connection in the parameter `FGHandle`. Note that if you use HALCON's COM or C^{++} interface and call the operator via the classes `HFramegrabberX` or `HFramegrabber`, no handle is returned because the instance of the class itself acts as your handle.

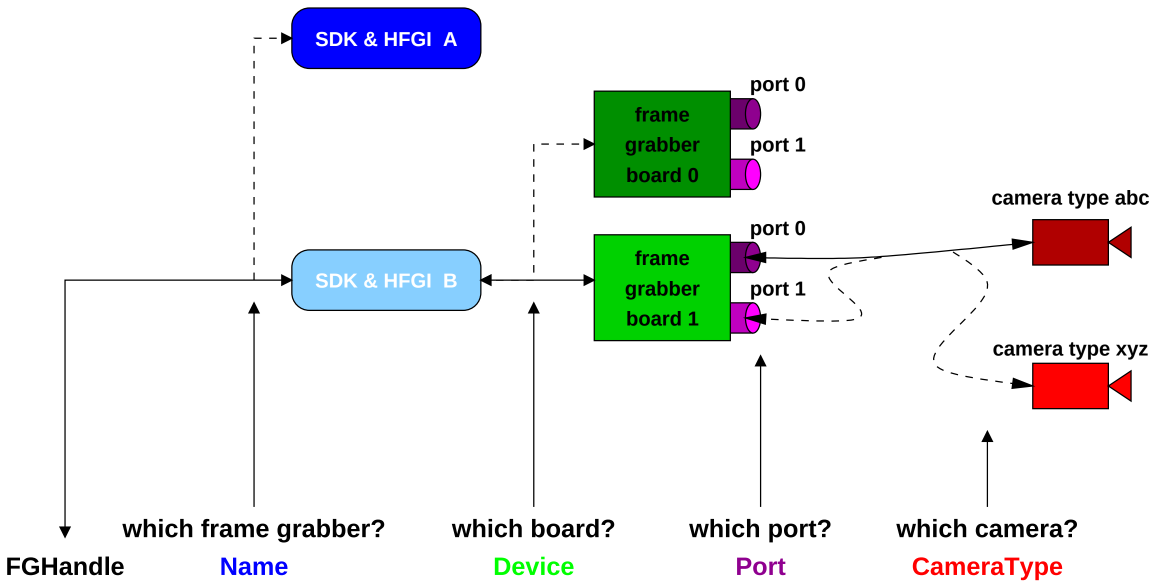
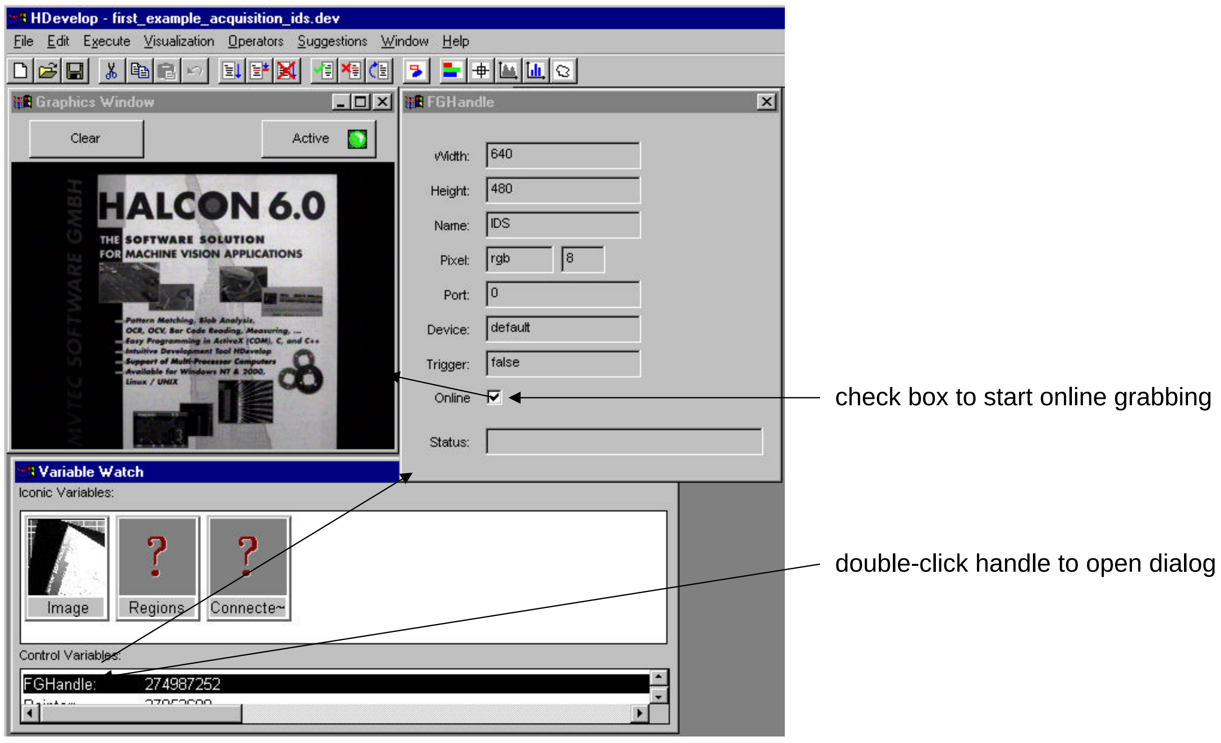


Figure 27: Describing a connection with the parameters of open_framegrabber .


HDevelop - first_example_acquisition_ids.dev
File Edit Execute Visualization Operators Suggestions Window Help
Graphics Window
Clear
Active
FGHandle
Variable Watch
Iconic Variables:
Image
Regions
Connecte~
Control Variables:
check box to start online grabbing
double-click handle to open dialog

Width:	640
Height:	480
Name:	IDS
Pixel:	rgb 8
Port:	0
Device:	default
Trigger:	false
Online	☑
Status:	

FGHandle: 274987252

Figure 28: Online grabbing in HDevelop .

In HDevelop, you can quickly check an opened connection by double-clicking FGHandle in the Variable Window as shown in figure 28. A dialog appears which describes the status of the connection. If you check the corresponding box, images are grabbed online and displayed in the Graphics Window. This mode is very useful to setup your vision system (illumination, focus, field of view).

3.2 Connecting to Multiple Boards and Cameras

Most HALCON frame grabbers interfaces allow to use multiple frame grabber boards and cameras. However, there is more than one way to connect cameras and boards and to access these configurations from within HALCON. Below, we describe the different configurations; please check the online documentation of the HALCON interface for your frame grabber (see `%HALCONROOT%\doc\html\manuals`, the HALCON folder in the Windows start menu, or `http://www.mvtec.com/halcon/framegrabber`) which configurations it supports.

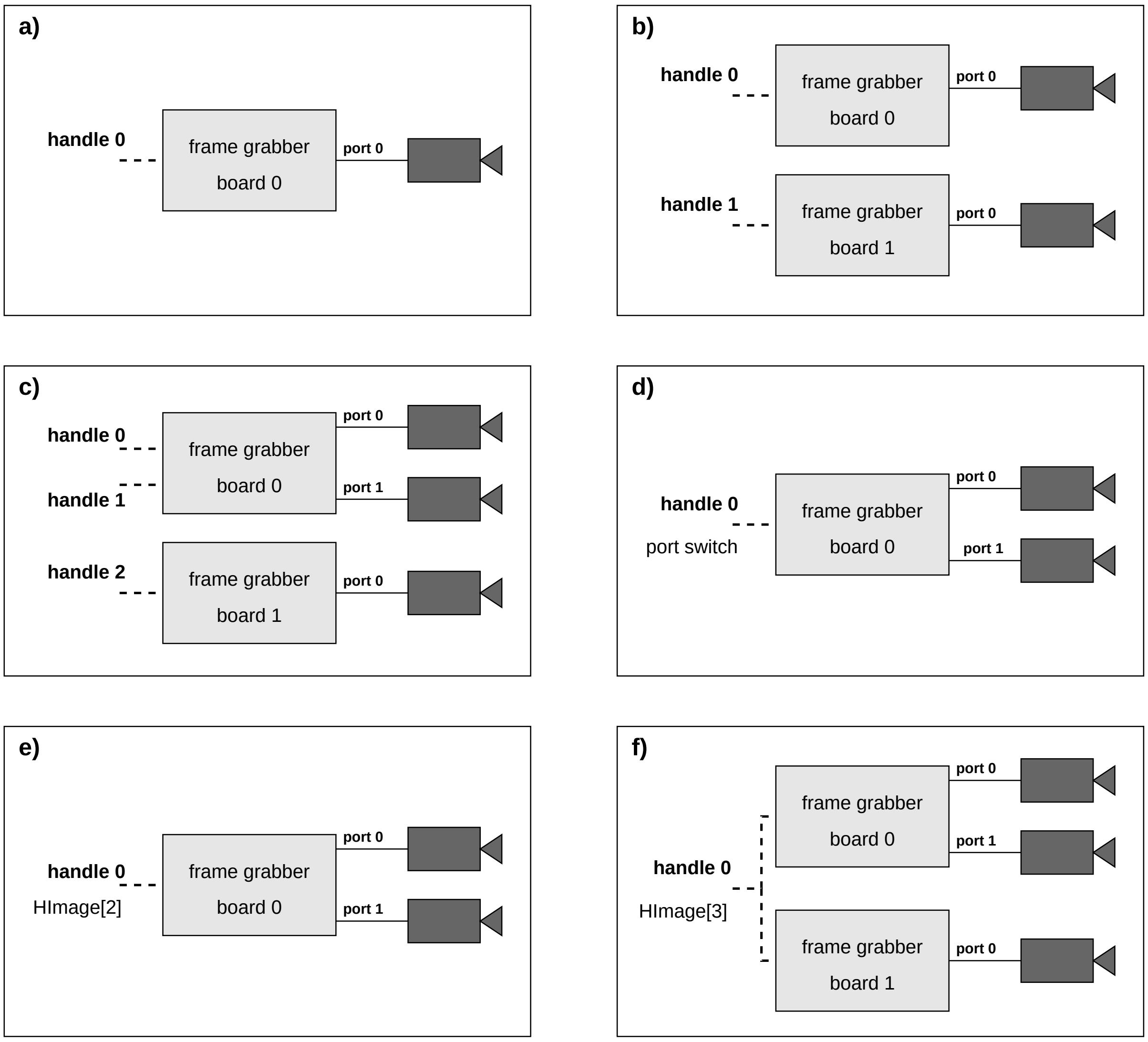


Figure 29: a) single board with single camera; b) multiple boards with one camera each; c) multiple boards with one or more cameras; d) single board with multiple cameras and port switching; e) single board with multiple cameras and simultaneous grabbing; f) simultaneous grabbing with multiple boards and cameras.

3.2.1 Single Camera

Figure 29a shows the simplest configuration: a single camera connected to a single board, accessible via a single handle. Some frame grabbers, especially digital ones, only support this configuration; as described in the following section, you can nevertheless use multiple cameras with such frame grabbers by connecting each one to an individual board.

3.2.2 Multiple Boards

Figure 29b shows a configuration with multiple cameras, each connected to a separate board. In this case you call the operator open_framegrabber once for each connection as in the HDevelop example program hdevelop\multiple_boards_px.dev. Note that the program is preconfigured for the HALCON Px interface; to use it with a different frame grabber, please adapt the parts which open the connection.

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', Board0, -1, -1, FGHandle0)
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', Board1, -1, -1, FGHandle1)
```

In this example, the two calls differ only in the value for the parameter Device ('0' and '1'); of course, you can use different values for other parameters as well, and even connect to different frame grabber interfaces.

To grab images from the two cameras, you simply call the operator grab_image once with the two handles returned by the two calls to open_framegrabber:

```
grab_image (Image0, FGHandle0)
grab_image (Image1, FGHandle1)
```

3.2.3 Multiple Handles Per Board

Many frame grabbers provide multiple input ports and thus allow to connect more than one camera to the board. Depending on the HALCON frame grabber interface, this configuration is accessed in different ways which are described in this and the following sections.

The standard HALCON method to connect to the cameras is depicted in figure 29c: Each connection gets its own handle, i.e., open_framegrabber is called once for each camera with different values for the parameter Port, like in the HDevelop example program hdevelop\multiple_ports_px.dev (preconfigured for the HALCON Px interface, please adapt the parts which open the connection for your own frame grabber):

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', 'default', Port0, -1, FGHandle0)
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', 'default', Port1, -1, FGHandle1)
grab_image (Image0, FGHandle0)
grab_image (Image1, FGHandle1)
```

As figure 29c shows, you can also use multiple boards with multiple connected cameras.

3.2.4 Port Switching

Some frame grabber interfaces access the cameras not via multiple handles, but by switching the input port dynamically (see figure 29d). Therefore, open_framegrabber is called only once, like in the HDevelop example program `hdevelop\port_switching_inspecta.dev` (preconfigured for the HALCON `Inspecta` interface, please adapt the parts which open the connection for your own frame grabber):

```
Port1 := 4
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
```

Between grabbing images you switch ports using the operator `set_framegrabber_param` (see section 4.2 for more information about this operator):

```
while (1)
set_framegrabber_param (FGHandle, 'port', Port0)
disp_image (Image0, WindowHandle0)
set_framegrabber_param (FGHandle, 'port', Port1)
```

Note that port switching only works for compatible (similar) cameras because `open_framegrabber` is only called once, i.e., the same set of parameters values is used for all cameras. In contrast, when using multiple handles as described above, you can specify different parameter values for the individual cameras (with some board-specific limitations).

3.2.5 Simultaneous Grabbing

In the configurations described above, images were grabbed from the individual cameras by multiple calls to the operator `grab_image`. In contrast, some frame grabber interfaces allow to grab images from multiple cameras with a single call to `grab_image`, which then returns a multi-channel image (see figure 29e; appendix A.1 contains more information about multi-channel images). This mode is called *simultaneous grabbing* (or *parallel grabbing*); like port switching, it only works for compatible (similar) cameras. For example, you can use this mode to grab synchronized images from a stereo camera system.

In this mode, `open_framegrabber` is called only once, as can be seen in the HDevelop example program `hdevelop\simultaneous_grabbing_inspecta.dev` (preconfigured for the HALCON `Inspecta` interface, please adapt the parts which open the connection for your own frame grabber):

```
             TM-6701/6705 1-plane, HD out'
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
```

You can check the number of returned images (channels) using the operator `count_channels`

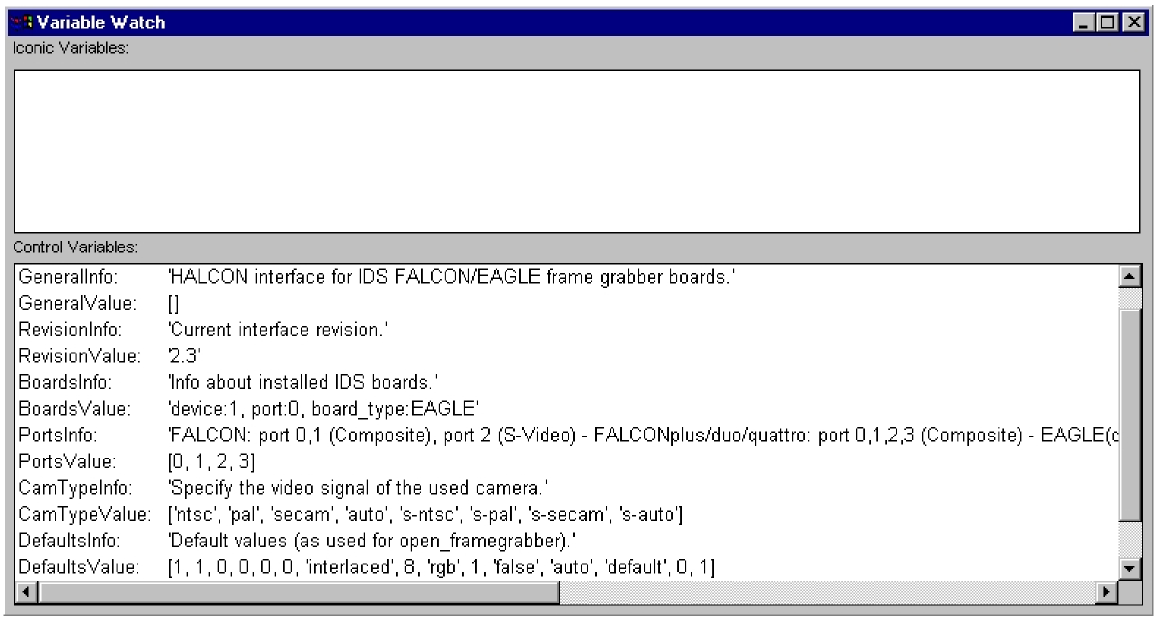
```
* step 2: open correctly sized windows
get_image_pointer1 (SimulImages, Pointer, Type, Width, Height)
```

and extract the individual images, e.g., using `decompose2`, `decompose3` etc., depending on the number of images:

```
grab_image (SimulImages, FGHandle)
if (num_channels = 2)
```

Alternatively, you can convert the multi-channel image into an image array using `image_to_channels` and then select the individual images via `select_obj`.

Note that some frame grabber interfaces allow simultaneous grabbing also for multiple boards (see figure 29f). Please refer to section 5.3.2 for additional information.



```
GeneralInfo:    'HALCON interface for IDS FALCON/EAGLE frame grabber boards.'
GeneralValue:   []
RevisionInfo:   'Current interface revision.'
RevisionValue:  '2.3'
BoardsInfo:     'Info about installed IDS boards.'
BoardsValue:    'device:1, port:0, board_type:EAGLE'
PortsInfo:      'FALCON: port 0,1 (Composite), port 2 (S-Video) - FALCONplus/duo/quattro: port 0,1,2,3 (Composite) - EAGLE(c
PortsValue:     [0, 1, 2, 3]
CamTypeInfo:    'Specify the video signal of the used camera.'
CamTypeValue:   ['ntsc', 'pal', 'secam', 'auto', 's-ntsc', 's-pal', 's-secam', 's-auto']
DefaultsInfo:   'Default values (as used for open_framegrabber).'
DefaultsValue:  [1, 1, 0, 0, 0, 0, 'interlaced', 8, 'rgb', 1, 'false', 'auto', 'default', 0, 1]
```

Figure 30: An example result of the operator info_framegrabber .

3.3 Requesting Information About the Frame Grabber Interface

As mentioned already, the individual HALCON frame grabber interfaces are described in detail on HTML pages which can be found in the directory `%HALCONROOT%\doc\html\manuals` or in the HALCON folder in the Windows start menu (if you installed the documentation). Another way to access information about a frame grabber interface is to use the operator `info_framegrabber`.

In the HDevelop example program `hdevelop\info_framegrabber_ids.dev` (preconfigured for the HALCON `IDS` interface, please adapt the interface name for your own frame grabber) this operator is called multiple times to query the version number of the interface, the available boards, port numbers, camera types, and the default values for all parameters of `open_framegrabber`; the result, i.e., the values displayed in the HDevelop Variable Windows, is depicted in figure 30.

```
info_framegrabber (FGName, 'general', GeneralInfo, GeneralValue)
info_framegrabber (FGName, 'revision', RevisionInfo, RevisionValue)
info_framegrabber (FGName, 'info_boards', BoardsInfo, BoardsValue)
info_framegrabber (FGName, 'ports', PortsInfo, PortsValue)
info_framegrabber (FGName, 'camera_types', CamTypeInfo, CamTypeValue)
info_framegrabber (FGName, 'defaults', DefaultsInfo, DefaultsValue)
```

The operator `info_framegrabber` can be called before actually connecting to a frame grabber with `open_framegrabber`. The only condition is that the HALCON frame grabber interface and the frame grabber SDK and driver have been installed.

4 Configuring the Acquisition

As explained in section 1, the intention of HALCON's frame grabber interfaces is to provide the user with a common interface for many different frame grabbers. This interface was kept as simple as possible; as shown, you can connect to your frame grabber and grab a first image using only two operators.

However, HALCON's second goal is to make the full functionality of a frame grabber available to the user. As frame grabbers differ widely regarding the provided functionality, this is a difficult task to realize within a simple, common interface. HALCON solves this problem by dividing the task of configuring a frame grabber connection into two parts: Those parameters which are common to most frame grabber interfaces (therefore called *general parameters*) are set when calling the operator `open_framegrabber`. In contrast, the functionality which is not generally available can be configured by setting so-called *special parameters* using the operator `set_framegrabber_param`.

4.1 General Parameters

When opening a connection via `open_framegrabber`, you can specify the following general parameters:

`HorizontalResolution`, `VerticalResolution`	spatial resolution of the transferred image in relation to the original size (see appendix B.1)
`ImageWidth`, `ImageHeight`, `StartRow`, `StartColumn`	size and upper left corner of the transferred image in relation to the original size (see appendix B.1)
`Field`	grabbing mode for analog cameras, e.g., interlaced-scan, progressive-scan, field grabbing (see appendix B.2)
`BitsPerChannel`, `ColorSpace`	data contained in a pixel (number of bits, number of channels, color encoding, see appendix B.3)
`Gain`	amplification factor for the video amplifier on the frame grabber board (if available)
`ExternalTrigger`	hooking the acquisition of images to an external trigger signal (see also section 5.2)
`CameraType`, `Device`, `Port`, `LineIn`	Configuration of frame grabber(s) and camera(s) from which images are to be acquired (see section 3.1).

In section 3.1, we already encountered the parameters describing the frame grabber / camera configuration. Most of the other parameters of `open_framegrabber` specify the image format; they are described in more detail in appendix B. The parameter `ExternalTrigger` activates a special grabbing mode which is described in detail in section 5.2. Finally, the parameter `Gain` can be used to manipulate the acquired images on the frame grabber board by configuring the video amplifier.

Note that when calling `open_framegrabber` you must specify values for all parameters, even if your frame grabber interface does not support some of them or uses values specified in a camera configuration file instead. To alleviate this task, the HALCON frame grabber interfaces provide suitable default values which are used if you specify 'default' or -1 for string or numeric parameters, respectively. The actually used default values can be queried using the operator info_framegrabber as shown in section 3.3.

After connecting to a frame grabber, you can query the current value of general parameters using the operator set_framegrabber_param; some interface even allow to modify general parameters dynamically. Please refer to section 4.3 for more information about these topics.

4.2 Special Parameters

Even the functionality which is not generally available for all frame grabber can be accessed and configured with a general mechanism: by setting corresponding special parameters via the operator set_framegrabber_param. Typical parameters are, for example:

'grab_timeout'	timeout after which the operators grab_image and grab_image_async stop waiting for an image and return an error (see also sections 5.2.1 and 6.2)
'volatile'	enable volating grabbing (see also section 5.1.3)
'continuous_grabbing'	switch on a special acquisition mode which is necessary for some frame grabbers to achieve real-time performance (see also section 5.1.5)
'trigger_signal'	signal type used for external triggering, e.g., rising or falling edge
'image_width', 'image_height', 'start_row', 'start_column', 'gain', 'external_trigger', 'port'	“doubles” of the some of the general parameters described in section 4.1, allowing to modify them dynamically, i.e., after opening the connection (see also section 4.3)

Depending on the frame grabber, various other parameters may be available, which allow, e.g., to add an offset to the digitized video signal or modify the brightness or contrast, to specify the exposure time or to trigger a flash. Some frame grabbers also offer special parameters for the use of line scan cameras (see also section 6.3), or parameters controlling digital output and input lines.

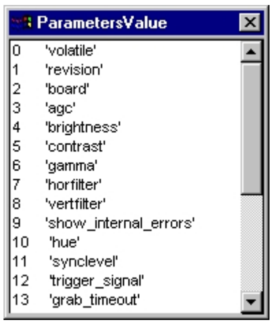
Which special parameters are provided by a frame grabber interface is described in the already mentioned online documentation. You can also query this information by calling the operator info_framegrabber as shown below; figure 31 depicts the result of double-clicking ParametersValue in the Variable Window after executing the line:

```
info_framegrabber (FGName, 'parameters', ParametersInfo, ParametersValue)
```

To set a parameter, you call the operator set_framegrabber_param, specifying the name of the parameter to set in the parameter Param and the desired value in the parameter Value. For example, in section 3.2.4 the following line was used to switch to port 0:

```
while (1)
```

You can also set multiple parameters at once by specifying tuples for Param and Value as in the following line:



```
0   'volatile'
1   'revision'
2   'board'
3   'agc'
4   'brightness'
5   'contrast'
6   'gamma'
7   'horfilter'
8   'vertfilter'
9   'show_internal_errors'
10  'hue'
11  'synclevel'
12  'trigger_signal'
13  'grab_timeout'
```

Figure 31: Querying available special parameters via info_framegrabber .

```
set_framegrabber_param (FGHandle, ['image_width','image_height'], [256,
                        256])
```

For all parameters which can be set with set_framegrabber_param, you can query the current value using the operator get_framegrabber_param. Some interfaces also allow to query additional information like minimum and maximum values for the parameters. For example, the HALCON Fire-i interface allows to query the minimum and maximum values for the brightness:

```
get_framegrabber_param (FGHandle, 'brightness_min_value', MinBrightness)
get_framegrabber_param (FGHandle, 'brightness_max_value', MaxBrightness)
```

Thus, you can check a new brightness value against those boundaries before setting it:

```
get_framegrabber_param (FGHandle, 'brightness', CurrentBrightness)
NewBrightness := CurrentBrightness + 10
if (NewBrightness > MaxBrightness)
  NewBrightness := MaxBrightness
endif
set_framegrabber_param (FGHandle, 'brightness', NewBrightness)
```

4.3 Fixed vs. Dynamic Parameters

The distinction between fixed and dynamic parameters is made relating to the lifetime of a frame grabber connection. *Fixed parameters*, e.g., the CameraType, are set once when opening the connection with open_framegrabber. In contrast, those parameters which can be modified via set_framegrabber_param during the use of the connection are called *dynamic parameters*.

As already noted in section 4.2, some frame grabber interfaces allow to modify general parameters like ImageWidth or ExternalTrigger dynamically via set_framegrabber_param, by providing a corresponding special parameter with the same name but written with small letters and underscores, e.g., 'image_width' or 'external_trigger'.

Independent of whether a general parameter can be modified dynamically, you can query its current value by calling the operator get_framegrabber_param with its “translated” name, i.e., capitals replaced by small letters and underscores as described above.

5 The Various Modes of Grabbing Images

Section 2 showed that grabbing images is very easy in HALCON– you just call `grab_image`! But of course there's more to image grabbing than just to get an image, e.g., how to assure an exact timing. This section therefore describes more complex grabbing modes.

5.1 Real-Time Image Acquisition

As a technical term, the attribute *real-time* means that a process guarantees that it meets given deadlines. Please keep in mind that **none of the standard operating systems, i.e., neither Windows NT/2000/XP nor Linux, are real-time operating systems**. This means that the operating system itself does not guarantee that your application will get the necessary processing time before its deadline expires. From the point of view of a machine vision application running under a non-real-time operating system, the most you can do is assure that real-time behavior is not already prevented by the application itself. **!**

In a machine vision application, real-time behavior may be required at multiple points:

Image delay: The camera must "grab" the image, i.e., expose the chip, at the correct moment, i.e., while the part to be inspected is completely visible.

Frame rate: The most common real-time requirement for a machine vision application is to "reach frame rate", i.e., acquire and process all images the camera produces.

Processing delay: The image processing itself must complete in time to allow a reaction to its results, e.g., to remove a faulty part from the conveyor belt. As this point relates only indirectly to the image acquisition it is ignored in the following.

5.1.1 Non-Real-Time Grabbing Using *grab_image*

Figure 32 shows the timing diagram for the standard grabbing mode, i.e., if you use the operator `grab_image` from within your application. This operator call is "translated" by the HALCON frame grabber interface and the SDK into the corresponding signal to the frame grabber board (marked with *'Grab'*).

The frame grabber now waits for the next image. In the example, a free-running analog progressive-scan camera is used, which produces images continuously at a fixed frame rate; the start of a new image is indicated by a so-called *vertical sync signal*. The frame grabber then digitizes the incoming analog image signal and transforms it into an image matrix. If a digital camera is used, the camera itself performs the digitizing and transfers a digital signal which is then transformed into an image matrix by the frame grabber. Please refer to appendix B.2 for more information about interlaced grabbing.

The image is then transferred from the frame grabber into computer memory via the PCI bus using *DMA* (direct memory access). This transfer can either be *incremental* as depicted in figure 32, if the frame grabber has only a FIFO buffer, or in a single burst as depicted in figure 33, if the frame grabber has a frame buffer on board. The advantage of the incremental transfer is that the transfer is concluded earlier. In contrast, the burst mode is more efficient; furthermore, if the incremental transfer via the PCI bus cannot proceed for some reason, a FIFO

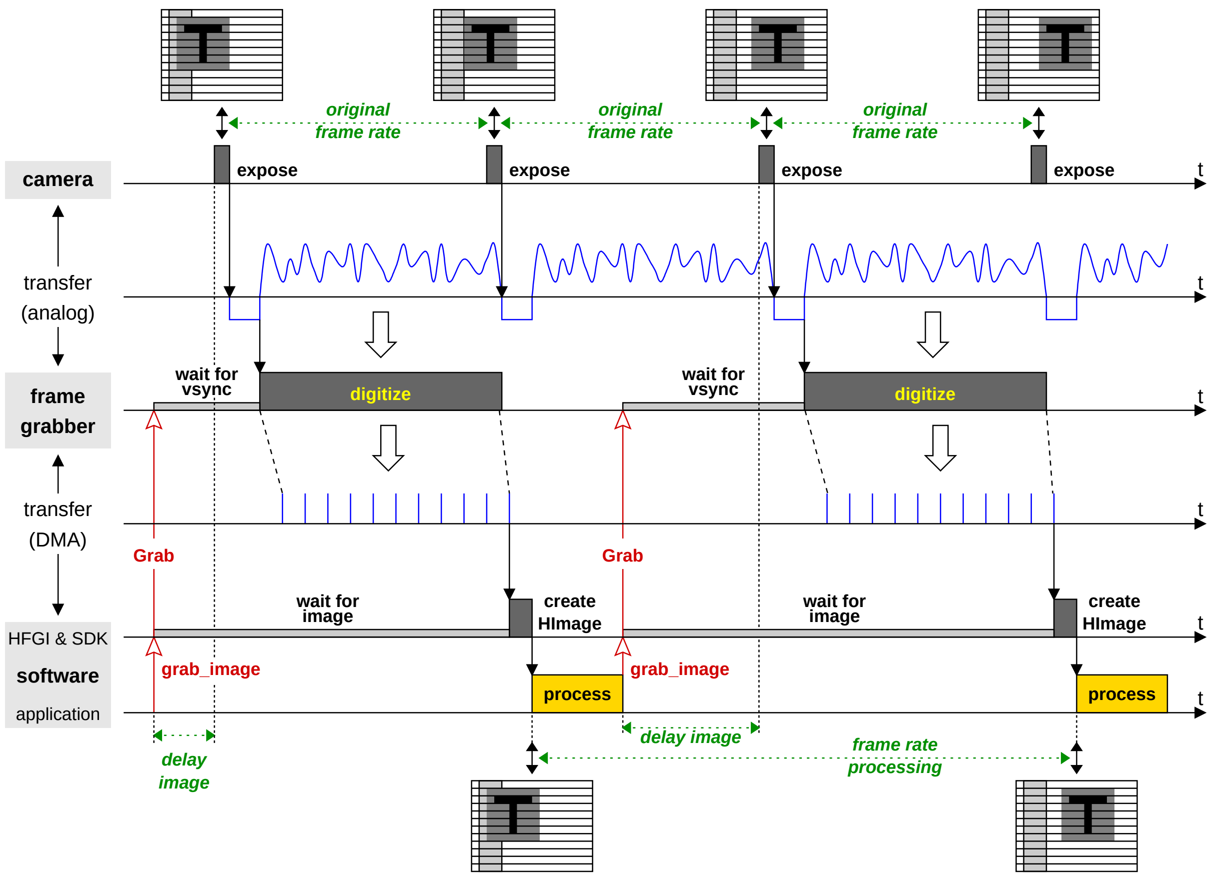


Figure 32: Standard timing using `grab_image` (configuration: free-running progressive-scan camera, frame grabber with incremental image transfer).

overflow results, i.e., image data is lost. Note that in both modes the transfer performance depends on whether the PCI bus is used by other devices as well!

When the image is completely stored in the computer memory, the HALCON frame grabber interface transforms it into a HALCON image and returns the control to the application which processes the image and then calls `grab_image` again. However, even if the processing time is short in relation to the frame rate, the camera has already begun to transfer the next image which is therefore "lost"; the application can therefore only process every second image.

You can check this behavior using the HDevelop example program `hdevelop\real_time_grabbing_ids.dev` (preconfigured for the HALCON IDS interface, please adapt the parts which open the connection for your own frame grabber), which determines achievable frame rates for grabbing and processing (here: calculating a difference image) first separately and then together as follows:

```
grab_image (BackgroundImage, FGHandle)
count_seconds (Seconds1)
for i := 1 to 20 by 1
  grab_image (Image, FGHandle)
  sub_image (BackgroundImage, Image, DifferenceImage, 1, 128)
endfor
count_seconds (Seconds2)
TimeGrabImage := (Seconds2-Seconds1)/20
FrameRateGrabImage := 1 / TimeGrabImage
```

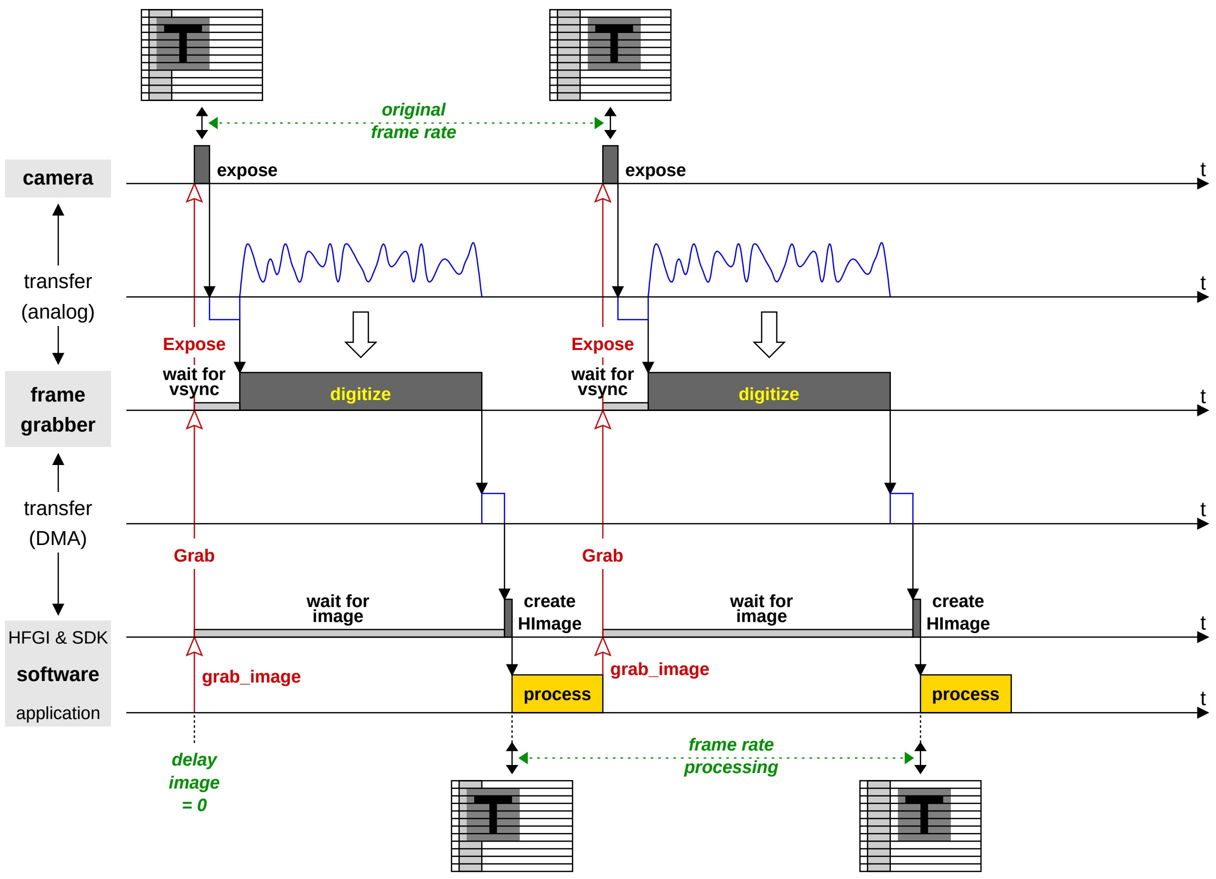


Figure 33: Using a asynchronously resettable camera together with grab_image (configuration: progressive-scan camera, frame grabber with burst transfer, volatile grabbing).

To see the non-deterministic image delay, execute the operator grab_image in the step mode by pressing $\boxed{\text{F6}}$; the execution time displayed in HDevelop's status bar will range between once and twice the original frame period. Please note that on UNIX system, the time measurements are performed with a lower resolution than on Windows systems.

5.1.2 Grabbing Without Delay Using Asynchronously Resettable Cameras

If you use a free-running camera, the camera itself determines the exact moment an image is acquired (exposed). This leads to a delay between the moment you call grab_image and the actual image acquisition (see figure 32). The delay is not deterministic, but at least it is limited by the frame rate; for example, if you use an NTSC camera with a frame rate of 30 Hz, the maximum delay can be 33 milliseconds.

Of course, such a delay is not acceptable in an application that is to inspect parts at a high rate. The solution is to use cameras that allow a so-called *asynchronous reset*. This means that upon a signal from the frame grabber, the camera resets the image chip and (almost) immediately starts to expose it. Typically, such a camera does not grab images continuously but only on demand.

An example timing diagram is shown in figure 33. In contrast to figure 32, the image delay is (almost) zero. Furthermore, because the application now specifies when images are to be grabbed, all images can be processed successfully; however, the achieved frame rate still includes the processing time and therefore may be too low for some machine vision applications.

5.1.3 Volatile Grabbing

As shown in figure 32, after the image has been transferred into the computer memory, the HALCON frame grabber interface needs some time to create a corresponding HALCON image which is then returned in the output parameter `Image` of `grab_image`. Most of this time (about 3 milliseconds on a 500 MHz Athlon K6 processor for a gray value NTSC image) is needed to copy the image data from the buffer which is the destination of the DMA into a newly allocated area.

You can switch off the copying by using the so-called *volatile grabbing*, which can be enabled via the operator `set_framegrabber_param` (parameter `'volatile'`):

```
set_framegrabber_param (FGHandle, 'volatile', 'enable')
```

Then, the time needed by the frame grabber interface to create the HALCON image is significantly reduced as visualized in figure 33. Note that usually volatile grabbing is only supported for gray value images!

The drawback of volatile grabbing is that grabbed images are overwritten by subsequent grabs. To be more exact, the overwriting depends on the number of image buffers allocated by the frame grabber interface or SDK. Typically, at least two buffers exist; therefore, you can safely process an image even if the next image is already being grabbed as in figure 35. Some frame grabber interfaces allow to use more than two buffers, and even to select their number dynamically via `set_framegrabber_param` (parameter `'num_buffers'`).

You can check this behavior using the HDevelop example program `hdevelop\volatile_grabbing_ids.dev` (preconfigured for the HALCON `IDS` interface, please adapt the parts which open the connection for your own frame grabber). After grabbing a first image and displaying it via

```
grab_image (FirstImage, FGHandle)
dev_open_window (0, 0, Width/2, Height/2, 'black', FirstWindow)
dev_display (FirstImage)
```

change the scene and grab a second image which is displayed in an individual window:

```
grab_image (SecondImage, FGHandle)
dev_open_window (0, Width/2 + 8, Width/2, Height/2, 'black', SecondWindow)
dev_display (SecondImage)
```

Now, images are grabbed in a loop and displayed in a third window. The two other images are also displayed each time. If you change the scene before each grab you can see how the first two images are overwritten in turn, depending on the number of buffers.

```
dev_open_window (Height/2 + 66, Width/4 + 4, Width/2, Height/2, 'black',
                 ThirdWindow)
for i := 1 to 10 by 1
  grab_image (CurrentImage, FGHandle)
  dev_set_window (ThirdWindow)
  dev_display (CurrentImage)
  dev_set_window (FirstWindow)
  dev_display (FirstImage)
  dev_set_window (SecondWindow)
  dev_display (SecondImage)
endfor
```

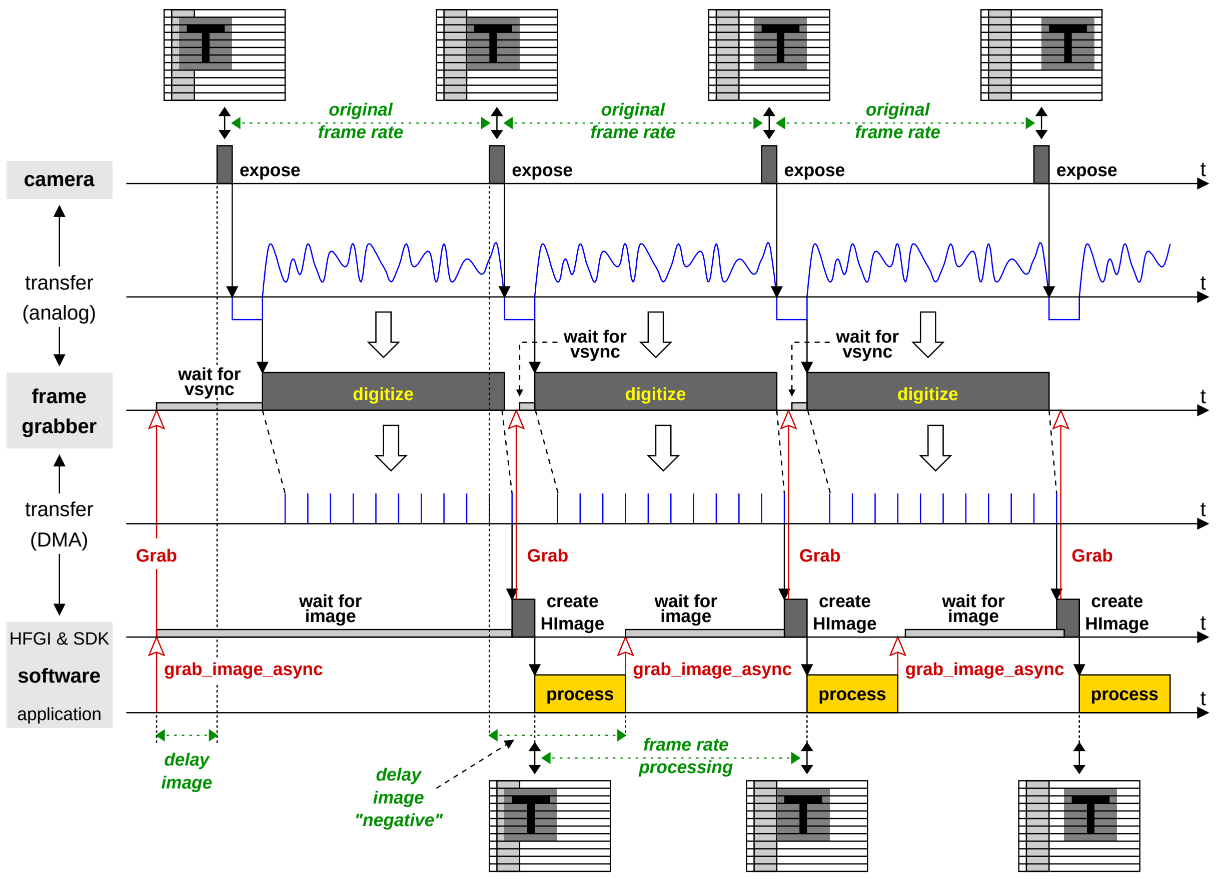


Figure 34: Grabbing and processing in parallel using grab_image_async .

5.1.4 Real-Time Grabbing Using *grab_image_async*

The main problem with the standard timing using grab_image is that the two processes of image grabbing and image processing run sequentially, i.e., one after the other. This means that the time needed for processing the image is included in the resulting frame rate, with the effect that the frame rate provided by the camera cannot be reached by definition.

This problem can be solved by using the operator grab_image_async. Here, the two processes are decoupled and can run asynchronously, i.e., **an image can be processed while the next image is already being grabbed**. Figure 34 shows a corresponding timing diagram: The first call to grab_image_async is processed similar to grab_image (compare figure 32). The difference becomes apparent after the transfer of the image into computer memory: Almost immediately after receiving the image, the frame grabber interface automatically commands the frame grabber to acquire a new image. Thus, the next image is grabbed while the application processes the previous image. After the processing, the application calls grab_image_async again, which waits until the already running image acquisition is finished. Thus, the full frame rate is now reached. Note that some frame grabbers fail to reach the full frame rate even with grab_image_async; section 5.1.5 shows how to solve this problem.

!

In the HDevelop example program hdevelop\real_time_grabbing_ids.dev, which was already described in section 5.1.1, the reached frame rate for asynchronous processing is determined as follows:

```
grab_image (BackgroundImage, FGHandle)
count_seconds (Seconds1)
for i := 1 to 20 by 1
  grab_image_async (Image, FGHandle, -1)
  sub_image (BackgroundImage, Image, DifferenceImage, 1, 128)
endfor
count_seconds (Seconds2)
TimeGrabImageAsync := (Seconds2-Seconds1)/20
FrameRateGrabImageAsync := 1 / TimeGrabImageAsync
```

As can be seen in figure 34, the first call to `grab_image_async` has a slightly different effect than the following ones, as it also triggers the first grab command to the frame grabber. As an alternative, you can use the operator `grab_image_start` which just triggers the grab command; then, the first call to `grab_image_async` behaves as the other ones. This is visualized, e.g., in figure 35; as you can see, the advantage of this method is that the application can perform some processing before calling `grab_image_async`.

In the example, the processing was assumed to be faster than the acquisition. If this is not the case, the image will already be ready when the next call to `grab_image_async` arrives. In this case, you can specify how "old" the image is allowed to be using the parameter `MaxDelay`. Please refer to section 5.1.7 for details.

Please note that when using `grab_image_async` it is not obvious anymore which image is returned by the operator call, because the call is decoupled from the command to the frame grabber! In contrast to `grab_image`, which always triggers the acquisition of a new image, `grab_image_async` typically returns an image which has been exposed before the operator was called, i.e., the image delay is negative (see figure 34)! Keep this effect in mind when changing parameters dynamically; contrary to intuition, the change will not affect the image returned by the next call of `grab_image_async` but by the following ones! Another problem appears when switching dynamically between cameras (see section 5.3.1).

5.1.5 Continuous Grabbing

For some frame grabbers `grab_image_async` fails to reach the frame rate because the grab command to the frame grabber comes too late, i.e., after the camera has already started to transfer the next image (see figure 35a).

As a solution to this problem, some frame grabber interfaces provide the so-called *continuous grabbing mode* which can be enables only via the operator `set_framegrabber_param` (parameter `'continuous_grabbing'`):

```
set_framegrabber_param (FGHandle, 'continuous_grabbing', 'enable')
```

In this mode, the frame grabber reads images from a free-running camera continuously and transfers them into computer memory as depicted in figure 35b. Thus, the frame rate is reached. If your frame grabber supports continuous grabbing you can test this effect in the example program `hdevelop\real_time_grabbing_ids.dev`, which was already described in the previous sections; the program measures the achievable frame rate for `grab_image_async` without and with continuous grabbing.

We recommend to use continuous grabbing only if you want to process every image; otherwise, images are transmitted over the PCI bus unnecessarily, thereby perhaps blocking other PCI transfers.

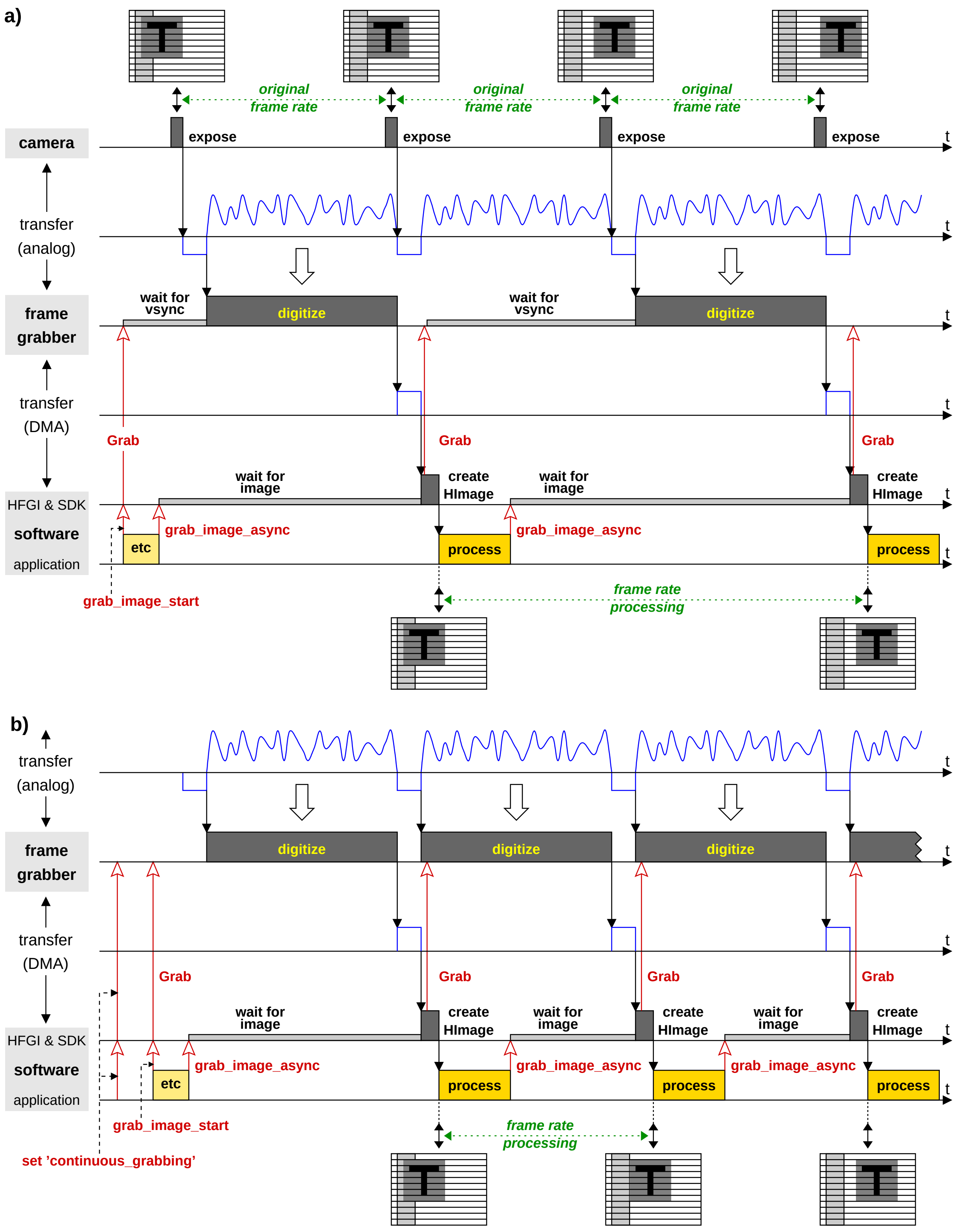


Figure 35: a) `grab_image_async` fails to reach frame rate; b) problem solved using continuous grabbing.

Note that some frame grabber interfaces provide additional functionality in the continuous grabbing mode, e.g., the HALCON `BitFlow` interface. Please refer to the corresponding documentation for more information.

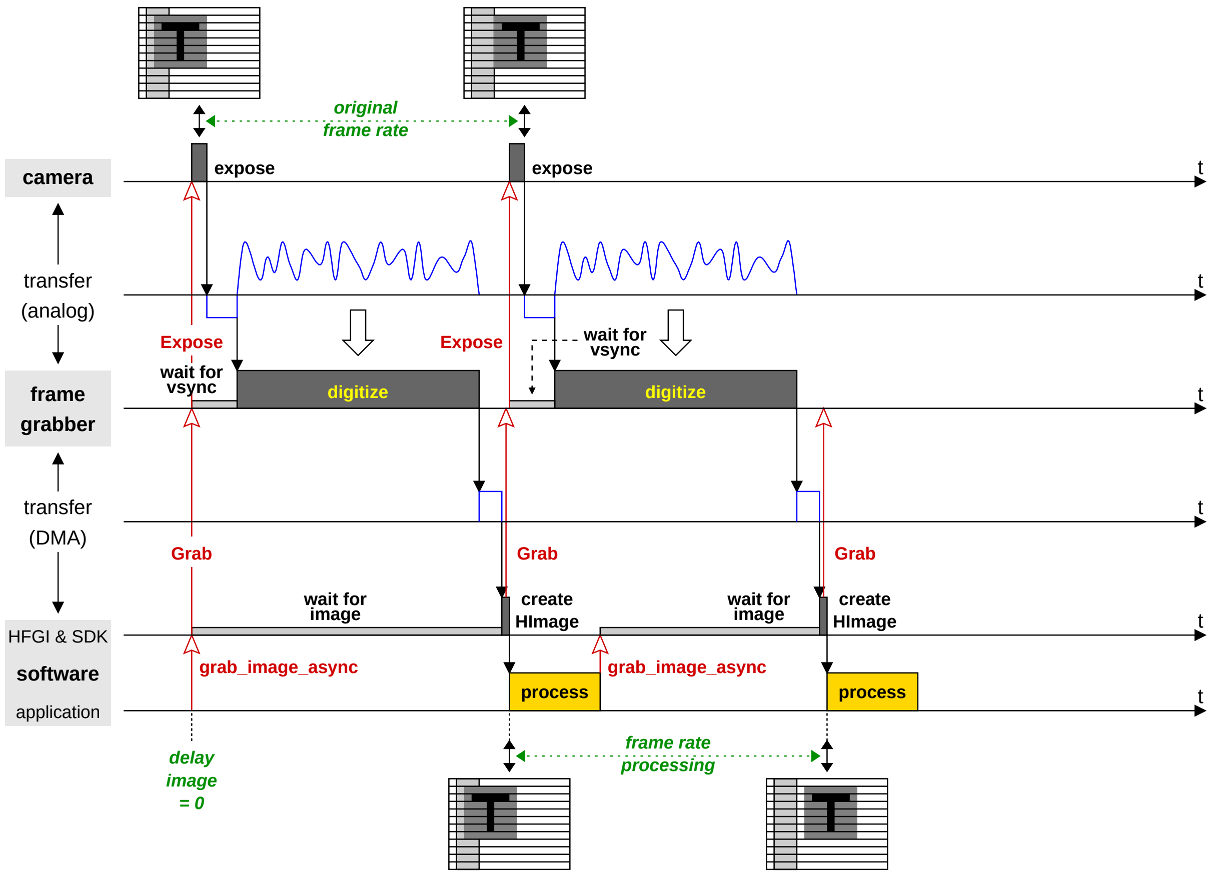


Figure 36: Using a asynchronously resettable camera together with `grab_image_async` (configuration as in figure 33.

5.1.6 Using *grab_image_async* Together With Asynchronously Resettable Cameras

As described in section 5.1.2, you can acquire images without delay by using an asynchronously resettable camera. Figure 36 shows the resulting timing when using such a camera together with `grab_image_async`. When comparing the diagram to the one in figure 33, you can see that a higher frame rate can now be reached, because the processing time is not included anymore.

5.1.7 Specifying a Maximum Delay

In contrast to `grab_image`, the operator `grab_image_async` has an additional parameter `MaxDelay`, which lets you specify how "old" an already grabbed image may be in order to be accepted. Figure 37 visualizes the effect of this parameter. There are two cases to distinguish: If the call to `grab_image` arrives before the next image has been grabbed (first call in the example), the parameter has no effect. However, if an image has been grabbed already (second and third call in the example), the elapsed time since the last grab command to the frame grabber is compared to `MaxDelay`. If it is smaller (second call in the example), the image is accepted; otherwise (third call), a new image is grabbed.

Please note that the delay is not measured starting from the moment the image is exposed, as you might perhaps expect! Currently, only a few frame grabber SDKs provide this information;

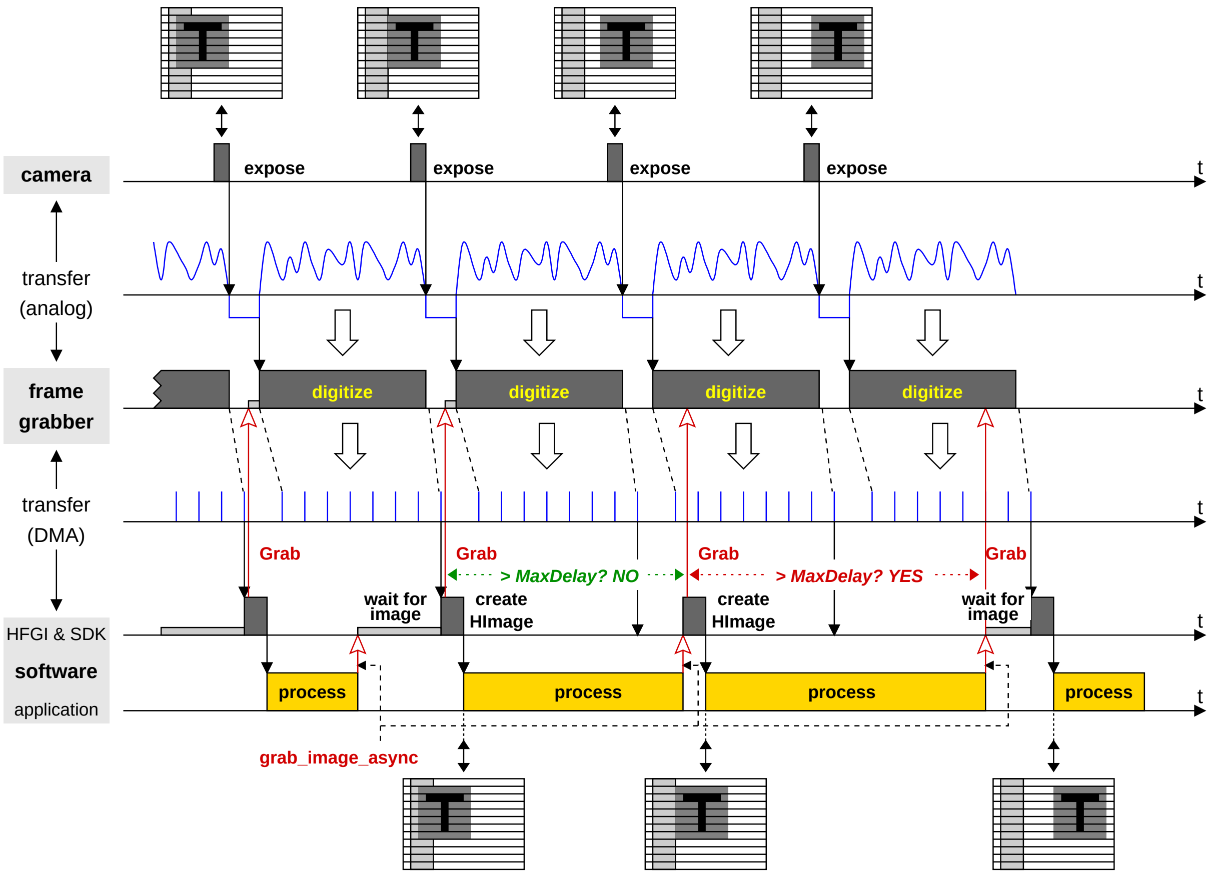


Figure 37: Specifying a maximum delay for grab_image_async (using continuous grabbing).

therefore, the last grab command from the interface to the the frame grabber is used as the starting point instead.

5.2 Using an External Trigger

In the previous section, the software performing the machine vision task decided when to acquire an image (*software trigger*). In industrial applications, however, the moment for image acquisition is typically specified externally by the process itself, e.g., in form of a hardware trigger signal indicating the presence of an object to be inspected. Most frame grabber boards are therefore equipped with at least one input line for such signals, which are called *external triggers*.

From HALCON's point of view, external triggers are dealt with by the frame grabber board, the only thing to do is to inform the frame grabber to use the trigger. You can do this simply by setting the parameter `ExternalTrigger` of `open_framegrabber` to `'true'`. Some frame grabber interfaces also allow to enable or disable the trigger dynamically using the operator `set_framegrabber_param` (parameter `'external_trigger'`).

Figure 38a shows the timing diagram when using an external trigger together with `grab_image` and a free-running camera. After the call to `grab_image`, the frame grabber board waits for the trigger signal. When it appears, the procedure described in the previous section follows: The frame grabber waits for the next image, digitizes it, and transfers it into computer memory;

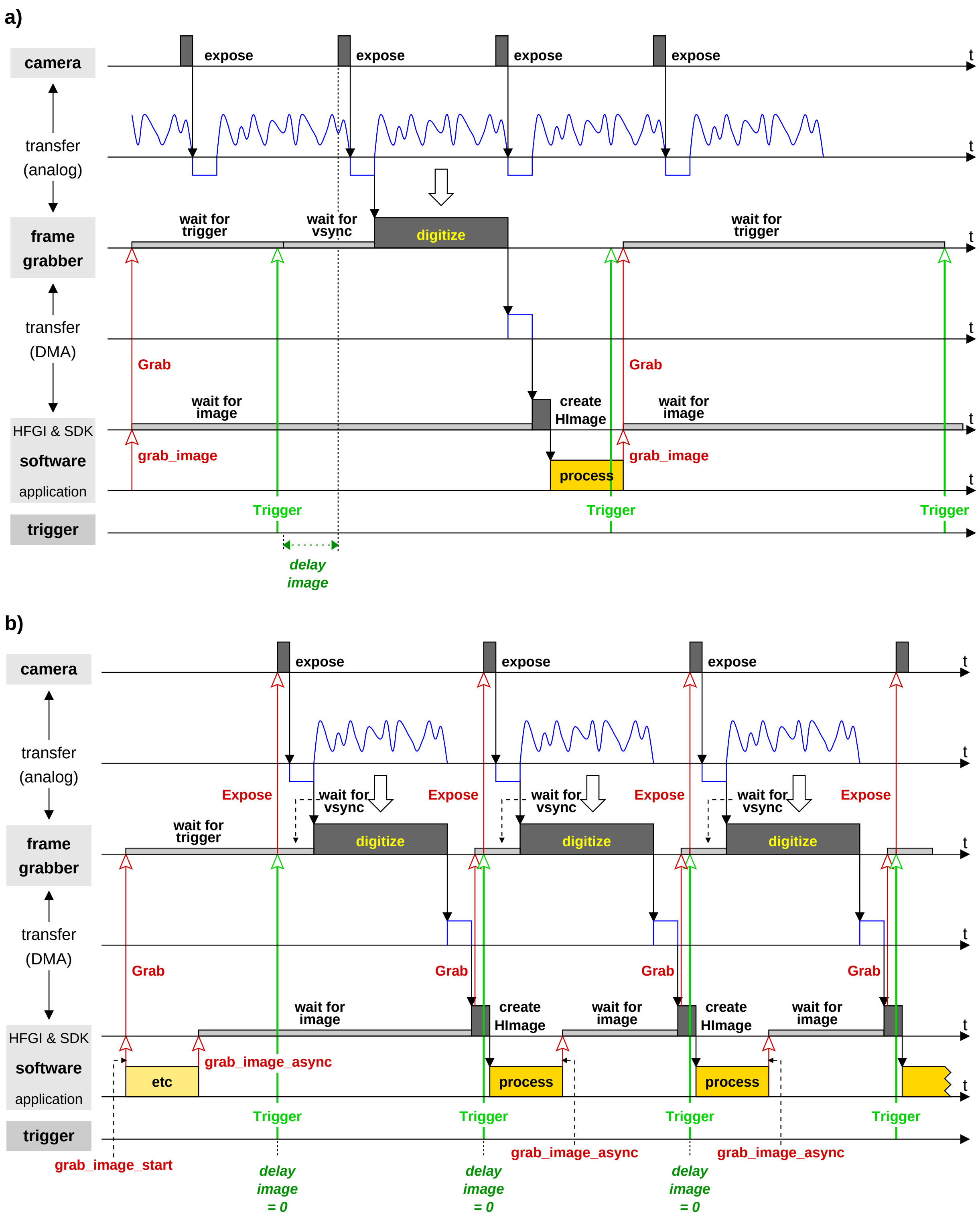


Figure 38: Using an external trigger together with: a) free-running camera and grab_image; b) asynchronously resettable camera and grab_image_async .

then, the HALCON frame grabber interface transforms it into a HALCON image and returns the control to the application which processes the image and then calls grab_image again, which causes the frame grabber board to wait for the next trigger signal.

The (bad) example in figure 38a was chosen on purpose to show an unsuitable configuration for using an external trigger: First of all, because of the free-running camera there is a non-deterministic delay between the arrival of the trigger signal and the exposure of the image, which may mean that the object to be inspected is not completely visible anymore. Secondly, because grab_image is used, trigger signals which arrive while the application is processing an image are lost.

Both problems can easily be solved by using an asynchronously resettable camera together with the operator grab_image_async as depicted in figure 38b.

The C++ example program cpp\error_handling_timeout_picport.cpp (preconfigured for the HALCON PicPort interface) shows how simple it is to use an external trigger: The connection is opened with ExternalTrigger set to 'true':

```
HFramegrabber     framegrabber;

framegrabber.OpenFramegrabber(fgname, 1, 1, 0, 0, 0, 0, "default", -1,
                              "gray", -1, "true", camtype, device,
                              -1, -1);
```

Then, images are grabbed:

```
HImage            image;

do
{
  image = framegrabber.GrabImageAsync(-1);
} while (button == 0);
```

The example contains a customized error handler which checks whether there is an external trigger; this part is described in detail in section 6.2.3.

5.2.1 Special Parameters for External Triggers

Most frame grabber interfaces allow to further configure the use of external triggering via the operator set_framegrabber_param. As mentioned in section 4.2, some interfaces allow to enable and disable the external trigger dynamically via the parameter 'external_trigger'. Another useful parameter is 'grab_timeout', which sets a timeout for the acquisition process (some interfaces provide an additional parameter 'trigger_timeout' just for triggered grabbing). Without such a timeout, the application would hang if for some reason no trigger signal arrives. In contrast, if a timeout is specified, the operators grab_image and grab_image_async only wait the specified time and then return an error code or raise an exception, depending on the programming language used. Section 6.2 shows how to handle such errors.

Other parameters allow to further specify the form of the trigger signal ('trigger_signal'), e.g., whether the falling or the rising edge is used as the trigger, select between multiple trigger input lines, or even filter trigger signals. Some frame grabber interfaces also allow to influence the exposure via the trigger signal.

5.3 Acquiring Images From Multiple Cameras

The timing diagrams shown in the previous sections depicted the case of a single camera. Below we discuss some issues which arise when acquiring images from multiple cameras (see section 3.2 for possible configurations).

5.3.1 Dynamic Port Switching and Asynchronous Grabbing

If you switch dynamically between multiple cameras connected to a single board as described in section 3.2.4 you must be careful when using `grab_image_async`: By default, the frame grabber interface commands the frame grabber board to grab the next image automatically after it received the current image — but before the next call of `grab_image_async`! If you switched to another camera before this call, the frame grabber might already be busy grabbing an image from the first camera.

Some frame grabber interfaces solve this problem by providing the parameter `'start_async_after_grab_async'` for the operator `set_framegrabber_param` which allows to disable the automatic grab command to the frame grabber board.

5.3.2 Simultaneous Grabbing

Some frame grabber interfaces provide special functionality to grab images *simultaneously* from multiple (synchronized) cameras. Typically, the cameras are connected to a single frame grabber board; the `PicPort` interface also allows to grab simultaneously from cameras connected to multiple boards. As described in section 3.2.5, the images are grabbed by a single call to `grab_image` or `grab_image_async`, which return them in form of a multi-channel image. Depending on the frame grabber interface, it may be necessary to switch on the simultaneous grabbing via the operator `set_framegrabber_param`.

Please keep in mind that even if a HALCON frame grabber interface supports simultaneous grabbing, this might not be true for every frame grabber board the interface supports! In order to grab multiple images simultaneously, a frame grabber board must be equipped with multiple "grabbing units"; for example, an analog frame grabber board must be equipped with multiple A/D converters. Please check this in the documentation of your frame grabber board.

Even if a HALCON frame grabber interface does not provide the special simultaneous grabbing mode, you can realize a similar behavior "manually", e.g., by connecting each (asynchronously resettable) camera to a single frame grabber board and then using a common external trigger signal to synchronize the grabbing.

6 Miscellaneous

6.1 Acquiring Images From Unsupported Frame Grabbers

If you want to use a frame grabber which is currently not supported by HALCON, i.e., for which no HALCON interface exists there exist two principal ways: First, you can create your own HALCON frame grabber interface; how to do this is described in detail in the Frame Grabber Integration Programmer's Manual .

As an alternative, you can pass externally created images, i.e., the raw image matrix, to HALCON using the operators `gen_image1`, `gen_image3`, or `gen_image1_extern`, which create a corresponding HALCON image. The main difference between the operators `gen_image1` and `gen_image1_extern` is that the former copies the image matrix when creating the HALCON image, whereas the latter doesn't, which is useful if you want to realize *volatile grabbing* as described in section 5.1.3.

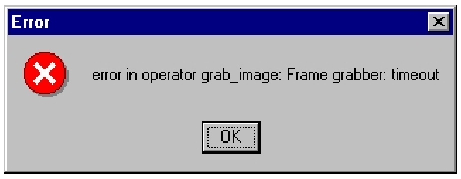
The C example program `c\use_extern_image.c` shows how to use the operator `gen_image1_extern` to pass standard gray value images to HALCON. In this case, the image matrix consists of 8 bit pixels (bytes), which can be represented by the data type `unsigned char`. At the beginning, the program calls a procedure which allocates memory for the images to be "grabbed"; in a real application this corresponds to the image buffer(s) used by the frame grabber SDK.

```
unsigned char *image_matrix_ptr;
long          width, height;

InitializeBuffer(&image_matrix_ptr, &width, &height);
```

The example program "simulates" the grabbing of images with a procedure which reads images from an image sequence and copies them into the image buffer. Then, the content of the image buffer is transformed into a HALCON image (type `byte`) via `gen_image1_extern`. The parameter `ClearProc` is set to `0` to signal that the program itself takes care of freeing the memory. The created HALCON image is then displayed. The loop can be exited by clicking into the HALCON window with any mouse button.

```
Hobject       image;
long          window_id;

open_window (0, 0, width, height, 0, "visible", "", &window_id);
while (!ButtonPressed(window_id))
{
  MyGrabImage((const unsigned char **) &image_matrix_ptr);
  gen_image1_extern(&image, "byte", width, height,
                    (long) image_matrix_ptr, (long) 0);
  disp_obj(image, window_id);
}
```

If your frame grabber supplies images with more than 8 bit pixels, you must adapt both the data type for the image matrix and the type of the created HALCON image (parameter `Type` of `gen_image1_extern`). In case of color images HALCON expects the image data in form of three separate image matrices. You can create a HALCON image either by calling the operator `gen_image3` with the three pointers to the matrices, or by calling the operator


Error

error in operator grab_image: Frame grabber: timeout

Figure 39: Popup dialog in HDevelop signaling a timeout.

gen_image1_extern three times and then using the operator channels_to_image to combine the three images into a multi-channel image. Please refer to appendix A for more information about HALCON images in general.

6.2 Error Handling

Just as the HALCON frame grabber interfaces encapsulate the communication with a frame grabber board, they also encapsulate occurring errors within the HALCON error handling mechanism. How to catch and react to these errors is described below, for HDevelop programs and also for programs using HALCON's programming language interfaces.

Some HALCON frame grabber interfaces provide special parameters for set_framegrabber_param which are related to error handling. The most commonly used one is the parameter 'grab_timeout' which specifies when the frame grabber should quit waiting for an image. The examples described in the following sections show how to handle the corresponding HALCON error.

Note that all example programs enable the signaling of low level errors via the operator set_system, e.g., in HDevelop syntax via

```
set_system ('do_low_error', 'true')
```

In this mode, low level errors occurring the frame grabber SDK (or in the HALCON interface) in are signaled by a message box.

6.2.1 Error Handling in HDevelop

The HDevelop example hdevelop\error_handling_timeout_picport.dev shows how to handle HALCON errors in a HDevelop program. To "provoke" an error, open_framegrabber is called with ExternalTrigger = 'true'. If there is no trigger, a call to grab_image results in a timeout; HDevelop reacts to this error with the popup dialog shown in figure 39 and stops the program.

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'true', CameraType, Device, -1, -1, FGHandle)
set_framegrabber_param (FGHandle, 'grab_timeout', 2000)
grab_image (Image, FGHandle)
```

HALCON lets you modify the reaction to an error with the operator set_check (in HDevelop: dev_set_check). If you set it to '~give_error', the program does not stop in case of an error but only stores its cause in form of an error code. To access this error code in HDevelop, you must define a corresponding variable using the operator dev_error_var. Note that this variable is updated after each operator call; to check the result of a single operator we therefore recommend to switch back into the standard error handling mode directly after the operator call as in the following lines:

```
dev_error_var (ErrorNum, 1)
dev_set_check ('~give_error')
grab_image (Image, FGHandle)
dev_error_var (ErrorNum, 0)
dev_set_check ('give_error')
```

To check whether a timeout occurred, you compare the error variable with the code signaling a timeout (5322); a list of error codes relating to image acquisition can be found in the Frame Grabber Integration Programmer's Manual . In the example, the timeout is handled by disabling the external trigger mode via the operator set_framegrabber_param (parameter 'external_trigger'). Then, the call to grab_image is tested again.

```
if (ErrorNum = 5322)
  set_framegrabber_param (FGHandle, 'external_trigger', 'false')
  dev_error_var (ErrorNum, 1)
  dev_set_check ('~give_error')
  grab_image (Image, FGHandle)
  dev_error_var (ErrorNum, 0)
  dev_set_check ('give_error')
endif
```

Now, the error variable should contain the value 2 signaling that the operator call succeeded; for this value HDevelop provides the constant H_MSG_TRUE. If you get another error code, the program accesses the corresponding error text using the operator get_error_text.

```
if (ErrorNum # H_MSG_TRUE)
  get_error_text (ErrorNum, ErrorText)
endif
```

If your frame grabber interface does not provide the parameter 'external_trigger', you can realize a similar behavior by closing the connection and then opening it again with ExternalTrigger set to 'false'.

6.2.2 Error Handling Using HALCON/C

The mechanism for error handling in a program based on HALCON/C is similar to the one in HDevelop; in fact, it is even simpler, because each operator automatically returns its error code. However, if a HALCON error occurs in a C program, the default error handling mode causes the program to abort.

The C example program c\error_handling_timeout_picport.c performs the same task as the HDevelop program in the previous section; if the call to grab_image succeeds, the program grabs and displays images in a loop, which can be exited by clicking into the window. The following lines show how to test whether a timeout occurred:

```
  set_check ("~give_error");
  error_num = grab_image (&image, fghandle);
  set_check ("give_error");
  switch (error_num)
  {
  case H_ERR_FGTIMEOUT:
```

As you see, in a C program you can use prefined constants for the error codes (see the Frame Grabber Integration Programmer's Manual for a list of image acquisition error codes and their corresponding constants).

6.2.3 Error Handling Using HALCON/C++

If your application is based on HALCON/C++, there are two methods for error handling: If you use operators in their C-like form, i.e., preceeded by a double colon (e.g., `::grab_image`), you can apply the same procedure as described for HALCON/C in the previous section.

In addition, HALCON/C++ provides an exception handling mechanism based on the class `HException`, which is described in the HALCON/C++ User's Manual . Whenever a HALCON error occurs, an instance of this class is created. The main idea is that you can specify a procedure which is then called automatically with the created instance of `HException` as a parameter. How to use this mechanism is explained in the C++ example program `cpp\error_handling_timeout_picport.cpp`, which performs the same task as the examples in the previous sections.

In the example program `cpp\error_handling_timeout_picport.cpp` (preconfigured for the HALCON `PicPort` interface), the procedure which is to be called upon error is very simple: It just raises a standard C++ exception with the instance of `HException` as a parameter.

```
void MyHalconExceptionHandler(const HException& except)
{
  throw except;
}
```

In the program, you "install" this procedure via a class method of `HException`:

```
int  main(int argc, char *argv[])
{
  HException::InstallHHandler(&MyHalconExceptionHandler);
```

Now, you react to a timeout with the following lines:

```
  try
  {
    image = framegrabber.GrabImage();
  }
  catch (HException except)
  {
    if (except.err == H_ERR_FGTIMEOUT)
    {
      framegrabber.SetFramegrabberParam("external_trigger", "false");
```

As already noted, if your frame grabber interface does not provide the parameter 'external_trigger', you can realize a similar behavior by closing the connection and then opening it again with ExternalTrigger set to 'false':

```
    if (except.err == H_ERR_FGTIMEOUT)
    {
      framegrabber.OpenFramegrabber(fgname, 1, 1, 0, 0, 0, 0, "default",
                                    -1, "gray", -1, "false", camtype,
                                    "default", -1, -1);
```

Note that when calling OpenFramegrabber via the class HFramegrabber as above, the operator checks whether it is called with an already opened connection and automatically closes it before opening it with the new parameters.

6.2.4 Error Handling Using HALCON/COM

The HALCON/COM interface uses the standard COM error handling technique where every method call passes both a numerical and a textual representation of the error to the calling framework. How to use this mechanism is explained in the Visual Basic example program vb\error_handling_timeout_picport\error_handling_timeout_picport.vbp, which performs the same task as the examples in the previous sections.

For each method, you can specify an error handler by inserting the following line at the beginning of the method:

```
On Error GoTo ErrorHandler
```

At the end of the method, you insert the code for the error handler. If a runtime error occurs, Visual Basic automatically jumps to this code, with the error being described in the variable Err. However, the returned error number does not correspond directly to the HALCON error as in the other programming languages, because low error numbers are reserved for COM. To solve this problem HALCON/COM uses an offset which must be subtracted to get the HALCON error code. This offset is accessible as a property of the class HSystemX:

```
ErrorHandler:
  Dim sys As New HSystemX
  ErrorNum = Err.Number - sys.ErrorBaseHalcon
```

The following code fragment checks whether the error is due to a timeout. If yes, the program disables the external trigger mode and tries again to grab an image. If the grab is successful the program continues at the point the error occurred; otherwise, the Visual Basic default error handler is invoked. Note that in contrast to the other programming languages HALCON/COM does not provide constants for the error codes.

```
  If (ErrorNum = 5322) Then
    Call FG.SetFramegrabberParam("external_trigger", "false")
    Set Image = FG.GrabImage
    Resume Next
```

If the error is not caused by a timeout, the error handler raises it anew, whereupon the Visual Basic default error handler is invoked.

```
  Else
    Err.Raise (Err.Number)
  End If
```

If your frame grabber interface does not provide the parameter 'external_trigger', you can realize a similar behavior by closing the connection and then opening it again with ExternalTrigger set to 'false'. Note that the class HFramegrabberX does not provide a method to close the connection; instead you must destroy the variable with the following line:

```
    Set FG = Nothing
```

6.3 Line Scan Cameras

From the point of view of HALCON there is no difference between area and line scan cameras: Both acquire images of a certain width an height; whether the height is 1, i.e., a single line, or larger does not matter. In fact, in many line scan applications the frame grabber combines multiple acquired lines to form a so-called *page* which further lessens the difference between the two camera types.

The main problem is therefore whether your frame grabber supports line scan cameras. If yes, you can acquire images from it via HALCON exactly as from an area scan camera. With the parameter ImageHeight of the operator open_framegrabber you can sometimes specify the height of the page; typically, this information is set in the camera configuration file. Some HALCON frame grabber interfaces allow to further configure the acquisition mode via the operator set_framegrabber_param.

The images acquired from a line scan camera can then be processed just like images from area scan cameras. However, line scan images often pose an additional problem: The objects to inspect may be spread over multiple images (pages). To solve this problem, HALCON provides special operators: tile_images allows to merge images into a larger image, merge_regions_line_scan and merge_cont_line_scan_xld allow to merge the (intermediate) processing results of subsequent images.

How to use these operators is explained in the HDevelop example program hdevelop\line_scan.dev. The program is based on an image file sequence which is read using the HALCON virtual frame grabber interface File; the task is to extract paper clips and calculate their orientation. Furthermore, the gray values in a rectangle surrounding each clip are determined.

An important parameter for the merging is over how many images an object can be spread. In the example, a clip can be spread over 4 images:

```
MaxImagesRegions := 4
```

The continuous processing is realized by a simple loop: At each iteration, a new image is grabbed, and the regions forming candidates for the clips are extracted using thresholding.

```
while (1)
  grab_image (Image, FGHandle)
  threshold (Image, CurrRegions, 0, 80)
```

The current regions are then merged with ones extracted in the previous image using the operator merge_regions_line_scan. As a result, two sets of regions are returned: The parameter

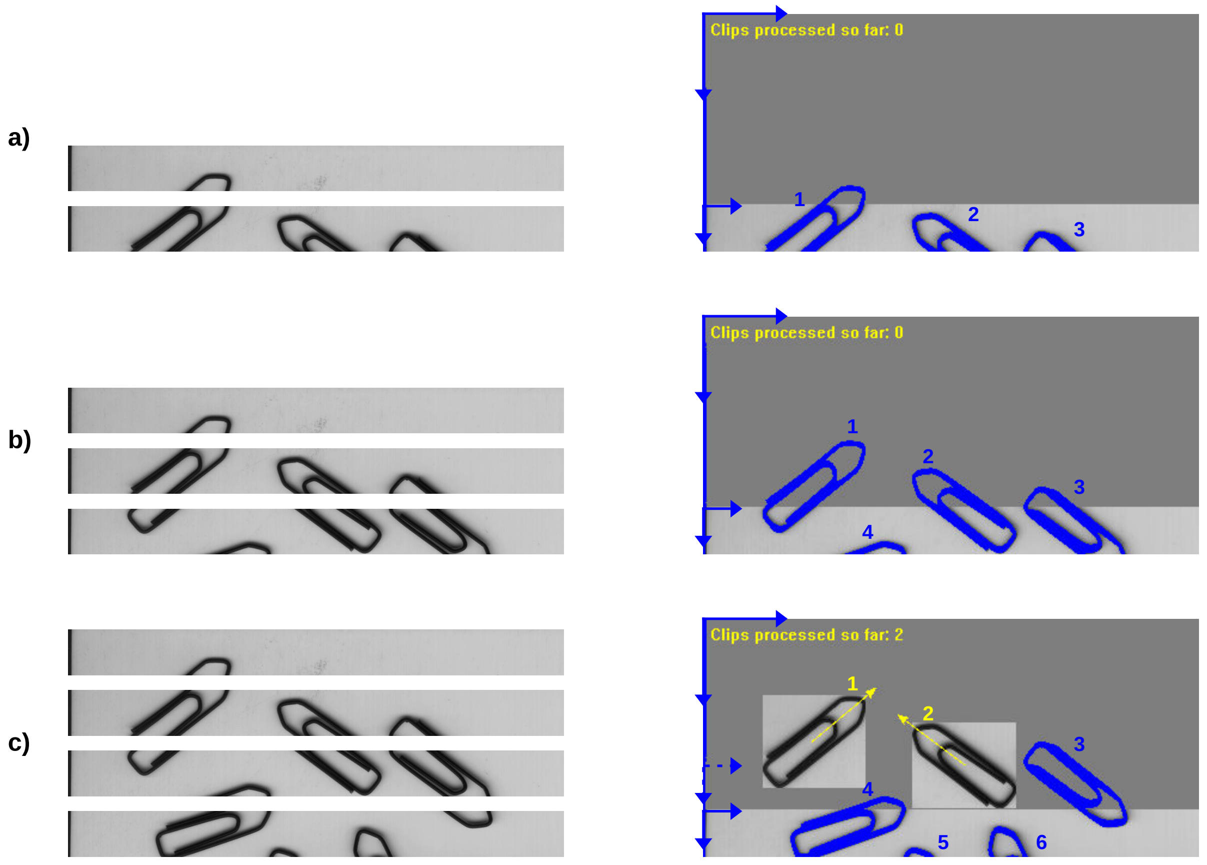

a)
b)
c)
Clips processed so far: 0
Clips processed so far: 0
Clips processed so far: 2
1
2
3
4
5
6

Figure 40: Merging regions extracted from subsequent line scan images: state after a) 2, b) 3, c) 4 images (large coordinate system: tiled image; small coordinate systems: current image or most recent image).

CurrMergedRegions contains the current regions, possibly extended by fitting parts of the previously extracted regions, whereas the parameter PrevMergedRegions contains the rest of the previous regions.

```
    merge_regions_line_scan (CurrRegions, PrevRegions, CurrMergedRegions,
                             PrevMergedRegions, ImageHeight, 'top',
                             MaxImagesRegions)
    connection (PrevMergedRegions, ClipCandidates)
    select_shape (ClipCandidates, FinishedClips, 'area', 'and', 4500, 7000)
```

The regions in PrevMergedRegions are “finished”; from them, the program selects the clips via their area further processes them later, e.g., determines their position and orientation. The regions in CurrMergedRegions are renamed and now form the previous regions for the next iteration.

```
    copy_obj (CurrMergedRegions, PrevRegions, 1, -1)
endwhile
```

Note that the operator copy_obj does not copy the regions themselves but only the corresponding HALCON objects, which can be thought of as references to the actual region data.

Before we show how to merge the images let’s take a look at figure 40, which visualizes the whole process: After the first two images CurrMergedRegions contains three clip parts; for the first one a previously extracted region was merged. Note that the regions are described in the coordinate frame of the current image; this means that the merged part of clip no. 1 has negative coordinates.

In the next iteration (figure 40b), further clip parts are merged, but no clip is finished yet. Note that the coordinate frame is again fixed to the current image; as a consequence the currently merged regions seem to move into negative coordinates.

After the fourth image (figure 40c), clips no. 1 and 2 are completed; they are returned in the parameter PrevMergedRegions. Note that they are still described in the coordinate frame of the previous image (depicted with dashed arrow); to visualize them together with CurrMergedRegions they must be moved to the coordinate system of the current image using the operator move_region:

```
  move_region (FinishedClips, ClipsInCurrentImageCoordinates,
               -ImageHeight, 0)
```

Let's get back to the task of merging images: To access the gray values around a clip, one must merge those images over which the PrevMergedRegions can be spread. At the beginning, therefore an empty image is created which can hold 4 images:

```
gen_image_const (TiledImage, 'byte', ImageWidth,
                 ImageHeight * MaxImagesRegions)
```

At the end of each iteration, the "oldest" image, i.e., the image at the top, is cut off from the tiled image using crop_part, and the current image is merged at the bottom using tile_images_offset:

```
  crop_part (TiledImage, TiledImageMinusOldest, ImageHeight, 0,
             ImageWidth, (MaxImagesRegions - 1) * ImageHeight)
  ImagesToTile := [TiledImageMinusOldest,Image]
  tile_images_offset (ImagesToTile, TiledImage, [0,
                     (MaxImagesRegions-1)*ImageHeight], [0, 0], [-1,
                     -1], [-1, -1], [-1, -1], [-1, -1], ImageWidth,
                     MaxImagesRegions * ImageHeight)
```

As noted above, the regions returned in PrevMergedRegions are described in the coordinate frame of the most recent image (depicted with dashed arrow in figure 40c); to extract the corresponding gray values from the tiled image, they must first be moved to its coordinate system (depicted with longer arrows) using the operator move_region. Then, the surrounding rectangles are created using shape_trans, and finally the corresponding gray values are extracted using add_channels:

```
  move_region (FinishedClips, ClipsInTiledImageCoordinates,
               (MaxImagesRegions-1) *  ImageHeight, 0)
  shape_trans (ClipsInTiledImageCoordinates, AroundClips, 'rectangle1')
  add_channels (AroundClips, TiledImage, GrayValuesAroundClips)
```

Appendix

A HALCON Images

In the following, we take a closer look at the way HALCON represents and handles images. Of course, we won't bother you with details about the low-level representation and the memory management; HALCON takes care of it in a way to guarantee optimal performance.

A.1 The Philosophy of HALCON Images

There are three important concepts behind HALCON's image objects:

1. **Multiple channels**

   Typically, one thinks of an image as a matrix of pixels. In HALCON, this matrix is called a *channel*, and images may consist of one or more such channels. For example, gray value images consist of a single channel, color images of three channels.

   The advantage of this representation is that many HALCON operators automatically process all channels at once; for example, if you want to subtract gray level or color images from another, you can apply sub_image without worrying about the image type. Whether an operator processes all channels at once can be seen in the parameter description in the reference manual: If an image parameter is described as (multichannel-)image or (multichannel-)image(-array) (e.g., the parameter ImageMinuend of sub_image), all channels are processed; if it is described as image or image(-array) (e.g., the parameter Image of threshold), only the first channel is processed.

   For more information about channels please refer to appendix A.3.2.

2. **Various pixel types**

   Besides the standard 8 bit (type byte) used to represent gray value image, HALCON allows images to contain various other data, e.g. 16 bit integers (type int2 or uint2) or 32 bit floating point numbers (type real) to represent derivatives.

   Most of the time you need not worry about pixel types, because HALCON operators that output images automatically use a suitable pixel type. For example, the operator derivate_gauss creates a real image to store the result of the derivation. As another example, if you connect to a frame grabber selecting a value $>$ 8 for the parameter BitsPerChannel, a subsequent grab_image returns an uint2 image.

3. **Arbitrarily-shaped region of interest**

   Besides the pixel information, each HALCON image also stores its so-called *domain* in form of a HALCON region. The domain can be interpreted as a region of interest, i.e., HALCON operators (with some exceptions) restrict their processing to this region.

   The image domain inherits the full flexibility of a HALCON region, i.e., it can be of arbitrary shape and size, can have holes, or even consist of unconnected points. For more information about domains please refer to appendix A.3.3.

The power of HALCON's approach lies in the fact that it offers full flexibility but does not require you to worry about options you don't need at the moment. For example, if all you do is grab and process standard 8 bit gray value images, you can ignore channels and pixel types. At the moment you decide to use color images instead, all you need to do is to add some lines to decompose the image into its channels. And if your camera / frame grabber provides images with more than 8 bit pixel information, HALCON is ready for this as well.

A.2 Image Tuples (Arrays)

Another powerful mechanism of HALCON is the so-called *tuple processing*: If you want to process multiple images in the same way, e.g., to smooth them, you can call the operator (e.g., `mean_image`) once passing it all images as a tuple (array), instead of calling it multiple times. Furthermore, some operators always return image tuples, e.g., `gen_gauss_pyramid` or `inspect_shape_model`.

Whether an operator supports tuple processing can be seen in the parameter description in the reference manual: If an input image parameter is described as `image(-array)` or `(multichannel-)image(-array)` (e.g., the parameter `Image` of `mean_image`), it supports tuple processing; if it is described as `image` or `(multichannel-)image` (e.g., the parameter `Image` of `find_1d_bar_code`), only one image is processed.

For information about creating or accessing image tuples please refer to appendix A.3.6.

A.3 HALCON Operators for Handling Images

Below you find a brief overview of operators that allow to create HALCON images or to modify "technical aspects" like the image size or the number of channels.

A.3.1 Creation

HALCON images are created automatically when you use operators like `grab_image` or `read_image`. You can also create images from scratch using the operators listed in the HDevelop menu `Operators` ▷ `Image` ▷ `Creation`, e.g., `gen_image_const` or `gen_image1_extern` (see also section 6.1).

A.3.2 Channels

Operators for manipulating channels can be found in the HDevelop menu `Operators` ▷ `Image` ▷ `Channel`. You can query the number of channels of an image with the operator `count_channels`. Channels can be accessed using `access_channel` (which extracts a specified channel without copying), `image_to_channels` (which converts a multi-channel image into an image tuple), or `decompose2` etc. (which converts a multi-channel image into 2 or more single-channel images). Vice versa, you can create a multi-channel image using `channels_to_image` or `compose2` etc., and add channels to an image using `append_channel`.

A.3.3 Domain

Operators for manipulating the domain of an image can be found in the HDevelop menu `Operators ▷ Image ▷ Domain`. Upon creation of an image, its domain is set to the full image size. You can set it to a specified region using `change_domain`. In contrast, the operator `reduce_domain` takes the original domain into account; the new domain is equal to the intersection of the original domain with the specified region. Please also take a look at the operator `add_channels`, which can be seen as complementary to `reduce_domain`.

A.3.4 Access

Operators for accessing infomation about a HALCON image can be found in the HDevelop menu `Operators ▷ Image ▷ Access`. For example, `get_image_pointer1` returns the size of an image and a pointer to the image matrix of its first channel.

A.3.5 Manipulation

You can change the size of an image using the operators `change_format` or `crop_part`, or other operators from the HDevelop menu `Operators ▷ Image ▷ Format`. The menu `Operators ▷ Image ▷ Type-Conversion` lists operators which change the pixel type, e.g., `convert_image_type`. Operators to modify the pixel values, can be found in the menu `Operators ▷ Image ▷ Manipulation`, e.g., `paint_gray`, which copies pixels from one image into another.

A.3.6 Image Tuples

Operators for creating and accessing image tuples can be found in the HDevelop menu `Operators ▷ Object ▷ Manipulation`. Image tuples can be created using the operators `gen_empty_obj` and `concat_obj`, while the operator `select_obj` allows to access an individual image that is part of a tuple.

B Parameters Describing the Image

When opening a connection with open_framegrabber, you can specify the desired image format, e.g., ist size or the number of bits per pixel, using 9 parameters, which are described in the following.

B.1 Image Size

The following 6 parameters influence the size of the grabbed images: HorizontalResolution and VerticalResolution specify the *spatial resolution* of the image in relation to the original size. For example, if you choose VerticalResolution = 2, you get an image with half the height of the original as depicted in figure 41b. Another name for this process is (vertical and horizontal) *subsampling*.

With the parameters ImageWidth, ImageHeight, StartRow, and StartColumn you can grab only a part of the (possibly subsampled) image; this is also called *image cropping*. In figure 41, image part to be grabbed is marked with a rectangle in the original (or subsampled) image; to the right, the resulting image is depicted. Note that the resulting HALCON image always starts with the coordinates (0,0), i.e., the information contained in the parameters StartRow and StartColumn cannot be recovered from the resulting image.

Depending on the involved components, both subsampling and image cropping may be executed at different points during the transfer of an image from the camera into HALCON: in the camera, in the frame grabber, or in the software. Please note that in most cases you get no direct effect on the performance in form of a higher frame rate; exceptions are CMOS cameras which adapt their frame rate to the requested image size. Subsampling or cropping on the software side has no effect on the frame rate; besides, you can achieve a similar result using reduce_domain. If the frame grabber executes the subsampling or cropping you may get a positive effect if the PCI bus is the bottleneck of your application and prevents you from getting the

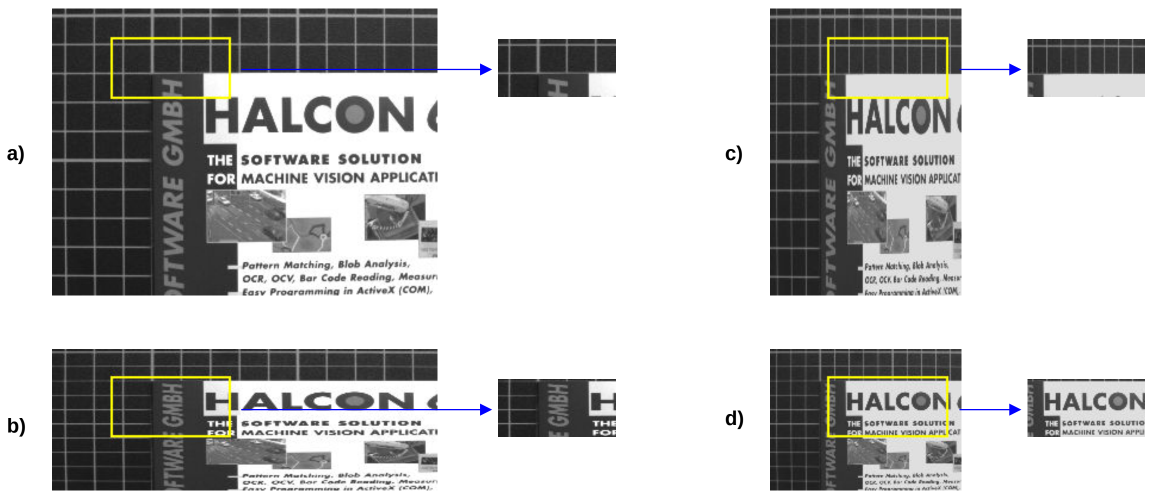


Figure 41: The effect of image resolution (subsampling) and image cropping (ImageWidth = 200, ImageHeight = 100, StartRow = 50, StartColumn = 100): a) HorizontalResolution (HR) = VerticalResolution (VR) = 1; b) HR = 1, VR = 2; c) HR = 2, VR = 1; d) HR = VR = 2.

full frame rate. Some frame grabber interfaces allow *dynamic image cropping* via the operator set_framegrabber_param.

Note that HALCON itself does not differentiate between area and line scan cameras as both produce images – the former in form of frames, the latter in form of so-called *pages* created from successive lines (number specified in the parameter ImageHeight). Section 6.3 contains additional information regarding the use of line scan cameras.

B.2 Frames vs. Fields

The parameter Field is relevant only for analog cameras that produce signals following the video standards originally developed for TV, e.g., NTSC or PAL. In these standards, the camera transmits images (also called *frames*) in form of two so-called *fields*, one containing all odd lines of a frame, the other all even lines of the next frame. On the frame grabber board, these two fields are then *interlaced*; the resulting frame is transferred via the PCI bus into the computer memory using DMA (*direct memory access*).

Figure 42 visualizes this process and demonstrates its major draw back: If a moving object is observed (in the example a dark square with the letter 'T'), the position of the object changes from field to field, the resulting frame shows a distortion at the vertical object boundaries (also called *picket-fence effect*). Such a distortion seriously impairs the accuracy of measurements; industrial vision systems therefore often use so-called *progressive scan* cameras which transfer full frames (see figure 43). Some cameras also "mix" interlacing with progressive scan as depicted in figure 44.

You can also acquire the individual fields by specifying VerticalResolution = 2. Via the parameter Field you can then select which fields are to be acquired (see also figure 45): If you select 'first' or 'second', all you get odd or all even fields, respectively; if you select 'next', you get every field. The latter mode has the advantage of a higher field rate, at the

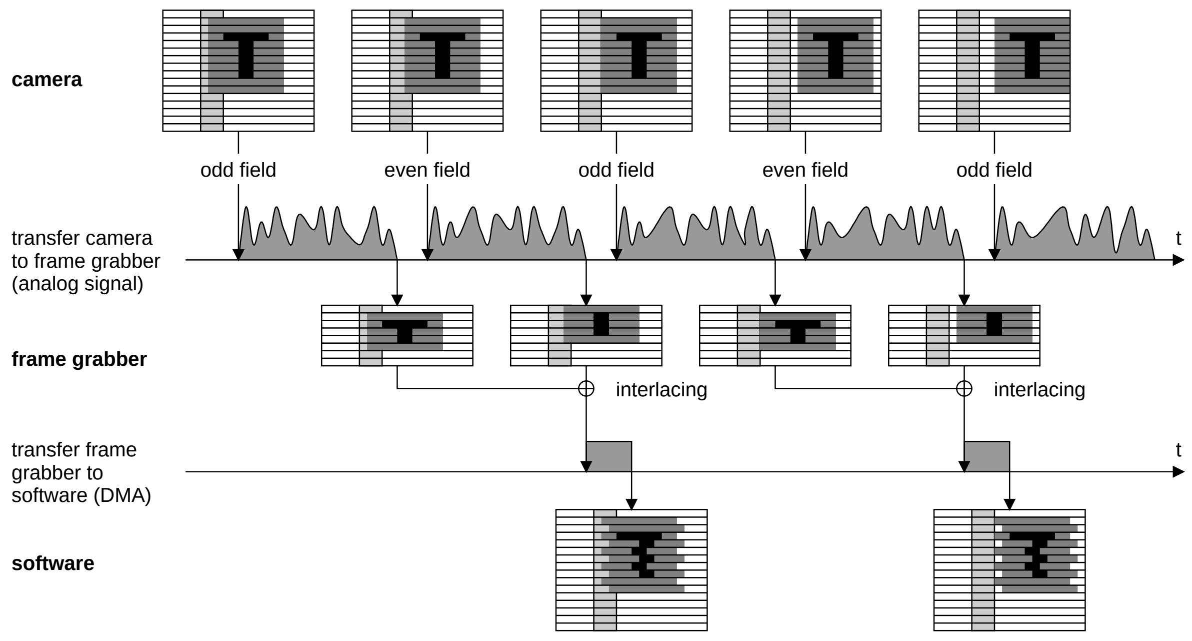


Figure 42: Interlaced grabbing (Field = 'interlaced').

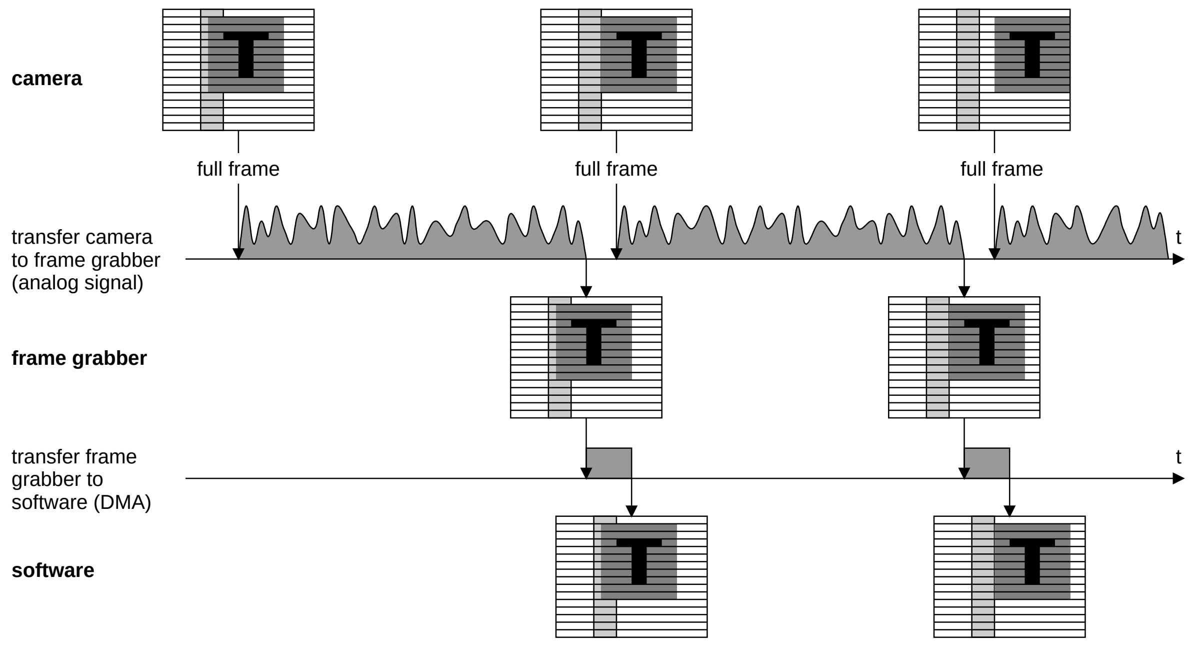

camera
full frame
full frame
full frame
transfer camera to frame grabber (analog signal)
t
frame grabber
transfer frame grabber to software (DMA)
t
software

Figure 43: Progressive scan grabbing (`Field` = `'progressive'`).

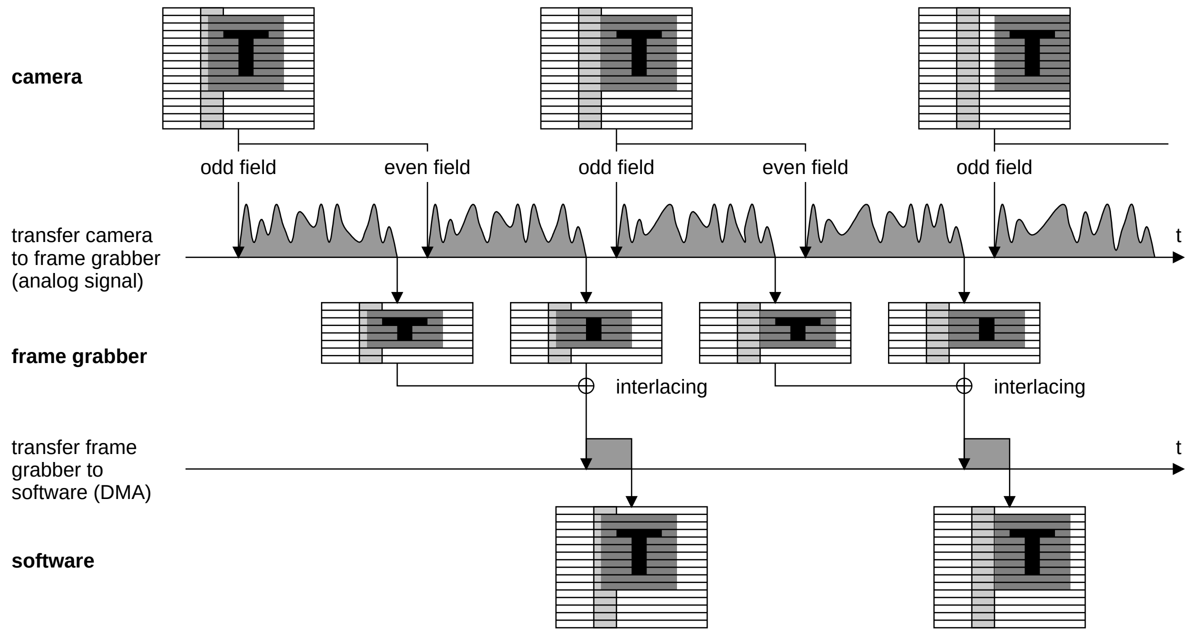


Figure 44: Special form of interlaced grabbing supported by some cameras.

cost, however, of the so-called *vertical jitter*: Objects may seem to move up and down (like the square in figure 45), while structures that are one pixel wide appear and disappear (like the upper part of the 'T').

By specifying `Field` = `'first'`, `'second'`, or `'next'` for a full resolution image (`VerticalResolution` = 1), you can select with which field the interlacing starts.

Figure 46 shows a timing diagram for using `grab_image` together with an interlaced-scan camera. Here, you can in some cases increase the processing frame rate by specifying `'next'` for the parameter `Field`. The frame grabber then starts to digitize an image when the next field

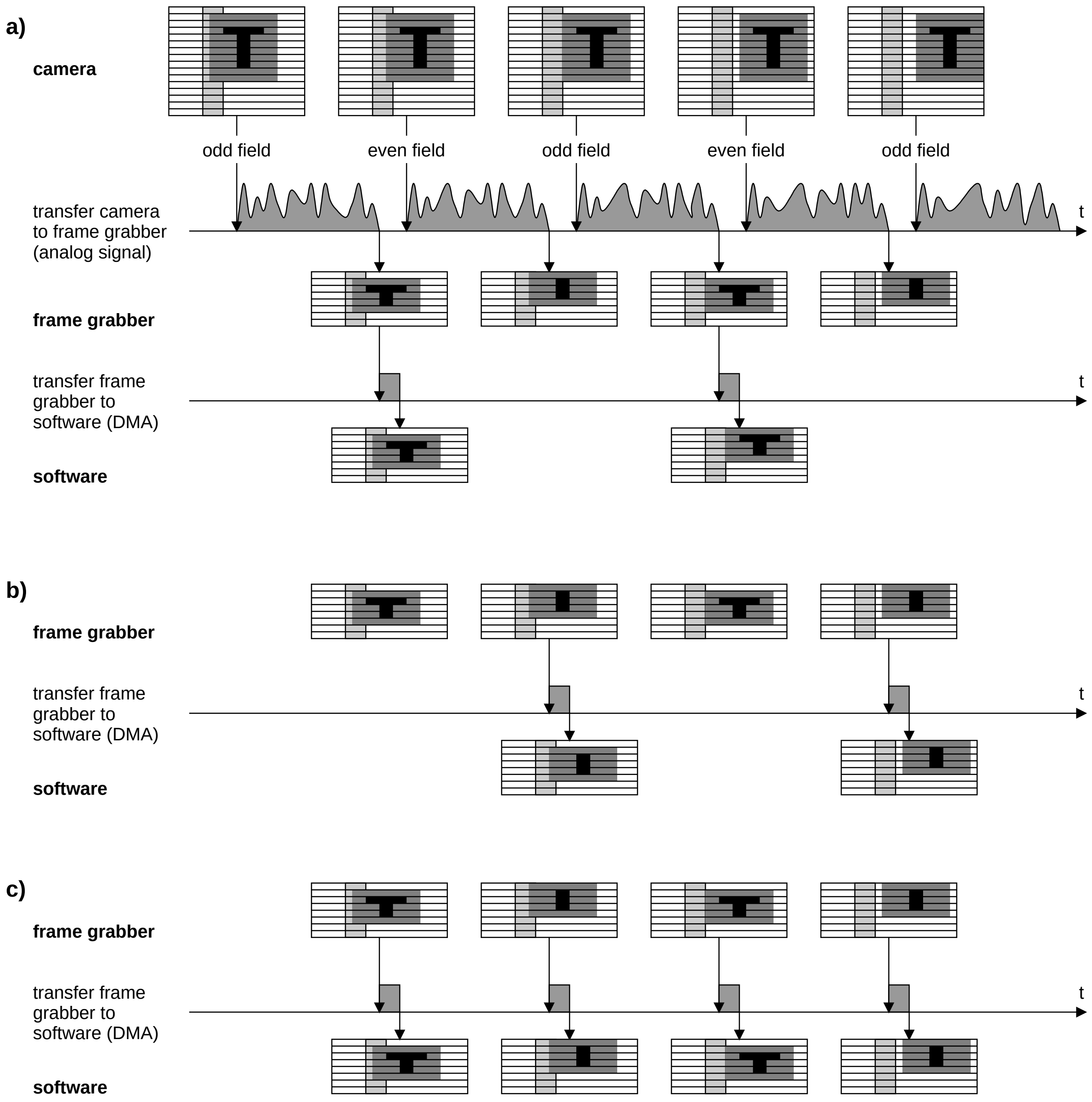

a)
camera
odd field
even field
odd field
even field
odd field
transfer camera to frame grabber (analog signal)
t
frame grabber
transfer frame grabber to software (DMA)
t
software
b)
frame grabber
transfer frame grabber to software (DMA)
t
software
c)
frame grabber
transfer frame grabber to software (DMA)
t
software

Figure 45: Three ways of field grabbing: a) `'first'`; b) `'second'`; c) `'next'` .

arrives; in the example therefore only one field is lost.

B.3 Image Data

The parameters described in the previous sections concentrated on the size of the images. The *image data*, i.e., the data contained in a pixel, is described with the parameters `BitsPerChannel` and `ColorSpace`. To understand these parameters, a quick look at the HALCON's way to represent images is necessary: A HALCON image consists of one or more matrices of pixels, which are called *channels*. Gray value images are represented as single-channel images, while color images consist of three channels, e.g., for the red, green, and blue part of an RGB image. Each image matrix (channel) consists of *pixels*, which may be of different *data types*, e.g., standard 8 bit (type `byte`) or 16 bit integers (type `int2` or `uint2`) or 32 bit floating

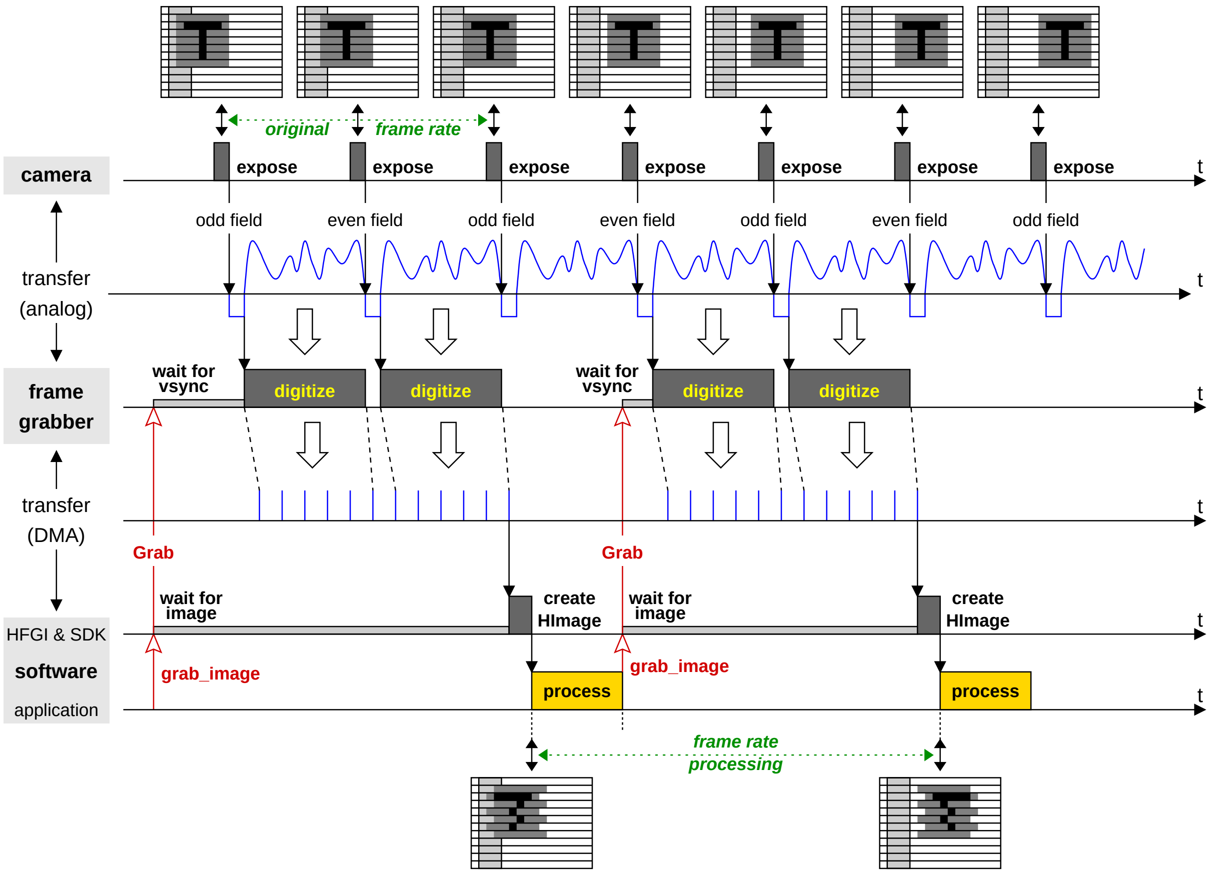


Figure 46: Grabbing interlaced images starting with the `'next'` field.

point numbers (type `real`). For detailed information about HALCON images please refer to appendix A.

The two parameters correspond to the two main aspects of HALCON's images: With the parameter `ColorSpace` you can select whether the resulting HALCON image is to be a (single-channel) gray value image (value `'gray'`) or a (multi-channel) color image (e.g., value `'rgb'`). The parameter `BitsPerChannel` specifies how many bits are *transmitted* per pixel per channel from the frame grabber to the computer; the pixel type of the HALCON image is then chosen to accommodate the transmitted number of pixels.

For example, if a frame grabber is able to transmit 10 bit gray value images, one selects `ColorSpace` = `'gray'` and `BitsPerChannel` = `10` and gets a single-channel HALCON image of the type `'uint2'`, i.e., 16 bit per channel. Another example concerns RGB images: Some frame grabbers allow the values `8` and `5` for `BitsPerChannel`. In the first case, $3 \times 8 = 24$ bit are transmitted per pixel, while in the second case only $3 \times 5 = 15$ (padded to 16) bit are transmitted; in both cases, a three-channel `'byte'` image results.