HALCON

a product of MVTec

Solution Guide II-B

Matching



HALCON 21.05 *Progress*

MVTec Software GmbH

Building Vision for Business

How to Use Matching to Find and Localize Objects, Version 21.05.0.0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Copyright © 2010-2021 by MVTec Software GmbH, München, Germany



Protected by the following patents: US 7,062,093, US 7,239,929, US 7,751,625, US 7,953,290, US 7,953,291, US 8,260,059, US 8,379,014, US 8,830,229. Further patents pending.

Microsoft, Windows, Windows Server 2008/2012/2012 R2/2016, Windows 7/8/8.1/10, Microsoft .NET, Visual C++, and Visual Basic are either trademarks or registered trademarks of Microsoft Corporation.

All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

More information about HALCON can be found at: http://www.halcon.com/

About This Manual

In a broad range of applications matching is suitable to find and locate objects in images. This Solution Guide leads you through the variety of approaches that are provided by HALCON.

An introduction to the available matching approaches, including tips for the selection of a specific approach for a specific application, is given in section 1 on page 7.

Some general topics that are valid for multiple approaches are discussed in section 2 on page 17. These comprise, e.g., the selection of a proper template, tips for a speed-up, and the use of the results.

Section 3 on page 47 then provides you with detailed information about the individual matching approaches.

The HDevelop example programs that are presented in this Solution Guide can be found in the specified subdirectories of the directory `%HALCONEXAMPLES%`.

Contents

1 Introduction **7**
- 1.1 How to Use This Manual? . . . 7
- 1.2 What is Matching? . . . 7
- 1.3 How to Generally Apply a Matching? . . . 8
- 1.4 Which Approaches are Available? . . . 8
- 1.5 Which Approach is Suitable in Which Situation? . . . 9
  - 1.5.1 The Matching Approaches: 2D versus 3D . . . 9
  - 1.5.2 Decisions for 3D Objects and 2D Objects in 3D Space . . . 10
  - 1.5.3 First Decisions for Orthogonally Imaged 2D Objects . . . 11
  - 1.5.4 Shape-Based vs. Correlation-Based Matching . . . 12
  - 1.5.5 Quick Guide to the Matching Approaches . . . 13

2 General Topics **17**
- 2.1 Prepare the Template . . . 17
  - 2.1.1 Reduce the Reference Image to a Template Image . . . 17
  - 2.1.2 Influence of the Region of Interest . . . 18
  - 2.1.3 Synthetic Models as Alternatives to Template Images . . . 20
- 2.2 Reuse the Model . . . 24
- 2.3 Speed Up the Search . . . 25
  - 2.3.1 Restrict the Search Space . . . 25
  - 2.3.2 About Subsampling . . . 25
- 2.4 Use the Results of Matching . . . 28
  - 2.4.1 Results of the Individual Matching Approaches . . . 28
  - 2.4.2 About Transformations . . . 29
  - 2.4.3 Use the Estimated 2D Position and Orientation . . . 32
  - 2.4.4 Use the Estimated 2D Scale . . . 41
  - 2.4.5 Use the Estimated 2D Homography . . . 42
  - 2.4.6 Use the Estimated 3D Pose . . . 45
  - 2.4.7 About the Score . . . 46

3 The Individual Approaches **47**
- 3.1 Correlation-Based Matching . . . 47
  - 3.1.1 A First Example . . . 47
  - 3.1.2 Select the Model ROI . . . 48
  - 3.1.3 Create a Suitable NCC Model . . . 49
  - 3.1.4 Optimize the Search Process . . . 50
- 3.2 Shape-Based Matching . . . 52
  - 3.2.1 A First Example . . . 52
  - 3.2.2 Select the Model ROI . . . 54
  - 3.2.3 Create a Suitable Shape Model . . . 55
  - 3.2.4 Optimize the Search Process . . . 61
  - 3.2.5 Use the Specific Results of Shape-Based Matching . . . 71
  - 3.2.6 Adapt to a Changed Camera Orientation . . . 74
- 3.3 Component-Based Matching . . . 74
  - 3.3.1 A First Example . . . 74
  - 3.3.2 Extract the Initial Components . . . 76
  - 3.3.3 Create a Suitable Component Model . . . 78
  - 3.3.4 Search for Model Instances . . . 85

3.3.5 Use the Specific Results of Component-Based Matching . . . 88
3.4 Local Deformable Matching . . . 89
3.4.1 A First Example . . . 90
3.4.2 Select the Model ROI . . . 92
3.4.3 Create a Suitable Local Deformable Model . . . 93
3.4.4 Optimize the Search Process . . . 95
3.4.5 Use the Specific Results of Local Deformable Matching . . . 97
3.5 Perspective Deformable Matching . . . 99
3.5.1 A First Example . . . 99
3.5.2 Select the Model ROI . . . 101
3.5.3 Create a Suitable Perspective Deformable Model . . . 101
3.5.4 Optimize the Search Process . . . 104
3.5.5 Use the Specific Results of Perspective Deformable Matching . . . 107
3.6 Descriptor-Based Matching . . . 107
3.6.1 A First Example . . . 108
3.6.2 Select the Model ROI . . . 110
3.6.3 Create a Suitable Descriptor Model . . . 110
3.6.4 Optimize the Search Process . . . 112
3.6.5 Use the Specific Results of Descriptor-Based Matching . . . 115

Index **117**

Chapter 1

Introduction

This section introduces you to HALCON's matching functionality. In particular, it provides you with an overview of

- how to use this manual (section 1.1),
- the general meaning of matching (section 1.2),
- how to generally apply a matching (section 1.3),
- the available approaches (section 1.4), and
- information about which approaches are suitable in which situation (section 1.5 on page 9).

1.1 How to Use This Manual?

If you have no or only little experience with matching applications using HALCON, the following subsections introduce you to matching with HALCON in general and guide you through the most salient differences between the available matching approaches. Thus, they help you to select the matching approach that is best suited for your specific application. Further, we recommend to read section 2 on page 17 before you step into the subsection of section 3 on page 47 that in detail describes your selected matching approach.

If you are an experienced HALCON user that is familiar with matching and you are looking for further tips on how to optimize your specific application, it may be sufficient to immediately step into the subsection of section 3 on page 47 that describes the matching approach of your choice and possibly have an additional look on selected subsections of section 2 on page 17, which describe some general topics that are needed for multiple matching approaches.

1.2 What is Matching?

With matching, HALCON provides a method for the robust location of objects in images, which can be used for many different applications. To suit the different requirements of the applications, different approaches are available. All approaches consist of a small set of operators for which only a few parameters have to be adjusted. Further, none of the approaches requires an explicit segmentation of the objects in the images. Thus, you can successfully locate your objects even if you have no special knowledge in machine vision.

The main idea of matching is to use a prototype object (template), create a model of it and search the model in other images. For most matching tasks you obtain the model from a **reference image** that shows the object of interest. To suppress other structures or objects that are contained in this image, the image is reduced to a region of interest (ROI) that only contains the object and which may have an arbitrary shape. The reduced image is the **template image** from which the model is created by an approach-specific operator. Another approach-specific operator uses the obtained model to find the object in different search images. That is, it searches for image structures that match the model (within small tolerances).

The different matching approaches provided by HALCON differ amongst others in the image structures that are used to build the model. Some matching approaches use, e.g., the relations of gray values to their surrounding (neighborhood) to build a model. Others use, e.g., the shapes of contours (see figure 1.1). The result of the matching is in each case information about the position, for most approaches the orientation, and for some approaches also the scale of the found object in the search image.

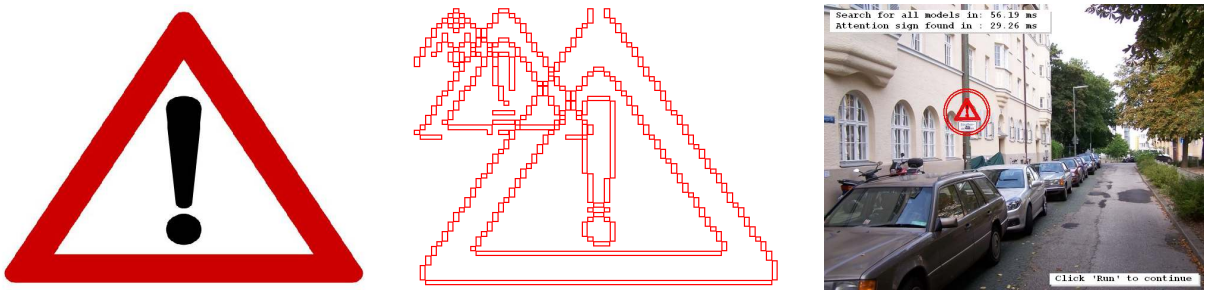


Figure 1.1: Matching using shapes of contours: a) the image of an attention sign is used to build a b) (shape) model in different resolutions (pyramid levels), which is used to c) locate an instance of the model in an image.

1.3 How to Generally Apply a Matching?

Although different matching approaches use different sets of operators, the main proceeding is similar for all and consists of the following main steps:

1. **Create a model** for the object of interest.

   A model of an object, i.e., the internal data structure describing a searched object, is created using either a representative reference image of it or a synthetic model. Which way to follow depends on the selected approach and the available data.

2. **Find the model** in search images.

   Instances of the previously created model are searched in images and their position, in most cases their orientation, and for some approaches also their scale are returned.

Additional to these essential steps, most approaches provide means to **modify** a model, **reuse** it (i.e., store it to file and read it from file), and to **query information** from it.

1.4 Which Approaches are Available?

The matching approaches described in this Solution Guide comprise

- an approach that describes a model by the gray-value relations of the contained pixels:
  - the **correlation-based matching** (section 3.1 on page 47), which uses normalized cross correlation (NCC) to match objects or patterns, respectively.
- approaches that describe a model by the shapes of its contours:
  - the **shape-based matching** (section 3.2 on page 52),
  - the **component-based matching** (section 3.3 on page 74), which is designed for the specific case that several components (rigid parts) of an object move relative to each other,
  - the **local deformable matching** (section 3.4 on page 89), which can handle and return local deformations of the object and allows to rectify the image part containing the deformed model, and

  - the **perspective deformable matching** (section 3.5 on page 99), which can handle also perspective distortions and provides a calibrated version with which also a 3D pose instead of 2D transformation parameters can be derived.

- an approach that describes a model by a set of significant image points:
  - the **descriptor-based matching** (section 3.6 on page 107) matches a set of so-called interest points. Similar to the perspective deformable matching it can handle perspective distortions and provides a calibrated version with which also a 3D pose instead of 2D transformation parameters can be derived.

Additionally, the so-called point-based matching and the 3D matching are available. For the **point-based matching**, corresponding points are used to combine overlapping images. This method is also called uncalibrated mosaicking. In the broader sense, the search for corresponding points is also a kind of matching, but the focus of this Solution Guide is on matching of "2D objects". The **3D matching** consists of different methods and is shortly introduced in the following section, but is also not subject to this Solution Guide. Please refer to the Solution Guide I, chapter 11 on page 123 for further details on 3D matching.

1.5 Which Approach is Suitable in Which Situation?

The first step when selecting a matching approach is to decide if two or three spatial dimensions are needed. Afterwards, further criteria like the needed transformation parameters or the object's appearance within the images can be included into the decision. The following sections provide you with the background for your decisions. For a quick overview, this background is clearly illustrated in the figures of section 1.5.5 on page 13.

1.5.1 The Matching Approaches: 2D versus 3D

The different matching approaches are suitable for different applications. Some are suitable only for 2D objects imaged from an orthogonal view, others can also handle perspective distortions, and some approaches even match 3D shapes in a full 3D space (see figure 1.2). Note that with the term "2D object" a fixed view on a planar object part is meant and not the actual tangible object, which in reality is naturally three-dimensional. In contrast, the "3D object" is an object that is viewed from arbitrary directions.

- 2D objects, imaged from an orthogonal view:

  The **correlation-based, shape-based, component-based, and local deformable matching** can be used to find 2D objects in images. The objects must be taken from an orthogonal view for the reference image as well as for the search images. Theoretically, you can also use the perspective deformable or the descriptor-based matching to find orthogonally imaged 2D objects, but as these are not as fast as the strict 2D approaches, they are recommended rather for the case that the objects must be imaged from a perspective view.

- 2D objects, imaged from a perspective view:

  The **perspective deformable and the descriptor-based matching** can be used to find planar object parts, too. But in addition to the approaches used for the orthogonal view, the objects may be perspectively deformed. Additionally, if a camera calibration was applied, not only the 2D position and orientation but the 3D pose of the object can be derived. Although both approaches can be used also for orthogonally imaged 2D objects, they are recommended rather for the perspective case, as they are not as fast as the strict 2D approaches.

- 3D objects:

  The **3D matching** searches for "real" 3D objects in 2D images. Here, different approaches are available. For example, for the shape-based 3D matching, no reference image is used to create the model but a synthetic 3D model, in particular a DXF CAD model, must be provided. If a 3D object contains a characteristic planar part which is visible in all search images, alternatively the calibrated perspective 2D approaches, i.e., the **perspective deformable** or the **descriptor-based matching**, can be used, which are more convenient and significantly faster. The 3D matching is needed only if more than one planar part of the object is needed to differentiate the object from other image parts.

Note that the 3D matching is not part of this Solution Guide! For further information on 3D matching, please refer to the Solution Guide I, chapter 11 on page 123.



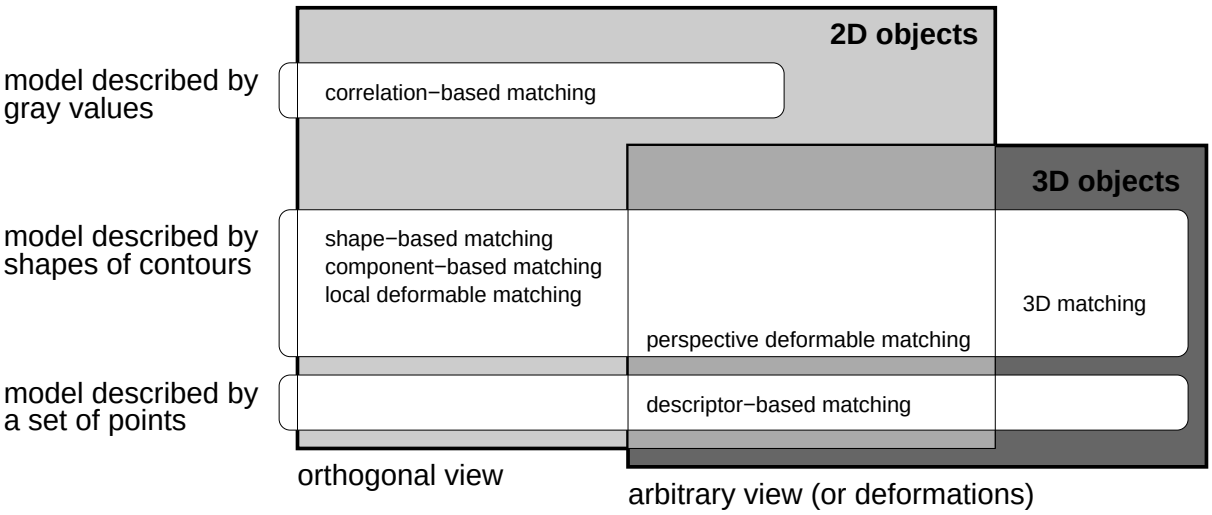


Figure 1.2: Available 2D and 3D matching approaches.

1.5.2 Decisions for 3D Objects and 2D Objects in 3D Space

If you search for a **complex 3D object** that differs from other objects in all three dimensions (see, e.g., figure 1.3), the selection of the appropriate matching approach is easy as you have to use the **3D matching**.

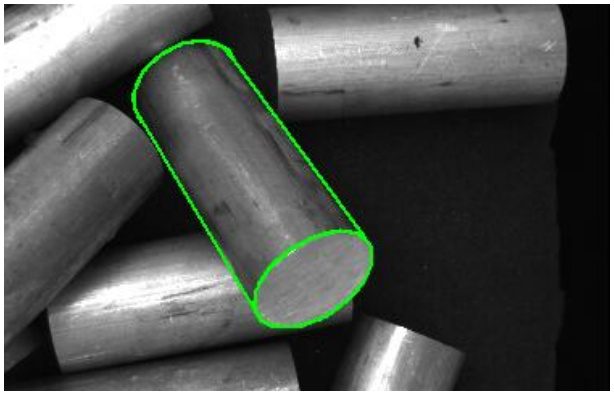


Figure 1.3: 3D object: the third dimension is needed to locate the object.

If the **3D object contains a unique but planar part** that significantly differs from the other structures in the expected images or if you search for a planar object that may be oriented arbitrarily in the 3D space you can also use the **perspective deformable** or the **descriptor-based matching**, which can handle perspective distortions of 2D objects. Both approaches are faster and more convenient to use than the 3D matching.

Which of both perspective approaches to select depends on the specific application. If the object of interest is expected to be anisotropically **scaled** in the search images, only the perspective deformable matching is suitable. If it is expected to be only translated and rotated, both approaches are suitable and the appearance of the object in the images must be taken into account. The most important difference between both approaches is the way the object is modeled. The perspective deformable matching describes the model by the shapes of the object's contours. Thus, it is suitable for objects that contain **clearly visible contours** (see, e.g., figure 1.4).

In contrast, the descriptor-based matching describes the model by interest points. Thus, for objects that are rather **characterized by an arbitrary but fixed texture** (see, e.g., figure 1.5), it is most probably to be preferred.

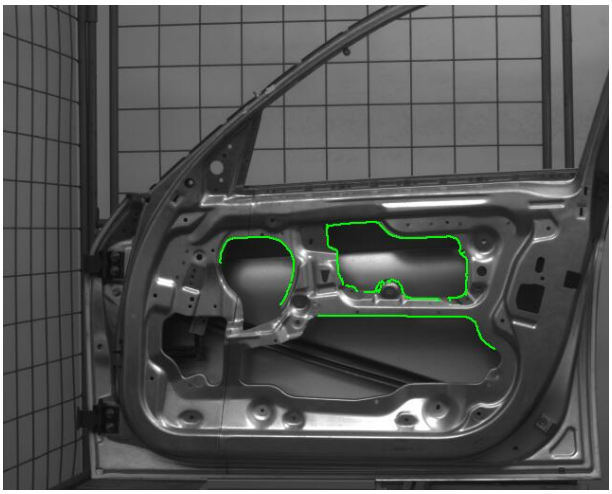


Figure 1.4: 3D object: unique parts of the object lie in a plane and can be described by clearly visible contours.



Figure 1.5: 3D object: a plane object part is characterized by a specific texture.

1.5.3 First Decisions for Orthogonally Imaged 2D Objects

If you can image planar objects from an orthogonal view and only the 2D transformation parameters of the found object instances are needed, the perspective deformable or the descriptor-based matching would lead to the desired result as well, but the approaches that are strictly restricted to two dimensions are to be preferred as they are significantly faster. The 2D approaches comprise the correlation-based, shape-based, component-based, and local deformable matching.

Three of theses approaches are used only in specific situations:

- The **component-based matching** is applied only if the object of interest consists of different **components that move relative to each other** (see section 3.3 on page 74 and figure 1.6) and only if the object is not expected to be scaled in the search images.
- The **local deformable matching** is applied only if the **object of interest is locally deformed**, e.g., because of a contorted surface (see section 3.4 on page 89 and figure 1.7). In contrast to the other 2D approaches, it returns only the position but no orientation for the found model instance. In exchange, it allows to rectify the image part containing the deformed object and returns a vector field that describes the actual deformations.

In most orthogonal 2D cases, you have to select one of the two remaining approaches, i.e., either the **shape-based matching** or the **correlation-based matching**.

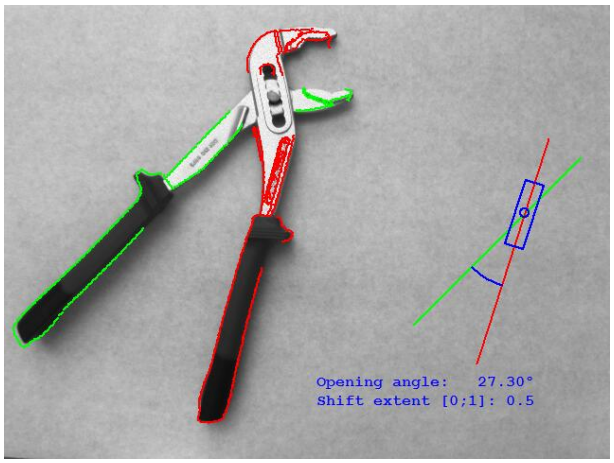


Figure 1.6: Object consisting of two components that move relative to each other.



Figure 1.7: Object with local deformations.

1.5.4 Shape-Based vs. Correlation-Based Matching

To decide if the shape-based or the correlation-based matching is more appropriate for your specific application, you should further investigate the requirements of your application.

For example, you should know which **transformation parameters** are needed to describe the relation of the object in the search image to the object described by the model. Will the object be only translated and rotated or also scaled? If a **scaling** is needed, correlation-based matching can not be used, but one of the shape-based matching approaches must be used. Here, you can further choose between approaches that use uniform scaling or anisotropic scaling, i.e., a scaling with different scaling factors for the x- and y-direction.

Additionally, the **appearance of the object** may change from image to image. Reasons can be, e.g., occlusions, clutter, or illumination changes that may lead to a changing polarity of the object. Furthermore, the images may be defocused or the object is a complex pattern in front of a complex background, i.e., it is textured. To get a robust and fast result, the changes of the object's appearance should be minimized as much as possible already when imaging the object. Nevertheless, sometimes distortions like occlusions, clutter, or defocus can not be avoided. These distortions have to be involved into the decision which approach to use for a specific application.

In particular, shape-based matching should be chosen if occlusions (see, e.g., figure 1.8) or clutter can not be avoided or if a matching of objects with changing color is applied.

In contrast, correlation-based matching is suitable for objects with a random and changing texture or for objects with a slightly changing shape (see figure 1.9). Additionally, correlation-based matching is to be preferred when handling strongly defocused images.



Figure 1.8: Occlusions can be handled by shape-based matching.

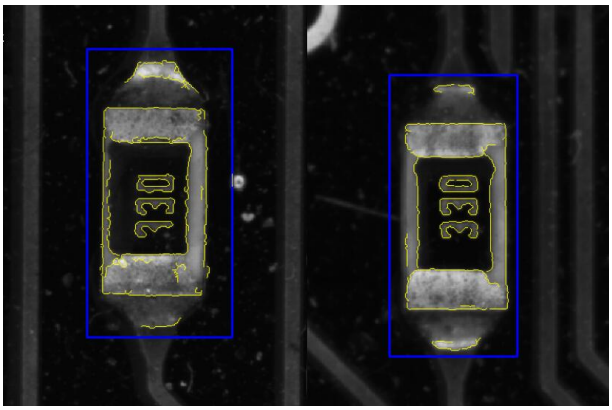


Figure 1.9: Changing shapes can be handled by correlation-based matching.

1.5.5 Quick Guide to the Matching Approaches

Figure 1.10 to figure 1.15 clearly summarize the information needed to decide which matching approach is suitable for a specific application. In particular,

- figure 1.10 summarizes the first coarse decision steps needed to select a matching approach,
- figure 1.11 and figure 1.12 illustrate the transformations that can be handled by the individual matching approaches,
- figure 1.13 lists the transformation parameters that are returned by the individual matching approaches, and
- figure 1.14 and figure 1.15 introduce typical changes of the appearance of objects and show which characteristics of the appearance can be handled by the individual matching approaches.

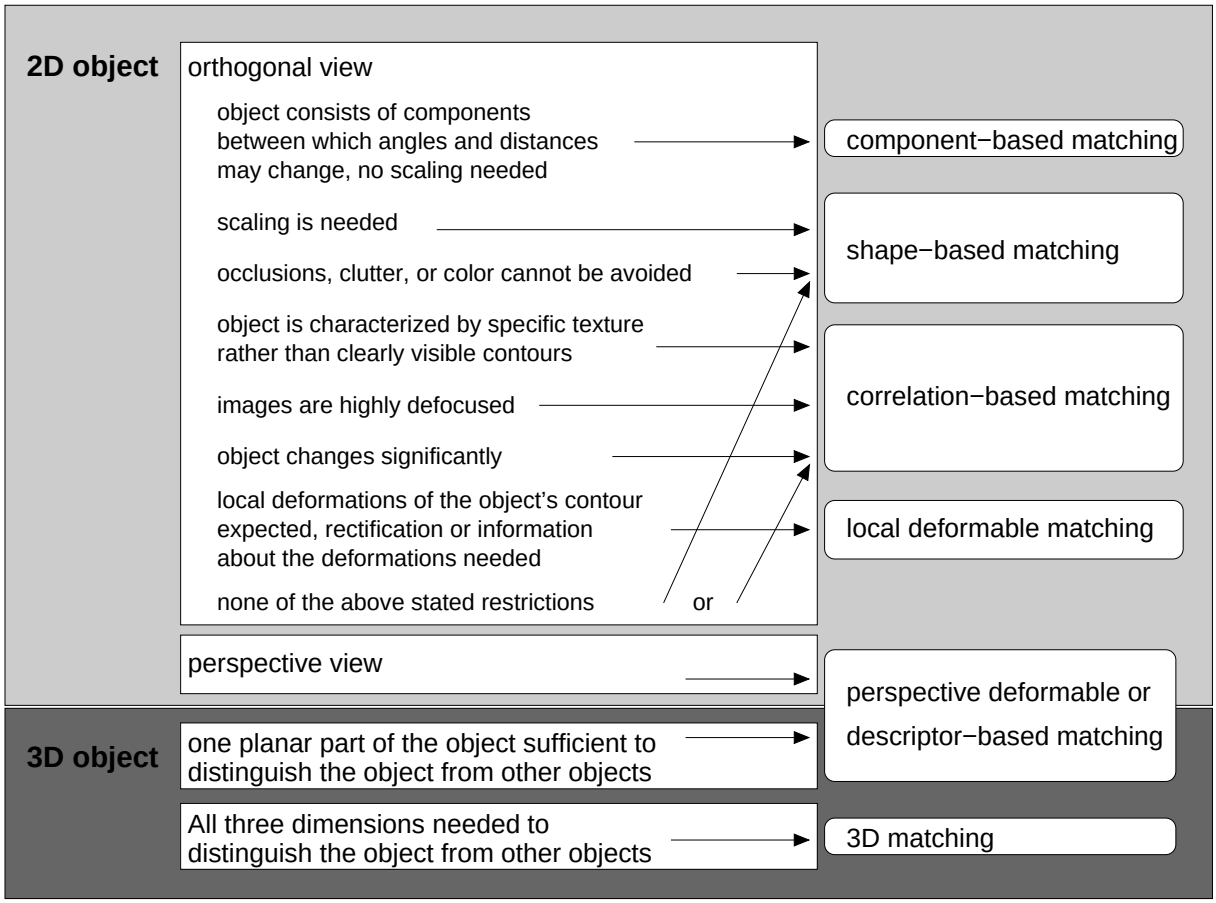


Figure 1.10: First decision steps when selecting the matching approach.

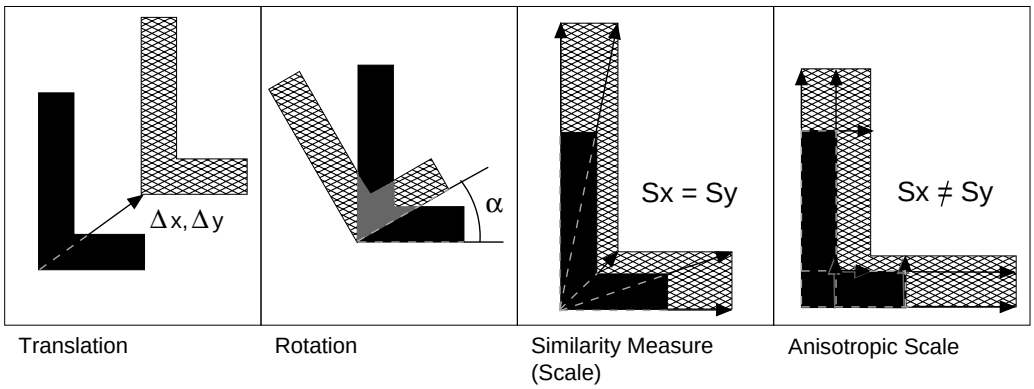


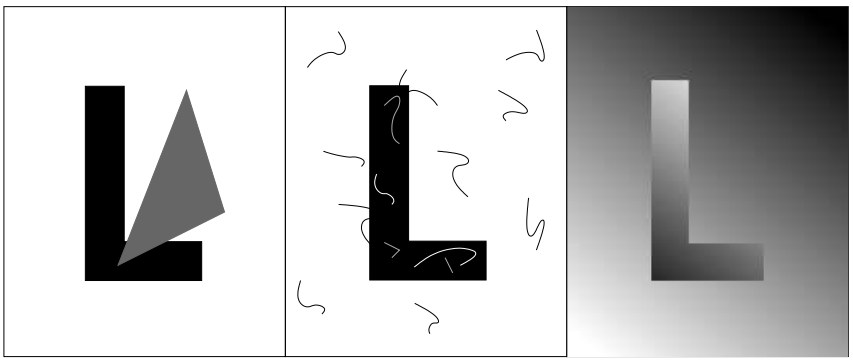
Figure 1.11: Geometric transformations of an object.

Matching Approach	Translation (2D)	Rotation (2D)	Scaling (uniform)	Scaling (anisotropic)
correlation-based	X	X	-	-
shape-based	X	X	X	X
component-based	X	X	-	-
local deformable	X	X	X	X
perspective deformable	X	X	X	X
descriptor-based	X	X	X	-

Figure 1.12: Transformations that can be handled by the matching approaches.

Matching Approach	Position (2D)	Angle (2D)	Scale (uniform)	Scale (anisotropic)	Projective Transformation Matrix	Pose (3D)
correlation-based	X	X	-	-	-	-
shape-based	X	X	X	X	-	-
component-based	X	X	-	-	-	-
local deformable	X	-	-	-	-	-
perspective deformable	-	-	-	-	X	X
descriptor-based	-	-	-	-	X	X

Figure 1.13: Transformation parameters returned by the matching approaches.



Occlusion (Missing Contours) | Clutter (Additional Contours) | Illumination Changes

Figure 1.14: The object's appearance may change, e.g., because of (from left to right): occlusions, clutter, and illumination changes.

Matching Approach	Occlusion	Clutter	Illumination changes	Texture	Color	Defocus
correlation-based	-	-	X (only linear)	X	-	X
shape-based	X	X	X	-	X	X
component-based	X	X	X	-	X	-
local deformable	X	X	X	-	X	-
perspective deformable	X	X	X	-	X	-
descriptor-based	X	X	X	X (needed)	-	-

Figure 1.15: Characteristics of the appearance the matching approaches can cope with.

Chapter 2

General Topics

A **general proceeding** for a matching is to prepare the reference image, e.g., by a preprocessing (if necessary), prepare the template, create the model from it, modify the already existing model (if necessary), store it to file (if it should be reused), query information from it (which is needed, e.g., when reusing a previously stored model), restrict the search space to speed up the following matching, search for the model in (possibly preprocessed) search images, and further process the result of the matching.

In the following, we discuss different **general topics** that are valid for many matching approaches at once. In particular,

- section 2.1 shows how to **prepare a template**,
- section 2.2 on page 24 shows how to **reuse a model**,
- section 2.3 on page 25 shows how to **speed up the search**, and
- section 2.4 on page 28 shows how to **further process the results** of matching.

2.1 Prepare the Template

After an optional preprocessing of the reference image, the first step of the matching is to prepare a template of the object of interest. From this, a model is derived, which in a later step is used to locate instances of the object in search images. In most cases, the model is derived from a reference image, which is reduced to a so-called template image. How to obtain this template image using a region of interest (ROI) is described in section 2.1.1. The influence of the ROI on the further matching process is described in section 2.1.2. For some matching approaches, a synthetic model can be used instead of the template image. This can be either a synthetically created template image or an XLD contour (see section 2.1.3 on page 20).

2.1.1 Reduce the Reference Image to a Template Image

To create a model from a reference image, which is the common way for most matching approaches, the reference image must be reduced to a template image that contains only those structures of the reference image that are needed to derive the model.

For this, you select a region within the reference image that shows the part of the image that should serve as the template or, respectively, from which the model should be derived. After selecting the region, the domain of the reference image is reduced to an ROI using the operator `reduce_domain`. The resulting image is our template image, which is input to one of the approach-specific operators that are provided for the actual model generation.

Note that a region and therefore also the model can have an arbitrary shape (see the Quick Guide, section 2.1.1 on page 10 for information to regions in HALCON). A region can be created in different ways, as is described in the Solution Guide I, chapter 3 on page 27 and chapter 4 on page 35. Summarized, a **region can be selected**, e.g., by the following means:

- **A region can be specified by explicitly defining its parameters**. That is, 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. To use the 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. If these parameters are not explicitly known, you can get them using draw operators, i.e., operators that let you draw a shape on the displayed image and then return the shape parameters. These operators are available, e.g., in the HDevelop menu Operators ▷ Graphics ▷ Drawing.
- **A region can be specified by image processing** using, e.g., a blob analysis. Then, the image is segmented, e.g., using a threshold operator, and the obtained region is further processed to select only those parts of it having specific features (commonly applied operators are, e.g., connection, fill_up, and select_shape). In HDevelop, you can determine suitable features and values using the dialog Visualization ▷ Feature Inspection. More complex regions can be created using set theory, i.e., by adding or subtracting standard regions using the operators union2 and difference. That way, e.g., also ring-shaped ROIs can be created like is shown in section 2.1.2.2.

Before creating the ROI, it is often suitable to optimize the reference image by a preprocessing. Note that when using shape-based matching, you can use an HDevelop Assistant that guides you through the matching process, which includes also the preprocessing of the reference image as well as the creation of the ROI. How to use the Matching Assistant is described in the HDevelop User's Guide, section 7.3 on page 217.

Note that also the gray values outside of the ROI have an influence on the model generation. In particular, the influence is approximately $2^{NumLevels}$ pixels, with n being the number of pyramid levels. Thus, the gray values outside of the ROI selected in the reference image should be similar to those that will occur in the search images.

2.1.2 Influence of the Region of Interest

The ROI used when creating the model determines the quality of the model and thus strongly influences the success of the later search. If the ROI is selected inappropriately, no or the wrong image structures are identified as instances of the model in the search images. So, if a matching is not successful or the result is inaccurate, you should have a very critical look at your ROI and try to enhance it such that the model represents the object and not the clutter that may be contained in the image. For approaches that are based on contours like shape-based matching and local or perspective deformable matching, you can use the operator inspect_shape_model to visually check a potential model. To create a model of good quality, you must pay attention to the proper selection of the ROI's center, i.e., the "point of reference" (see section 2.1.2.1) as well as to the proper selection of the ROI's outline (see section 2.1.2.2).

2.1.2.1 The Point of Reference

The ROI on the one hand influences the quality of the model and thus the general success of the subsequent matching. On the other hand, also the numerical results returned by the matching are influenced. In particular, the center point of the ROI by default acts as the so-called **point of reference** of the model for the estimated position, rotation, and scale. Note that if no ROI is selected, the point of reference is located at the center of the image (see figure 2.1).

The point of reference also influences the success of the later search, as 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 3.2.4.1 on page 63). That is, if a point of reference is placed away from the actual object (as shown, e.g., in figure 2.1, left), a later matching might fail although the object itself is fully contained in the search image, just because the point of reference is not contained. Thus, it is **very important to select an appropriate ROI** even if the template object is the only object in the image.

Note that for some approaches, after creating a model, you can modify the point of reference. But as a modified point of reference may lead to a decreased accuracy of the estimated position (see section 3.2.4.8 on page 68), **if possible, the point of reference should not be changed!** Additionally, even if you modified the point of reference, the test, if the point of reference lies within the domain of the search image, is always performed for the original point of reference, i.e., the center point of the initially selected ROI. Thus, the selection of an appropriate ROI for the template image is important right from the start.

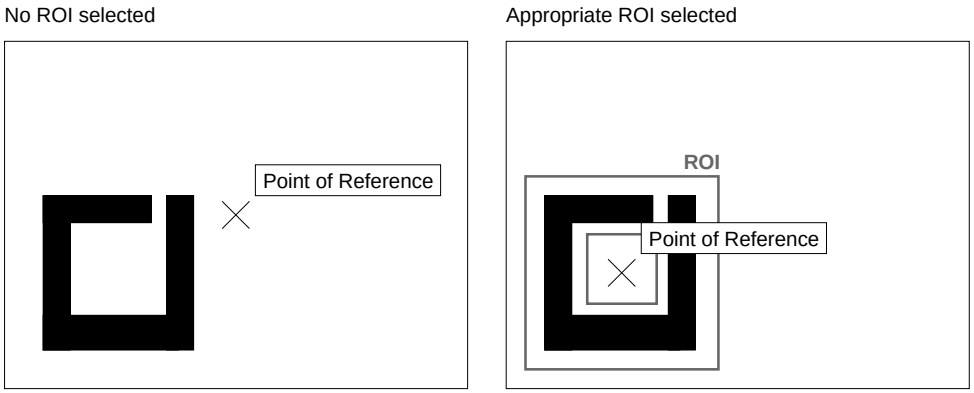


Figure 2.1: The point of reference depends on the selected ROI, not on the object or its bounding box.

2.1.2.2 The Outline of the ROI

The quality of the model and thus the accuracy of the result of the matching is influenced strongly by the model's outline, in particular, by its shape, size, and orientation.

To allow a good quality of the model, the ROI should be selected so that noise or clutter are minimized. This can be obtained, e.g., by "masking" parts of the object that contain clutter. In figure 2.2, e.g., the model of a capacitor is needed for shape-based matching. To exclude clutter as much as possible, instead of a circular ROI a ring-shaped ROI is created using the difference between two circular regions.

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

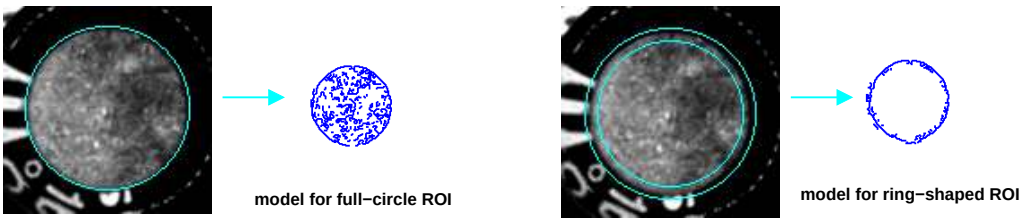


Figure 2.2: Masking the part of a region containing clutter.

Additionally, the accuracy of the location of the object is influenced by the number of contour points contained in the model. That is, a model with many contour points can be found more accurately than a model with few contour points (see figure 2.3).

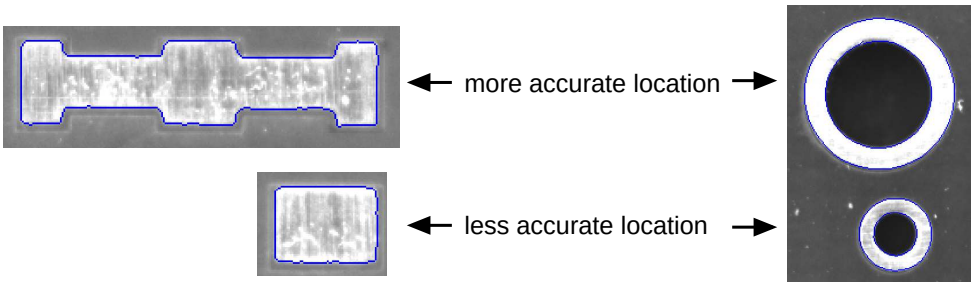


Figure 2.3: Location accuracy: the accuracy is better for models with many contour points.

Furthermore, the accuracy of a matching result can vary for different directions, depending on the direction of the contours that are contained in the model. If, e.g, a model consists mainly of horizontal contours as depicted in figure 2.4a, a good vertical accuracy can be obtained but the horizontal accuracy is bad. The model in figure 2.4b contains contours in vertical and horizontal direction. Thus, a sufficient accuracy in both directions can be obtained. The optimal pattern for a good accuracy in all directions would be a circular structure as shown in figure 2.4c. So please, carefully select the model so that it contains enough contours that are perpendicular to the direction that you want to inspect or measure after the matching.

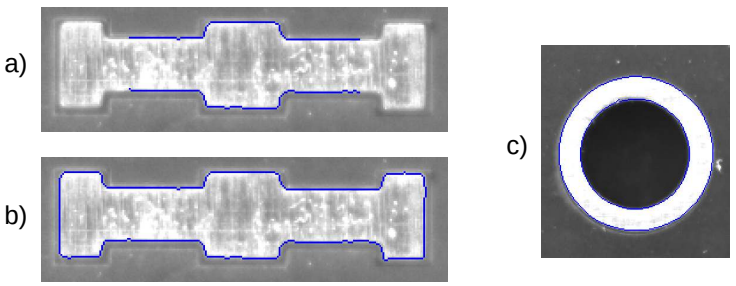


Figure 2.4: Direction of accuracy depends on the direction of the contours: (a) good vertical but bad horizontal accuracy, (b) sufficient accuracy in both direction, (c) optimal pattern for good accuracy in all directions.

The accuracy of the rotation angle returned by the matching depends on the one hand on the distance of the contour points from the rotation center and on the other hand on the orientation of the contour in relation to the rotation center (see figure 2.5). In particular, points that are far away from the rotation center can be determined more accurately. That is, assuming the same number of model contours, the orientation of a contour can be determined more accurately if the rotation center lies in the center of the contour and the contour is elongated.

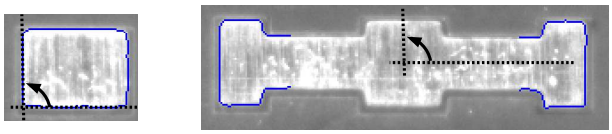


Figure 2.5: Rotational accuracy: the angle for the (left) compact contour is not determined with the same accuracy as the angle for the (right) extended model that is oriented radially to the rotation center.

2.1.3 Synthetic Models as Alternatives to Template Images

For some matching approaches, synthetic models can be used instead of a template image. Three means to work with synthetic models are available:

- create a synthetic template image,
- use XLD contours directly as models (this can be applied only for specific matching approaches as listed in section 2.1.3.2 on page 22), or
- use a DXF model to derive XLD contours, which then can be used directly as model (this can be applied only for specific matching approaches as listed in section 2.1.3.2 on page 22) or which can be used to create a synthetic template image.

2.1.3.1 Synthetic Template Image

Synthetic template images are suitable mainly for correlation-based matching and all 2D approaches that are based on contours, i.e., shape-based, component-based, local deformable, and perspective deformable matching.

Depending on the application it may be difficult to create a suitable model from a reference image because no image is available that contains a perfect, easy to extract instance of the object. An example of such a case is depicted in figure 2.6 for the location of capacitors. The task seems to be simple at first, as the capacitors are represented by prominent bright circles on a dark background. But the shape model that is derived from a circular

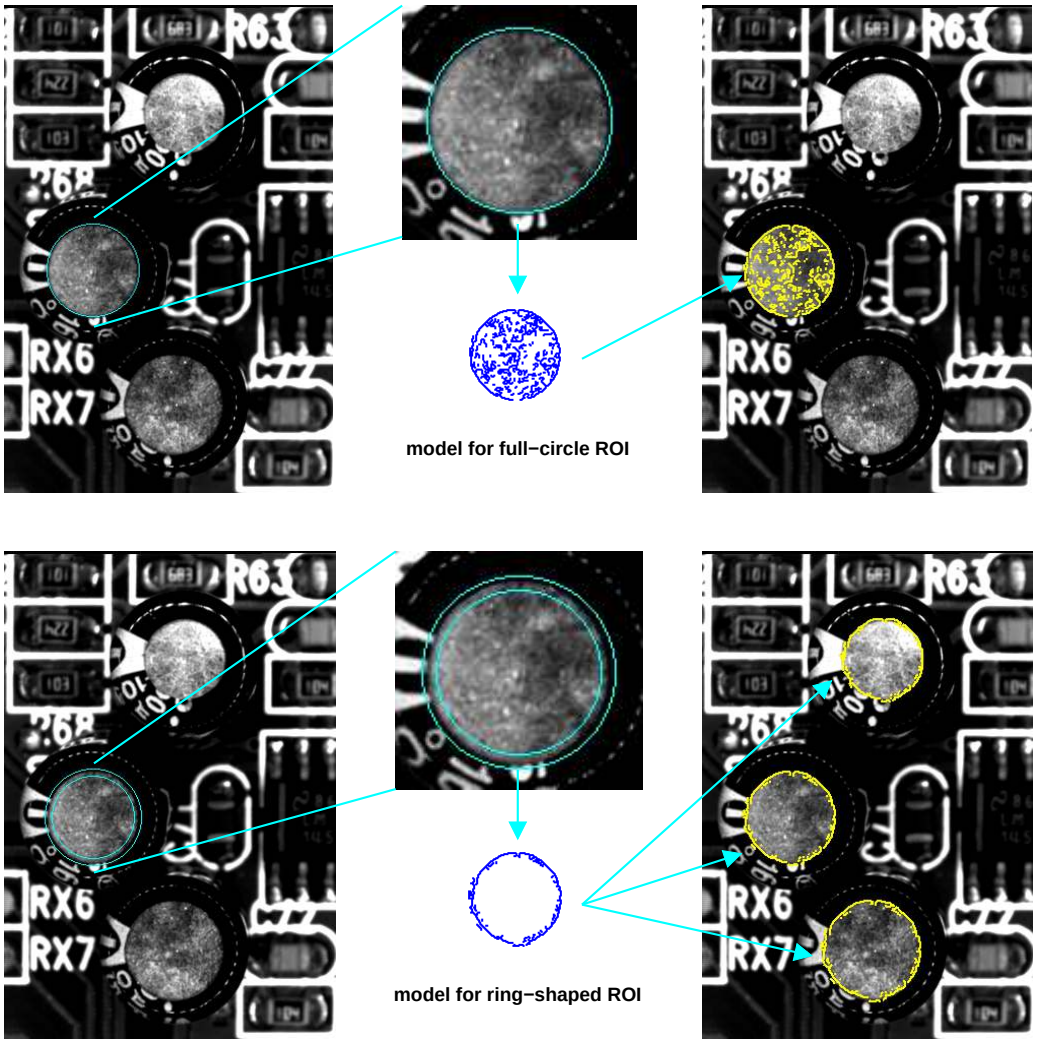


Figure 2.6: Locating capacitors using (top) a circular ROI or (bottom) a complex ring-shaped ROI to derive the model.

ROI is faulty, because inside and outside the circle the image contains clutter, i.e., high-contrast points that are not part of the object. A better result is obtained for a ring-shaped ROI that "masks" the parts of the object containing clutter. Nevertheless, the model is still not perfect, because parts of the circle are missing and the model still contains some clutter points.

In such a case, it is often better to use a synthetic template image. How to create such an image for the capacitors is explained below. To follow the example actively, start the HDevelop program `%HALCONEXAMPLES%\solution_guide\matching\synthetic_circle.hdev`.

Step 1: Create an XLD contour

First, we create a circular region using the operator `gen_ellipse_contour_xld`. You can determine a suitable radius by inspecting the image with the HDevelop dialog `Visualization` ▷ `Zoom Window` or more conveniently create the region using the ROI tool in HDevelop's graphics window.

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

Note that the synthetic image should be larger than the region because also pixels outside the region are used when creating the image pyramid for the shape-based matching, which in this case is the selected matching approach (for image pyramids, see section 2.3.2 on page 25).

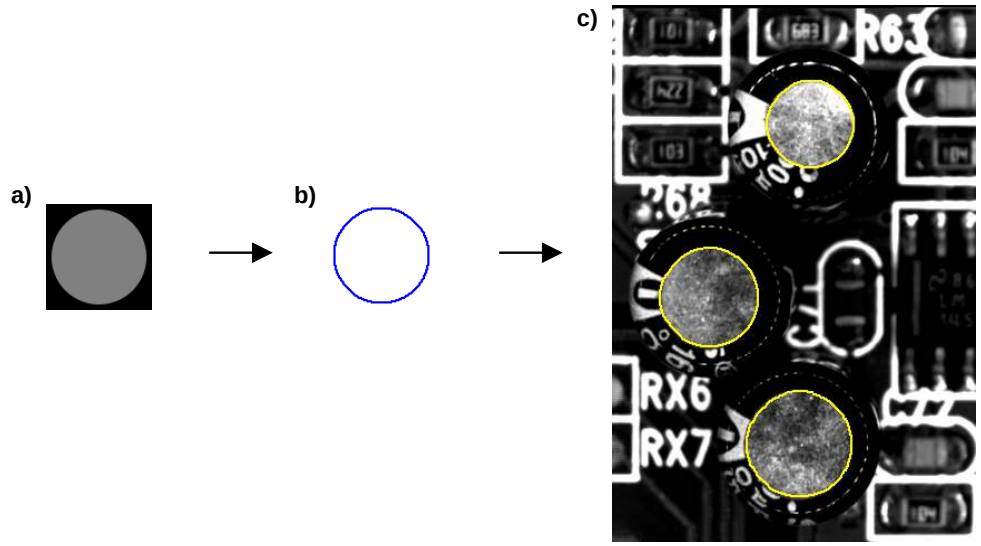


Figure 2.7: Locating capacitors using a synthetic model: a) paint region into synthetic image; b) corresponding model; c) result of the search.

Step 2: Create an image and insert the XLD contour

Then, we create an empty image using the operator `gen_image_const` and insert the XLD contour with the operator `paint_xld`. In figure 2.7a the resulting image is depicted.

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

Step 3: Create the model

The model is created from the synthetic image.

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

Figure 2.7b shows the corresponding model region and figure 2.7c shows the search results. Note how the image itself, i.e., its domain, acts as the ROI in this example.

2.1.3.2 Models from XLD Contours

For the shape-based matching and the local and perspective deformable matching you do not have to create a synthetic template image to derive a model from an XLD contour, because you can use the XLD contour directly as model. Then, e.g., for shape-based matching, you do not have to provide an image, select an ROI, and call one of the operators `create_shape_model`, `create_scaled_shape_model`, or `create_aniso_shape_model` to build the model. Instead, you simply call one of the operators `create_shape_model_xld`, `create_scaled_shape_model_xld`, or `create_aniso_shape_model_xld` with the XLD contour as input. An example for the creation of a shape model from circular XLD contours is given by the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Matching\Shape-Based\create_shape_model_xld.dev`:

```
gen_circle_contour_xld (ContCircle, 300, 300, MeanRadius, 0, 6.28318, \
                        'positive', 1)
create_shape_model_xld (ContCircle, 'auto', 0, 0, 'auto', 'auto', \
                        'ignore_local_polarity', 10, ModelID)
```

The proceeding for the local and perspective deformable matching is applied accordingly. There, you apply the operators `create_local_deformable_model_xld` for the local deformable matching and `create_planar_uncalib_deformable_model_xld` or `create_planar_calib_deformable_model_xld`, respectively, for perspective deformable matching.

Note that when creating the model from XLD contours, there is no information about the polarity of the model available (see figure 2.8, left). Thus, when creating the model the parameter `Metric` must be set so that the polarity is locally ignored. As this leads to a dramatically slow search, HALCON provides means to determine the polarity for a found model instance. That is, you apply the slow search once only in a search image with a polarity that is representative for your set of search images, project the model contours to the found position within the search image (see figure 2.8, right), and call the operator `set_shape_model_metric` for shape-based matching, `set_local_deformable_model_metric` for local deformable matching, and `set_planar_uncalib_deformable_model_metric` or `set_planar_calib_deformable_model_metric`, respectively, for perspective deformable matching. Then, polarity information is stored in the model and for the following search passes the parameter `Metric` can be set to a more suitable value, e.g., to `'use_polarity'`. This proceeding is strongly recommended for a fast and robust search.

```
find_shape_model (Image, ModelID, 0, 0, 0.7, 0, 0, 'least_squares', 0, 0.9, \
                  Row, Column, Angle, Score)
... accessing the indices of the matches
... that represent suitable drill holes
get_hom_mat2d_from_matching_result (Row[HoleIndices[0]], \
                                    Column[HoleIndices[0]], \
                                    Angle[HoleIndices[0]], 1, 1, HomMat2D)
set_shape_model_metric (Image, ModelID, HomMat2D, 'use_polarity')
for Index := 2 to 9 by 1
    read_image (Image, 'brake_disk/brake_disk_part_' + Index$'02d')
    find_shape_model (Image, ModelID, 0, 0, 0.7, 0, 0, 'least_squares', 0, \
                      0.9, Row, Column, Angle, Score)
endfor
```

For further information about the polarity of models for shape-based matching, please refer to section 3.2.3.5 on page 60.

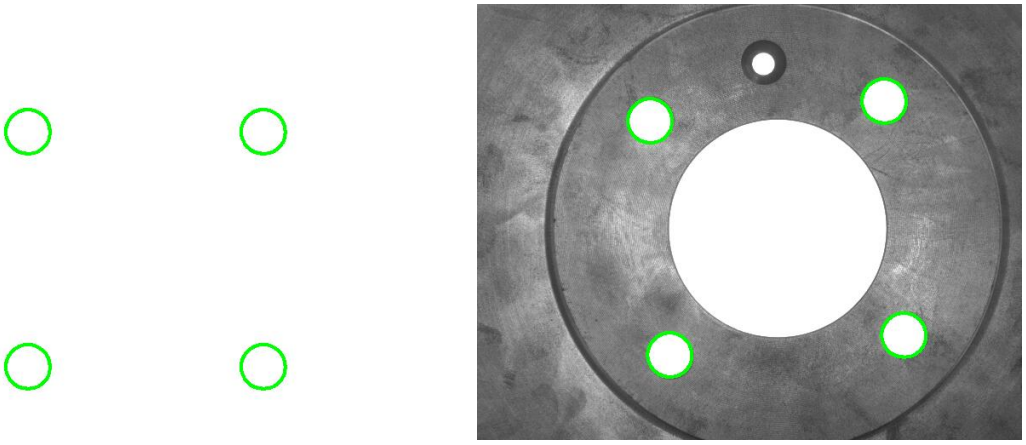


Figure 2.8: (left) synthetic XLD model; (right) model projected into a representative search image to determine the polarity of the model.

2.1.3.3 Models from DXF Files

The models needed for matching can also be derived from DXF files. In particular, you can extract the XLD contours from the DXF file using the operator `read_contour_xld_dxf` and then either use the XLD contours to create a synthetic image or, if the selected matching approach allows it (see section 2.1.3.2 on page 22), use the obtained XLD contours directly as model.

An example for the creation of a synthetic template image for shape-based matching using the XLD contours extracted from an DXF file is given by the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-2D\pm_multiple_dxf_models.dev`. Examples for the creation of perspective deformable models using the XLD contours extracted from DXF files directly as models are given by the HDevelop example programs `create_planar_calib_deformable_model_xld.dev` and `cre-`

ate_planar_uncalib_deformable_model_xld.dev (see figure 2.9) in the directory %HALCONEXAMPLES%\hdevelop\Matching\Deformable.



Figure 2.9: (left) synthetic XLD model accessed from a DXF model; (right) model found in a search image.

2.2 Reuse the Model

If you want to reuse created models or training results in other HALCON applications, all you need to do is to store the relevant information in files and then read them again.

The HDevelop example program %HALCONEXAMPLES%\solution_guide\matching\reuse_model.hdev shows exemplarily how to reuse a model for a uniformly scaled shape-based matching. First, the model is created.

```
create_scaled_shape_model (ImageROI, 'auto', -rad(30), rad(60), 'auto', 0.6, \
                           1.4, 'auto', 'none', 'use_polarity', 60, 10, \
                           ModelID)
```

Then, the model is stored in a file using the operator write_shape_model. With the model, HALCON automatically saves the XLD contour, the point of reference, and the parameters that were used in the call to create_scaled_shape_model.

```
write_shape_model (ModelID, ModelFile)
```

Note that the model region, i.e., the domain of the image, is not saved when storing the model. Thus, if you want to reuse it, too, you can store the template image with write_image.

```
ModelRegionFile := 'model_region_nut.png'
write_image (ImageROI, 'png', 0, ModelRegionFile)
```

The model, the XLD contour, and the point of reference are now read from the files using the operator read_shape_model. Then, the XLD contours and the point of reference are accessed using get_shape_model_contours and get_shape_model_origin, respectively. They are needed to visualize the result of the matching after a later call to find_scaled_shape_model. Furthermore, the parameters that were used to create the model are accessed with the operator get_shape_model_params, because some of them are needed as input for find_scaled_shape_model.

```
read_shape_model (ModelFile, ReusedModelID)
get_shape_model_contours (ReusedShapeModel, ReusedModelID, 1)
get_shape_model_origin (ReusedModelID, ReusedRefPointRow, ReusedRefPointCol)
get_shape_model_params (ReusedModelID, NumLevels, AngleStart, AngleExtent, \
                        AngleStep, ScaleMin, ScaleMax, ScaleStep, Metric, \
                        MinContrast)
```

The previously stored model region can be read with read_image and the corresponding domain is accessed by get_domain.

```
read_image (ImageModelRegion, 'model_region_nut.png')
get_domain (ImageModelRegion, DomainModelRegion)
```

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, 0.65, 0, 0, \
                         'least_squares', 0, 0.8, RowCheck, ColumnCheck, \
                         AngleCheck, ScaleCheck, Score)
for i := 0 to |Score| - 1 by 1
    get_hom_mat2d_from_matching_result (RowCheck[i], ColumnCheck[i], \
                                        AngleCheck[i], ScaleCheck[i], \
                                        ScaleCheck[i], \
                                        MoveAndScalingOfObject)
    affine_trans_contour_xld (ReusedShapeModel, ModelAtNewPosition, \
                              MoveAndScalingOfObject)
    dev_display (ModelAtNewPosition)
endfor
```

Note that the information that is stored in a model and that can be queried after reading a model from file depends on the selected approach. Similar to shape-based matching, many of the approaches provide operators to query the origin of the model and several parameters that were used to create or modify the model. Some approaches additionally provide operators to query the contour of the model or in case of the descriptor-based matching the interest points that define the model.

2.3 Speed Up the Search

You can speed up the search by

- restricting the search space (section 2.3.1) and
- using subsampling (section 2.3.2).

2.3.1 Restrict 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. Depending on the matching approach, this space encompasses not only the two 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.

The most obvious way to restrict the search space, which is suitable for all matching approaches, is to apply the operator that is used to find the model in an image to an ROI instead of the whole image.

Other parameters that can be used to restrict the search space depend on the specific matching approach. Some approaches are already very strict, others allow different changes of the object like rotation or scale and thus a restriction of the search space is strongly recommended. Shape-based matching, e.g., allows arbitrary orientation and scale changes as well as occlusions of the model instances in the search images. Thus, for shape-based matching also the range of orientation and scale as well as the visibility, i.e., the amount of allowed occlusions of the model instance in the image, should be restricted to speed up the search (see section 3.2.4 on page 61).

2.3.2 About Subsampling

For all matching approaches except the descriptor-based matching a so-called image pyramid can be used to speed up the search. In some approaches, the image pyramid is created for the search image as well as for the template

image The pyramid consists of the original, full-sized image and a set of downsampled images. For example, if 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 2.10 depicts four levels of an example image pyramid together with the corresponding model regions.

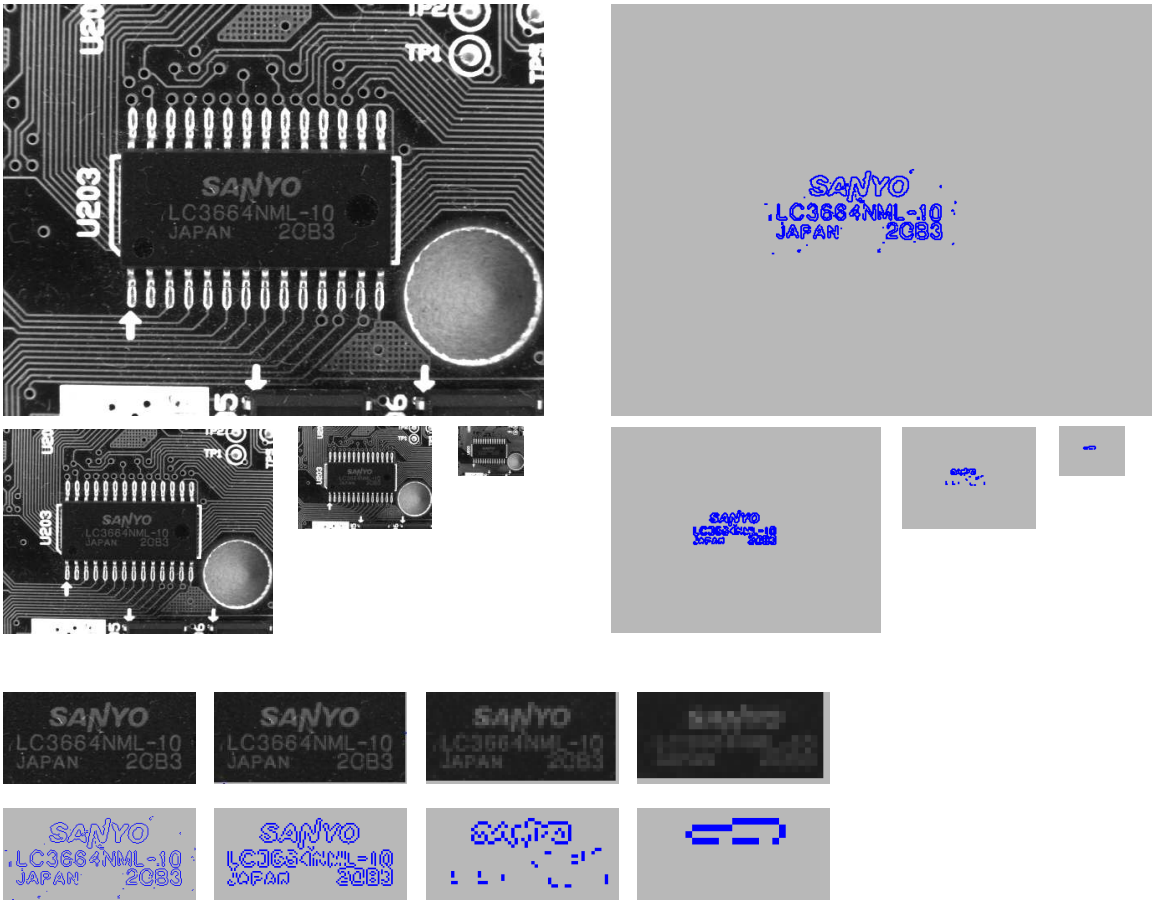


Figure 2.10: The image and the model region at four pyramid levels (original size and zoomed to equal size).

Note that when images are downsampled, different “events” occur. Primarily, specific structures disappear in higher pyramid levels. As shown in figure 2.10, thin structures disappear sooner than thick ones. Note that “thin structures” includes the different structures of the actual model but also the distances between them. Thus, small structures of the model may disappear, but they can just as well be merged if the distances between them are small. Additionally, the boundaries of structures blur in higher pyramid levels and their contrast decreases. In figure 2.11, this effect is shown for a one pixel wide line.

For more complex patterns, this may lead to the disappearance of a pattern or even to the creation of a new pattern. Examples for the disappearance of patterns that are caused by the blurring of structures are given for the small characters in figure 2.10, for a regular grid of one pixel wide high contrast lines in figure 2.12, and for a pattern that consists of high contrast rectangles with a distance of one pixel in figure 2.13. The latter example also shows that new patterns can be created within the pyramid. Before the pattern is blurred to a homogeneous area, a different regular pattern is created. Additionally, it is possible that new structures appear because the region of interest was selected too large and thus, contours that were not part of the model in the first pyramid level become part of the model in a higher level.

The blurring of structures induces also that besides the size also the contrast determines if a structure is still contained in a higher pyramid level or not. After the downsampling, a high contrast structure of sufficient size may still have enough contrast to be recognized, whereas a structure of the same size but with a low contrast can not be distinguished from the background anymore. For example, in figure 2.14 on page 28 the large but bright ’V’ of the ’MVTec’ logo disappears in front of the bright background at a pyramid level in which the large high contrast



Figure 2.11: In the pyramid, a (left) one pixel wide high contrast line blurs to a (right) broader line with a lower contrast.

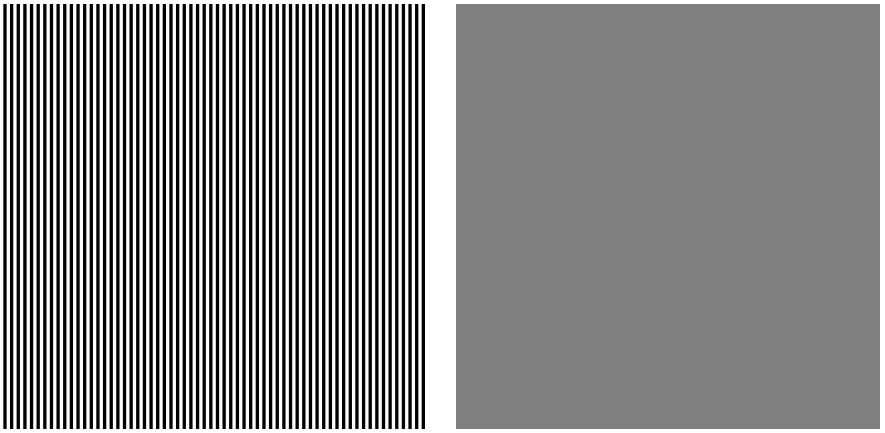


Figure 2.12: In the pyramid, a (left) grid with one pixel wide high contrast lines becomes a (right) homogeneous gray-value area.

characters are still contained. Thus, to benefit from the speed-up obtained by a subsampling, the images should

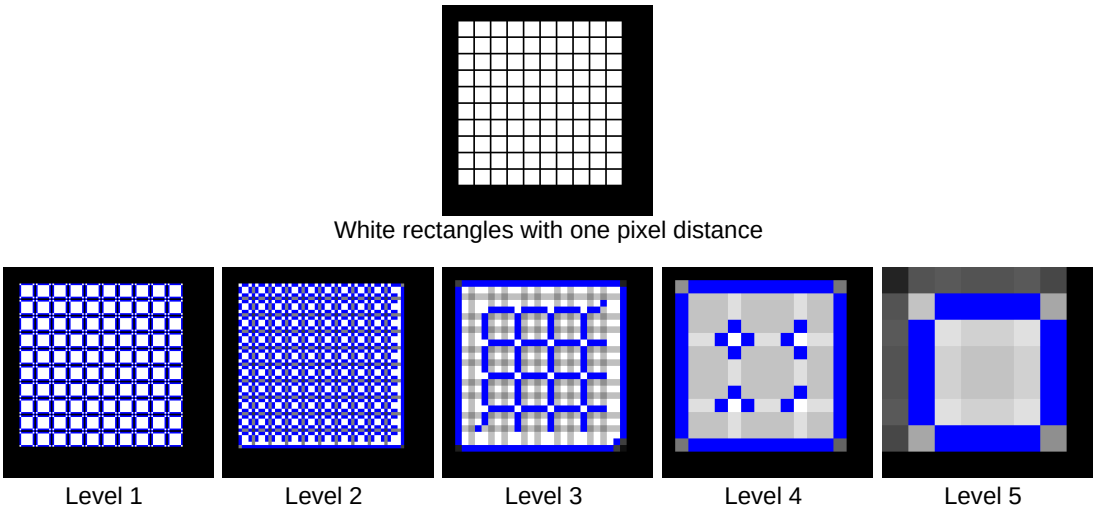


Figure 2.13: In the pyramid, a (top) regular structure of rectangles with distances of one pixel shows the following behavior: (bottom) level 1 and 2 still represent the correct model contours, level 3 and 4 produce new regular patterns, and in level 5 a homogeneous gray-value area is obtained.

have as much contrast as possible.

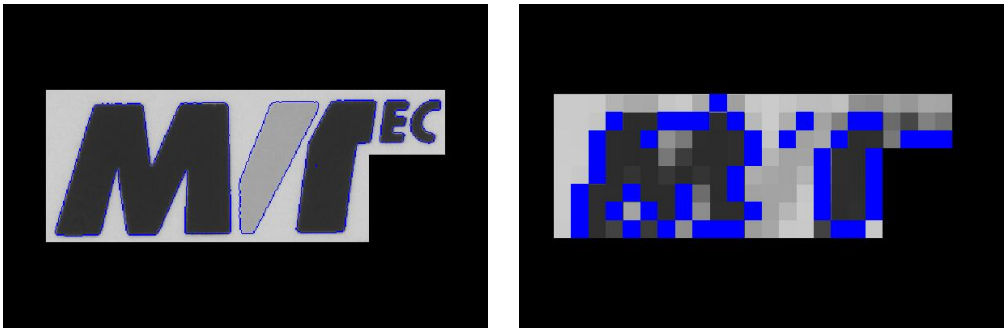


Figure 2.14: In the pyramid, a part of the model with a low contrast (the 'V' of MVTec) disappears.

2.4 Use the Results of Matching

The main goal of matching is to locate objects in images, i.e., above all get the position and in most cases also the orientation of a model instance in an image. Furthermore, many approaches return additional information like the scale of the object or a score that evaluates the quality of the returned object location. In the following sections

- the individual results of the different matching approaches are listed (section 2.4.1),
- the different types of transformations that can be handled with HALCON and that are often needed to further process the results of matching are introduced (section 2.4.2),
- the different uses of the most common results, i.e., the estimated position and orientation, are described (section 2.4.3 on page 32),
- the use of the scale is introduced (section 2.4.4 on page 41),
- the use of a 2D projective transformation matrix (2D homography) is introduced (section 2.4.5 on page 42),
- the use of a 3D pose is introduced (section 2.4.6 on page 45), and
- the returned score is shortly discussed (section 2.4.7 on page 46).

2.4.1 Results of the Individual Matching Approaches

To locate objects in images, information like the position and orientation of the searched objects must be returned by the matching. This information is available in different representations, depending on the selected matching approach. That is, for a strict 2D matching the position and orientation are returned separately, i.e., the position consists of a row and a column value and the orientation consists of a single value describing the angle. In contrast, the uncalibrated perspective approaches return the position and orientation together in a projective transformation matrix (2D homography) and the calibrated perspective approaches return them together in a 3D pose. Besides the position and orientation, many approaches return further information like the scale of the found object or information about the quality of the match, which is called score.

The results of the individual matching approaches are listed below. How to further process them is described in the following sections.

Matching Approach	Operator(s) and Results
correlation-based matching	`find_ncc_model`: Position, rotation, and score of the found model.
	`find_ncc_models`: Positions, rotation angles, and scores for multiple models, and the information to which model each instance belongs.
shape-based matching	`find_shape_model`: Position, rotation, and score of the found model.
	`find_shape_models`: Positions, rotation angles, and scores for multiple models, and the information to which model each instance belongs.
	`find_scaled_shape_model`: Position, rotation, a uniform scaling factor, and score of the found model
	`find_scaled_shape_models`: Positions, rotation angles, uniform scaling factors, and scores for multiple models, and the information to which model each instance belongs.
	`find_aniso_shape_model`: Position, rotation angle, scaling factors for row and column direction, and score of the found model.
	`find_aniso_shape_models`: Positions, rotation angles, scaling factors in row and column direction, and scores for multiple models, and the information to which model each instance belongs.
component-based matching	`find_component_model`: Start and end index for each found instance of the component model, a score for the found instances of the component model, positions and angles of the found component matches, the scores for the found matches, and the indices of the found components.
local deformable matching	`find_local_deformable_model`: Position, vector field, rectified image (part), contours of the deformed object, and score of the found model
perspective deformable matching	`find_planar_calib_deformable_model`: 3D pose, six mean square deviations or the 36 covariances of the 6 pose parameters, and score.
	`find_planar_uncalib_deformable_model`: 2D homography and score.
descriptor-based matching	`find_calib_descriptor_model`: 3D pose and score.
	`find_uncalib_descriptor_model`: 2D homography and score.

2.4.2 About Transformations

HALCON provides operators for different types of transformations. For the strict 2D matching approaches, 2D affine transformations are available, which allow to, e.g., translate, rotate, or scale 2D objects (see section 2.4.2.1). For the uncalibrated perspective matching approaches, 2D projective transformations are provided, which are applied in the context of perspective views (see section 2.4.2.2 on page 31). Finally, for the calibrated perspective matching approaches, 3D affine transformations are available (see section 2.4.2.3 on page 32).

2.4.2.1 2D Affine Transformations

"Affine transformation" is a technical term in mathematics describing a certain group of transformations. Figure 2.15 shows the types that occur, e.g., in the context of the shape-based matching: An object can be *translated*

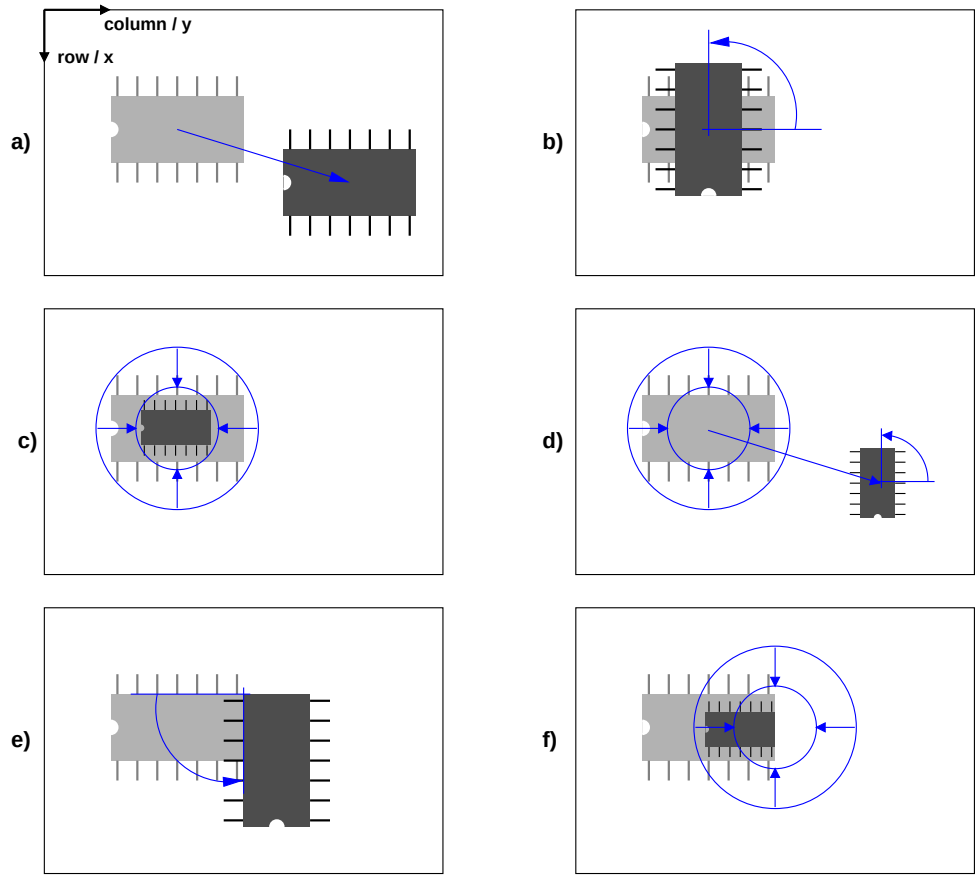


Figure 2.15: Typical affine transformations: a) translation along two axes; b) rotation around the IC center; c) scaling around the IC center; d) combining a, b, and c; e) rotation around the upper right corner; f) scaling around the right IC center.

(moved) along the two axes, *rotated*, and *scaled*. In figure 2.15d, all three transformations were applied in a sequence.

Note that for the rotation and the scaling a fixed point exists, around which the transformation is performed. It corresponds to the point of reference described in section 2.1.2.1 on page 18. In figure 2.15b, e.g., the IC is rotated around its center, in figure 2.15e 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 transformations in a simple way.

HALCON allows to transform pixels, regions, images, and XLD contours by providing the operators `affine_trans_pixel`, `affine_trans_region`, `affine_trans_image`, and `affine_trans_contour_xld`. The transformation in figure 2.15d corresponds to the line

```
affine_trans_region (IC, TransformedIC, ScalingRotationTranslation, \
                     'nearest_neighbor')
```

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 with `hom_mat2d_identity`.

```
hom_mat2d_identity (EmptyTransformation)
```

Then, the scaling around the center of the IC is added with `hom_mat2d_scale`.

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

Similarly, the rotation and the translation are added with hom_mat2d_rotate and hom_mat2d_translate.

```
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 2.15a clarifies the relation.

Transformation 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 and then use this matrix to compute the transformed region.

```
vector_angle_to_rigid (RowCenterIC, ColumnCenterIC, 0, \
                       TransformedRowCenterIC, TransformedColumnCenterIC, \
                       rad(90), RotationTranslation)
affine_trans_region (IC, TransformedIC, RotationTranslation, \
                     'nearest_neighbor')
```

If a pixel, image, or contour should be transformed, the proceeding is similar, but instead of affine_trans_region, you apply the operator affine_trans_pixel, affine_trans_image, or affine_trans_contour_xld, respectively (see, e.g., section 2.4.3.1 on page 33).

2.4.2.2 2D Projective Transformations

A 2D projective transformation matrix describes a perspective projection as illustrated in figure 2.16. It consists of 3×3 values. Note that if the last row contains the values [0,0,1], it corresponds to a homogeneous transformation matrix, i.e., it describes a 2D affine transformation, which is a special case of the 2D projective transformation.

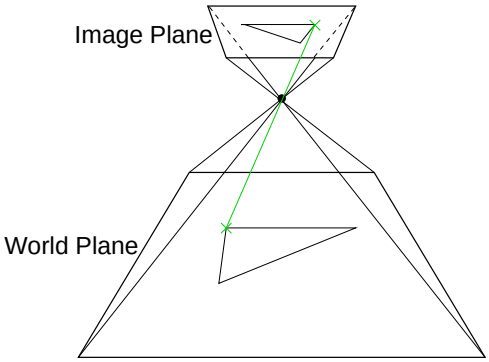


Figure 2.16: Perspective projection.

HALCON provides several operators to apply projective transformations. Similar to the affine transformation, with a projective transformation you can transform different HALCON structures like pixels (projective_trans_pixel), regions (projective_trans_region), images (projective_trans_image), and XLD contours (projective_trans_contour_xld). But now, also perspective deformations are considered by the transformation. An example for a 2D projective transformation is described in more detail in section 2.4.5 on page 42.

Note that in the context of matching, projective transformation matrices are even easier to handle than affine transformations, as you do not need to create a transformation matrix from corresponding points or by adding several transformation steps. Instead, the projective transformation matrix is directly returned by one of the operators that are used for the uncalibrated matching of perspectively distorted objects, and can be directly used to apply one of the above stated transformations.

2.4.2.3 3D Affine Transformations

Detailed information about 3D affine transformations can be found in the Solution Guide III-C. Here, we shortly summarize the information needed to overlay the results of the calibrated perspective matching approaches with structures obtained from the reference image.

HALCON's 3D affine transformation mainly is a transformation from a 3D point to another 3D point. Thus, in contrast to the 2D affine or 2D projective transformation, it cannot transform regions, images, or XLD contours. Thus, any 2D structure that you want to transform, e.g., for the visualization of the matching result, must be split into points before applying the transformation. Additionally, the points must be available in the world coordinate system (WCS) (step 1 in figure 2.17).

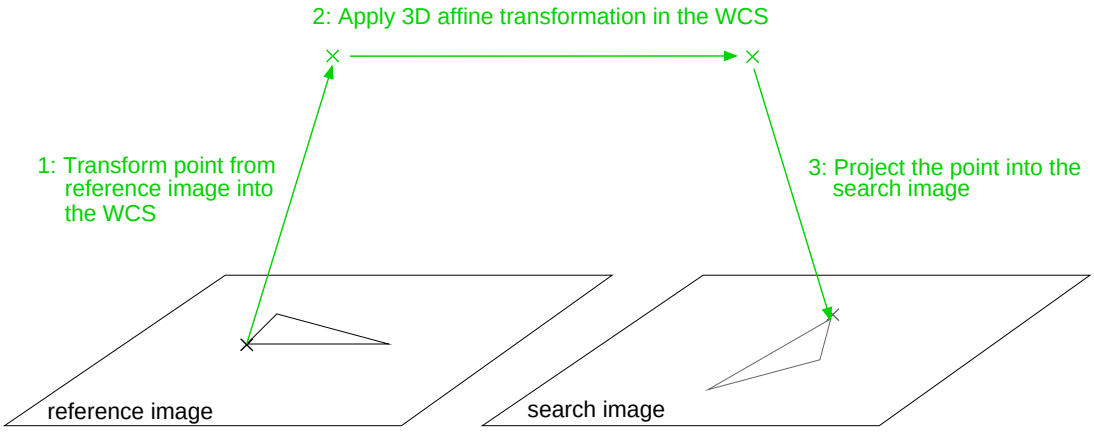


Figure 2.17: Different steps of the transformations needed in the context of calibrated perspective matching.

To transform a point from the reference image into the WCS, you need the camera parameters and the reference pose of the model, which describes the relation between the object and the camera. Both are typically obtained by a camera calibration (see Solution Guide III-C, section 3.2 on page 59). The image points can then be transformed into the WCS using `image_points_to_world_plane`. Alternatively, you can transform a contour to the plane with $z = 0$ of the WCS using `contour_to_world_plane_xld` and then get the individual points of it using `get_contour_xld`. Note that the latter operator returns only tuples for the x and the y coordinates. Thus, a further tuple for the z coordinates must be created using `gen_tuple_const`. This tuple must have the same number of elements as the other two tuples and all elements must be set to the value `0`.

If the individual points are available in the WCS, they can be transformed with `affine_trans_point_3d` using a 3D homogeneous transformation matrix (step 2 in figure 2.17). In the context of calibrated perspective matching, the result of the matching is a 3D pose that can be transformed into the corresponding 3D homogeneous transformation matrix using `pose_to_hom_mat3d`. It describes the relations between the world coordinates of the model and those of the model instance found in a specific match.

After transforming the 3D points with `affine_trans_point_3d`, you can project them from the WCS into the search image using `project_3d_point` (step 3 in figure 2.17). These image points can then be used to reconstruct and display the initial structure using, e.g., `gen_contour_polygon_xld` for the reconstruction of a contour. An example for a 3D affine transformation is described in more detail in section 2.4.6 on page 45.

2.4.3 Use the Estimated 2D Position and Orientation

The estimated position and orientation (2D pose) for a model instance of a strict 2D matching can be used in different ways. They can be used, e.g.,

- to display the found instance (section 2.4.3.1 for a single match or section 3.2.5.1 on page 72 for multiple matches),
- to align ROIs for other inspection tasks, e.g., measuring (section 2.4.3.2 on page 34), or
- to transform the search image so that the object is positioned as in the template image (section 2.4.3.3 on page 38).

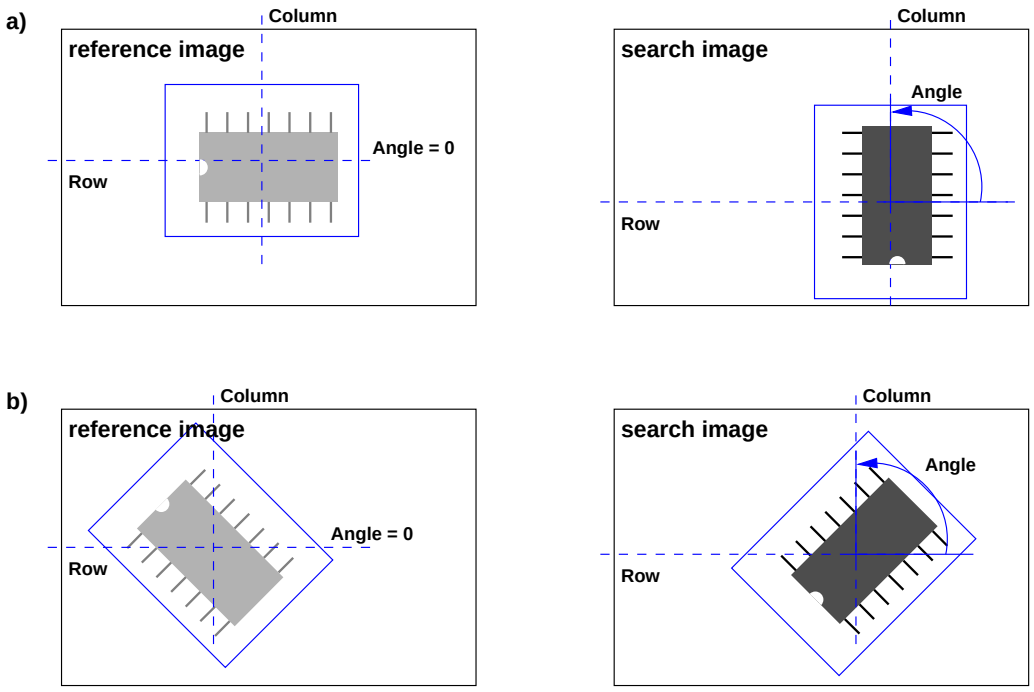


Figure 2.18: The position and orientation of a match: a) The center of the ROI acts as the default point of reference; b) In the template image, the orientation is always 0.

All of these applications can be realized with a few lines of code using the affine transformations that were explained in section 2.4.2.1 on page 29.

For most matching approaches, the position and orientation returned by the matching is determined relative to the model. That is, no absolute position but the distance to the point of reference is returned. The point of reference is defined by the center of the ROI that was used to create the model (see figure 2.18a). By default, the point of reference for the model is moved to the coordinates `(0,0)`. But even then, as different tasks in HALCON need different image coordinate systems (the positions for pixel-precise regions differ from those used for subpixel-precise XLD contours by 0.5 pixels, see page 37), **the estimated position returned by the matching can not be used directly** but is optimized for the creation of the transformation matrices that are used to apply the applications described above. Additionally, in the template image the object is taken as not rotated, i.e., its angle is `0`, even if it seems to be rotated as, e.g., in figure 2.18b.

2.4.3.1 Display the Matches

In many cases, especially during the development of a matching application, it is useful to display the matching results in the search image. This can be realized by different means. If the **model is represented by a contour**, we recommend to overlay the XLD contour on the search image, because XLD contours can be transformed more precisely and quickly than regions. Additionally, by displaying the contour of the model not only the position and orientation of the found model instance but also the deviation from the model's shape can be visualized.

If **no contour is available**, we recommend to overlay the ROI that was used to create the model on the search image. In some cases, the additional visualization of points is suitable. For example, for the descriptor-based matching, the interest points of the model or a found model instance can be queried as described in section 3.6.3 on page 110 and displayed using, e.g., `gen_cross_contour_xld`.

In the following we show how to overlay the XLD contour or the ROI on the search image. The HDevelop program `%HALCONEXAMPLES%\solution_guide\matching\first_example_shape_matching.hdev` exemplarily shows the steps needed to visualize the results of a shape-based matching.

Step 1: Access the XLD contour containing the model

First, the model is created using `create_shape_model`. Then, the XLD version of the model is accessed with `get_shape_model_contours`.

```
create_shape_model (ImageROI, NumLevels, 0, rad(360), 'auto', 'none', \
                    'use_polarity', 30, 10, ModelID)
get_shape_model_contours (ShapeModel, ModelID, 1)
```

Step 2: Determine the affine transformation

The matching is applied using the operator `find_shape_model`. A visualization is only reasonable if the matching was successful. Thus, the results of the matching are checked. If the matching failed, the operator returns empty tuples in parameters like `Score`. If the matching was successful, the corresponding affine transformation can be constructed with the operator `vector_angle_to_rigid` from the initial position and orientation of the XLD contour and the position and orientation of the match. Note that the initial XLD contour of the model is located at the origin of the image and not at the position of the model in the reference image. Thus, the first two parameters are set to the value `0`.

```
find_shape_model (SearchImage, ModelID, 0, rad(360), 0.7, 1, 0.5, \
                  'least_squares', 0, 0.7, RowCheck, ColumnCheck, \
                  AngleCheck, Score)
if (|Score| == 1)
    vector_angle_to_rigid (0, 0, 0, RowCheck, ColumnCheck, AngleCheck, \
                           MovementOfObject)
```

Step 3: Transform the XLD

Now, the transformation is applied to the XLD contour of the model using the operator `affine_trans_contour_xld` and the transformed contour is displayed. Figure 3.3 on page 54 in section 3.2 shows the search image and the overlaid XLD contour.

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

The general procedure for the projection of a contour or a region is the same. That is, in both cases the operator `vector_angle_to_rigid` is used to get the homogeneous transformation matrix (2D homography) that describes the relation between the model and the found model instance. As input it needs the initial position of the contour or the region that has to be transformed, the initial angle of the model (which is by default `0`), the position of the found model instance, and the angle of the found model instance. The returned 2D homography is then used by the operator `affine_trans_contour_xld` or `affine_trans_region`, respectively, to transform the contour or region so that the position and orientation of it corresponds to the position and orientation of the found model instance. The transformed region or contour can then be displayed with `dev_display`.

The **main difference between the projection of a contour and the projection of a region** regards the initial position of the contour or region. The initial position of the contour or region always describes the "relative" position of the point of reference, i.e., its distance to the default point of reference. The default position of the point of reference is `(0,0)`, i.e., it is the origin of the image and not the position of the model in the reference image. But depending on the structure that has to be transformed, the "relative" position can change. Whereas the **XLD contour** of a shape model is derived from the model and thus its point of reference is also the point of reference of the model, a **region** naturally was defined before the model was created. Thus, its point of reference describes the position of the model in the reference image instead of the point of reference of the later created model. That is, instead of `(0,0)`, the values obtained by the operator `area_center` for the respective region have to be passed as the first two parameters to `vector_angle_to_rigid`.

Furthermore, if you have changed the reference point of a model (see, e.g., section 2.1.2 on page 18), the values to insert as initial position change in the following way: If you query an XLD contour from a model after changing the point of reference, nothing changes. The initial position still is `(0,0)`, only another part of the model serves as point of reference. If you change the point of reference after querying the XLD contour or if you want to transform a region, which is naturally also accessed before changing the point of reference, you have to add the values of the initial point of reference to the values of the new point of reference.

2.4.3.2 Align ROIs for other Inspection Tasks

The results of 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 suitable, e.g., if the object to be inspected is allowed to move

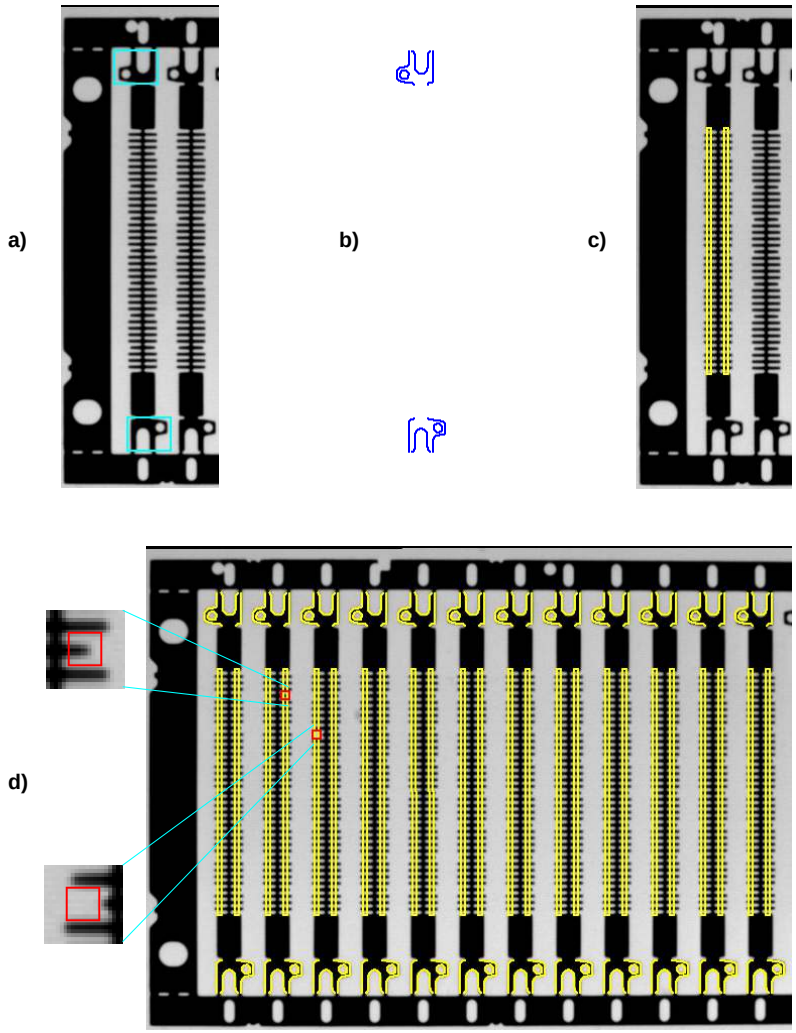


Figure 2.19: Aligning ROIs for inspecting parts of a razor: a) ROIs for the model; b) the model; c) measuring ROIs; d) inspection results with zoomed faults.

or if multiple instances of the object are to be inspected at once.

The HDevelop example program `%HALCONEXAMPLES%\solution_guide\matching\align_measurements.hdev`, e.g., uses shape-based matching to inspect multiple razor blades by measuring the width and the distance of their “teeth”.

First, a shape model is created that will be used later to align the measurement ROIs. Figure 2.19a shows the model ROI for the shape model, which consists of two united regions, and figure 2.19b shows the corresponding shape model.

Then, the inspection task is realized with the following steps:

Step 1: Position the measurement ROIs for the model blade

Two rectangular measurement ROIs are generated with `gen_rectangle2` so that they are placed over the teeth of the razor blade that acts as the model (see figure 2.19c).

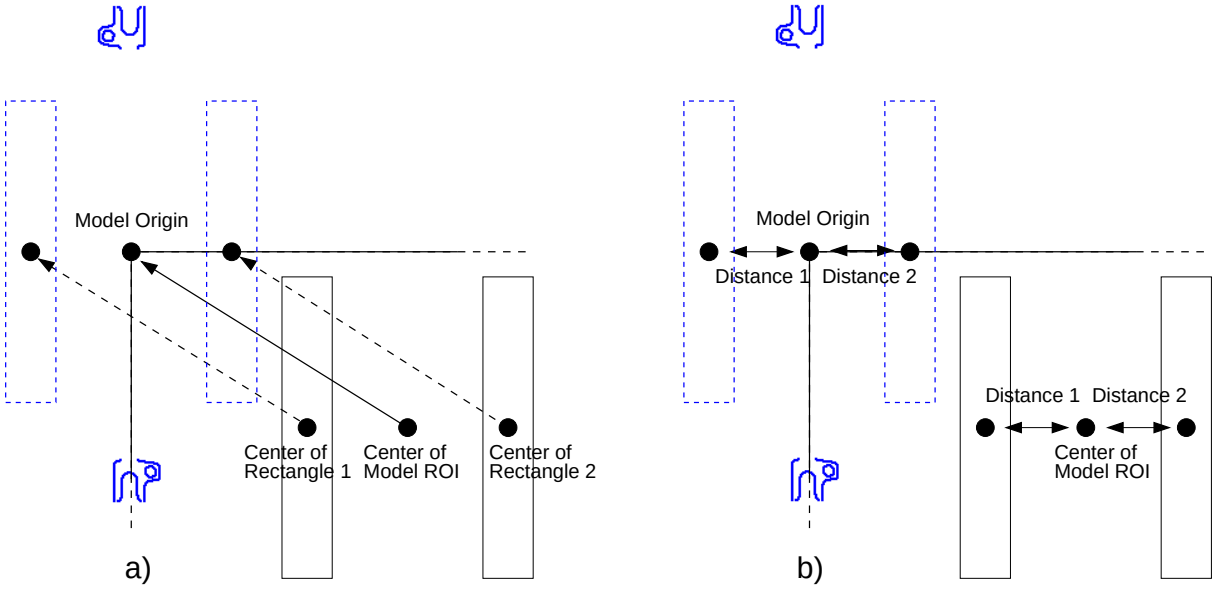


Figure 2.20: Align the measure ROIs relative to the shape model: a) first, move the measure ROIs to lie on the XLD model; b) then, determine the new reference positions for the measure ROIs by their distance to the model.

```
Rect1Row := 244
Rect1Col := 73
DistColRect1Rect2 := 17
Rect2Row := Rect1Row
Rect2Col := Rect1Col + DistColRect1Rect2
RectPhi := rad(90)
RectLength1 := 122
RectLength2 := 2
gen_rectangle2 (MeasureROI1, Rect1Row, Rect1Col, RectPhi, RectLength1, \
                RectLength2)
gen_rectangle2 (MeasureROI2, Rect2Row, Rect2Col, RectPhi, RectLength1, \
                RectLength2)
```

To be able to transform them later along with the XLD contour of the model, they are moved with `move_region` to lie on the XLD model, whose point of reference is the origin of the image (see figure 2.20a), which is queried with `area_center`. Note that before moving the regions the clipping must be switched off, otherwise the region parts that are moved "outside" the image are clipped. The distances between the center of the model ROI and the centers of the rectangular measure ROIs define the new reference positions for the measure ROIs (see figure 2.20b).

```
area_center (ModelROI, Area, CenterROIRow, CenterROIColumn)
get_system ('clip_region', OriginalClipRegion)
set_system ('clip_region', 'false')
move_region (MeasureROI1, MeasureROI1Ref, -CenterROIRow, -CenterROIColumn)
move_region (MeasureROI2, MeasureROI2Ref, -CenterROIRow, -CenterROIColumn)
set_system ('clip_region', OriginalClipRegion)
DistRect1CenterRow := Rect1Row - CenterROIRow
DistRect1CenterCol := Rect1Col - CenterROIColumn
DistRect2CenterRow := Rect2Row - CenterROIRow
DistRect2CenterCol := Rect2Col - CenterROIColumn
```

Step 2: Find all razor blades

Now, all instances of the shape model are searched for in the search image using `find_shape_model`.

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

Step 3: Determine the affine transformation

For each instance, i.e., for each found razor blade, the transformation between the model and the found model instance is calculated with `vector_angle_to_rigid` and applied to the XLD model with `affine_trans_contour_xld` so that it lies over the found model instance in the search image.

```
for i := 0 to |Score| - 1 by 1
    vector_angle_to_rigid (0, 0, 0, RowCheck[i], ColumnCheck[i], \
                           AngleCheck[i], MovementOfObject)
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition, \
                              MovementOfObject)
```

Step 4: Create measurement objects at the corresponding positions

Then, `affine_trans_pixel` is applied to calculate the corresponding positions of the measure ROIs, at which the measure objects are created.

```
    affine_trans_pixel (MovementOfObject, DistRect1CenterRow, \
                        DistRect1CenterCol, Rect1RowCheck, Rect1ColCheck)
    affine_trans_pixel (MovementOfObject, DistRect2CenterRow, \
                        DistRect2CenterCol, Rect2RowCheck, Rect2ColCheck)
    RectPhiCheck := RectPhi + AngleCheck[i]
    gen_measure_rectangle2 (Rect1RowCheck, Rect1ColCheck, RectPhiCheck, \
                            RectLength1, RectLength2, Width, Height, \
                            'bilinear', MeasureHandle1)
    gen_measure_rectangle2 (Rect2RowCheck, Rect2ColCheck, RectPhiCheck, \
                            RectLength1, RectLength2, Width, Height, \
                            'bilinear', MeasureHandle2)
```

Note that **you must use the operator `affine_trans_pixel` and not `affine_trans_point_2d`**, because the latter uses a different coordinate system. In particular, it uses the standard image coordinate system for which a position corresponds to the center of a pixel (see figure 2.21, right). In contrast, the operators `affine_trans_pixel`, `affine_trans_contour_xld`, `affine_trans_region`, and `affine_trans_image` use the coordinate system depicted in figure 2.21 (left).

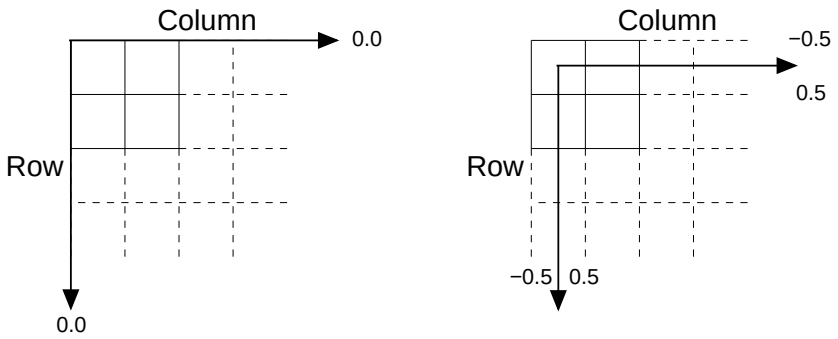


Figure 2.21: (left) image coordinate system used for the matching methods and operators like `affine_trans_pixel`; (right) standard image coordinate system.

In the example application, the individual razor blades are only translated but not rotated relative to the model position. Thus, instead of applying the full affine transformation to the measure ROIs and then creating new measure objects, you can 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 5: Measure the width and the distance of the "teeth"

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

```
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)
```

Step 6: Inspect the measurements

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 2.19d shows the inspection results for the example.

```
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)
```

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

2.4.3.3 Align the Search Results

The previous sections showed how to use the matching results to determine the so-called forward transformation. This was used to transform objects from the model into the search image or, respectively, to transform ROIs that were specified in the reference image into the search image.

You can also determine the inverse transformation which transforms objects from the search image back into the reference image. With this transformation, you can align the search image (or parts of it), i.e., transform it such that the matched object is positioned as it was in the reference 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 by alignment the image is only rotated and translated. To remove perspective or lens distortions, e.g., if the camera observes the scene under an oblique angle, you must rectify the image first (see section 3.2.6 on page 74 for more information).

Inverse Transformation

The inverse transformation can be determined and applied in a few steps. The task of the HDevelop program `%HALCONEXAMPLES%\solution_guide\matching\rectify_results.hdev`, e.g., is to extract the serial number on CD covers (see figure 2.22). The matching is realized using a shape-based matching.

Step 1: Calculate the inverse transformation

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

```
vector_angle_to_rigid (CenterModelROIRow, CenterModelROIColumn, 0, \
                       RowMatch, ColumnMatch, AngleMatch, \
                       MovementOfObject)
hom_mat2d_invert (MovementOfObject, InverseMovementOfObject)
```

Note that in contrast to the previous sections, the transformation is calculated based on the absolute coordinates of the point of reference, because the results have to be transformed such that they appear as in the reference image.

Step 2: Rectify the search image

The inverse transformation is applied to the search image using the operator `affine_trans_image`. Figure 2.22d shows the resulting rectified image of a different CD. Undefined pixels are marked in gray.

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

Step 3: Extract the numbers

The serial number is positioned correctly within the original ROI and can be extracted by blob analysis without problems. Figure 2.22e shows the result, which could then, e.g., be used as the input for OCR.

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

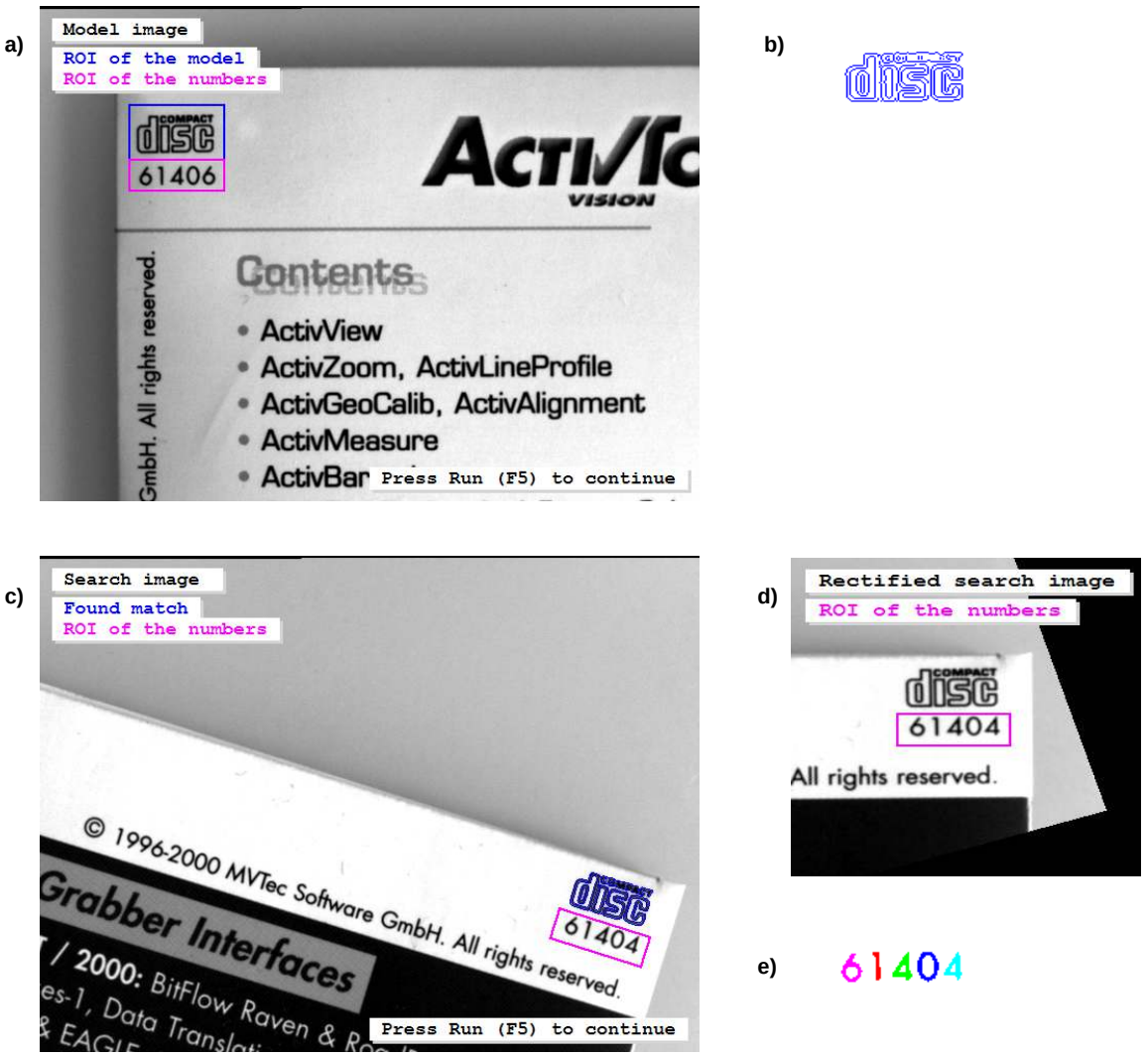


Figure 2.22: Rectify the search results: a) ROIs for the model and for the number extraction; b) the model; c) found model and number ROI at matched position; d) rectified search image (only relevant part shown); e) extracted numbers.

Unfortunately, the operator `affine_trans_image` transforms the full image even if its domain was restricted with the operator `reduce_domain`. In a time-critical application it may therefore be necessary to crop the search image before transforming it.

Image Cropping

In the following, the steps needed for the image cropping are described. Additionally, they are visualized in figure 2.23.

Step 1: Crop the search image

First, the smallest axis-parallel rectangle surrounding the transformed number ROI is computed using the operator `smallest_rectangle1`. Then, the search image is cropped to this part with `crop_rectangle1`. Figure 2.23b shows the resulting image overlaid on a gray rectangle to facilitate the comparison with the subsequent images.

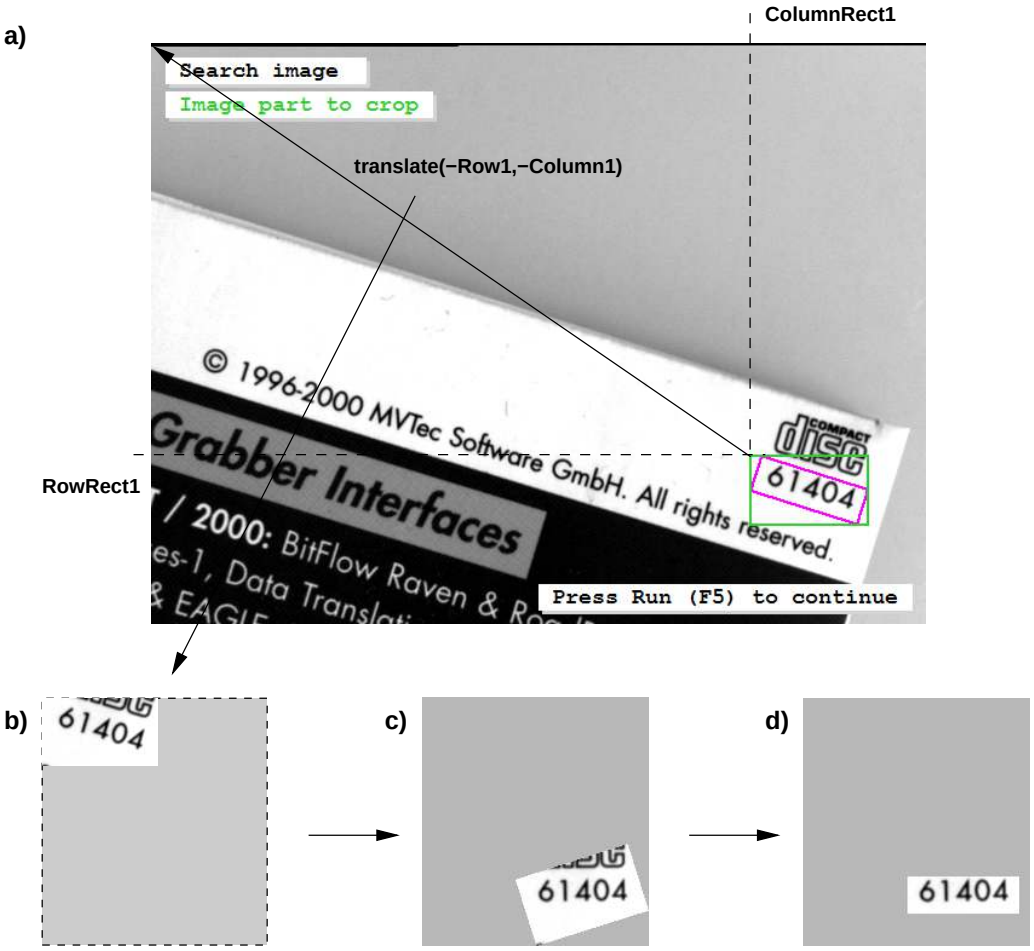


Figure 2.23: Rectifying only a part of the search image: a) smallest image part containing the ROI; b) cropped search image; c) result of the rectification; d) rectified image reduced to the original number ROI.

```
affine_trans_region (NumberROI, NumberROIAtNewPosition, \
                     MovementOfObject, 'nearest_neighbor')
smallest_rectangle1 (NumberROIAtNewPosition, RowRect1, ColumnRect1, \
                     RowRect2, ColumnRect2)
crop_rectangle1 (SearchImage, CroppedSearchImage, RowRect1, ColumnRect1, \
                 RowRect2, ColumnRect2)
```

Step 2: Create an extended affine transformation

In fact, the cropping can be interpreted as an additional affine transformation, in particular, as a translation by the negated coordinates of the upper left corner of the cropping rectangle (see figure 2.23a). We therefore add this transformation to the transformation that describes the movement of the object using the operator `hom_mat2d_translate`. Then, we invert this extended transformation with the operator `hom_mat2d_invert`.

```
hom_mat2d_translate (MovementOfObject, -RowRect1, -ColumnRect1, \
                     MoveAndCrop)
hom_mat2d_invert (MoveAndCrop, InverseMoveAndCrop)
```

Step 3: Transform the cropped image

Using the inverted extended transformation, the cropped image can easily be rectified with the operator `affine_trans_image` (figure 2.23c) and then be reduced to the original number ROI (figure 2.23d) in order to extract the numbers.

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

2.4.4 Use the Estimated 2D Scale

The transformations described in the previous section comprised only a translation and a rotation. For some matching approaches, the object that is searched for may be additionally scaled in the search image. For example, the perspective, deformable matching returns a projective transformation matrix (2D homography) or a 3D pose. That is, the scaling is returned implicitly and is already available as part of a transformation matrix. In contrast, for the shape-based matching the scaling is returned explicitly and is not yet part of a transformation matrix. Generally, scaling can be used similarly to the position and orientation. However, there is no convenient operator like `vector_angle_to_rigid` that creates an affine transformation including the scale. Therefore, the scaling must be added separately.

The HDevelop example program `%HALCONEXAMPLES%\hdevelop\Matching\Shape-Based\find_aniso_shape_model.hdev` shows how to add a scaling to a transformation matrix. The task is to find SMD capacitors that exhibit independent size changes in the row and column direction. That is, an anisotropic scaling is needed.



Figure 2.24: (left) synthetic template image; (right) result of anisotropic shape-based matching including the scaling factors in row and column direction.

Step 1: Create the model and access the model contours

First, to obtain a model, a synthetic template image containing a rectangle with rounded corners is created as described in section 2.1.3 on page 20 (see figure 2.24, left). From this image an anisotropic shape model is derived with `create_aniso_shape_model`. The model contours that will be projected into the search image after the actual matching are queried with `get_shape_model_contours`.

```
gen_contour_polygon_rounded_xld (Contour, [50,100,100,50,50], [50,50,150, \
                                 150,50], [6,6,6,6,6], 1)
gen_image_const (Image, 'byte', 200, 150)
paint_xld (Contour, Image, ImageModel, 128)
create_aniso_shape_model (ImageModel, 'auto', -rad(10), rad(20), 'auto', \
                          0.9, 1.7, 'auto', 0.9, 1.1, 'auto', 'none', \
                          'use_polarity', 'auto', 20, ModelID)
get_shape_model_contours (ModelContours, ModelID, 1)
```

Step 2: Find the model

Then, applying the actual matching with `find_aniso_shape_model`, several instances of the model are found in the search images.

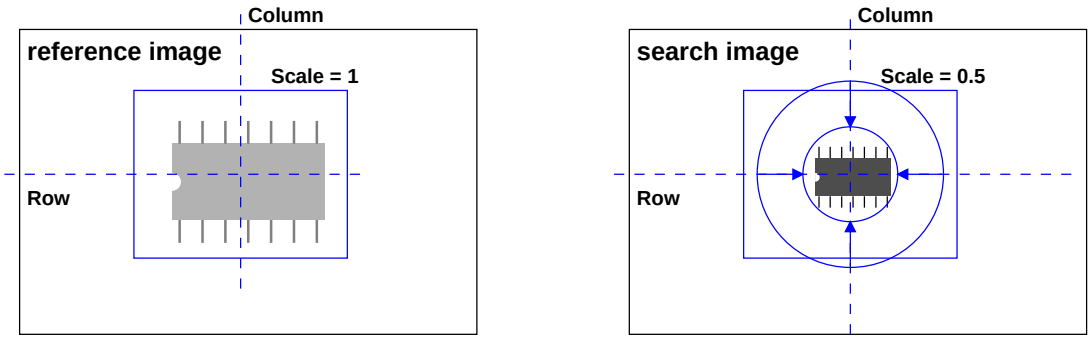


Figure 2.25: The center of the ROI acts as the point of reference for the scaling.

```
read_image (Image, 'smd/smd_capacitors_' + J$'02d')
find_aniso_shape_model (Image, ModelID, -rad(10), rad(20), 0.9, 1.7, \
                        0.9, 1.1, 0.7, 0, 0.5, 'least_squares', 0, 0.8, \
                        Row, Column, Angle, ScaleR, ScaleC, Score)
```

Step 3: Transform the model contours

For each found model instance, a transformation matrix is created with `hom_mat2d_identity`. Then, the scaling is added with `hom_mat2d_scale`. There, the previously created transformation matrix, the scaling factors that were returned by the matching, and the point of reference for the scaling are inserted. In this case, the scaling factors are different for row and column direction. In case of a uniform scaling, both values would be the same. Generally, similarly to the rotation (compare section 2.4.3 on page 32), the scaling is performed around the center of the ROI – if you did not change the point of reference. This is depicted in figure 2.25 at the example of an ROI whose center does not coincide with the center of the IC. Within the HDevelop program, the scaling is performed "before" the translation and rotation. Thus, the point of reference for the scaling corresponds to the point of reference of the model contour, which is by default (0,0). If you perform the scaling "after" the translation and rotation, e.g., if you used `vector_angle_to_rigid` to create an affine transformation that includes the translation and rotation in one step, you have to use the position of the match as the point of reference. An example is `%HALCONEXAMPLES%\solution_guide\matching\multiple_scales.hdev`.

After adding the rotation and translation to the transformation matrix, the model contour is transformed and projected to the corresponding SMD capacitor in the search image using `affine_trans_contour_xld` (see figure 2.24 on page 41, right).

```
for I := 0 to Num - 1 by 1
    hom_mat2d_identity (HomMat2D)
    hom_mat2d_scale (HomMat2D, ScaleR[I], ScaleC[I], 0, 0, HomMat2D)
    hom_mat2d_rotate (HomMat2D, Angle[I], 0, 0, HomMat2D)
    hom_mat2d_translate (HomMat2D, Row[I], Column[I], HomMat2D)
    affine_trans_contour_xld (ModelContours, ContoursTrans, HomMat2D)
endfor
```

Note that you can also call `get_hom_mat2d_from_matching_result`, which is a convenient procedure that encapsulates the steps described above in order to determine the wanted transformation matrix.

```
for I := 0 to Num - 1 by 1
    get_hom_mat2d_from_matching_result (Row[I], Column[I], Angle[I], \
                                        ScaleR[I], ScaleC[I], HomMat2D)
    affine_trans_contour_xld (ModelContours, ContoursTrans, HomMat2D)
endfor
```

2.4.5 Use the Estimated 2D Homography

When applying one of the uncalibrated perspective matching approaches, in contrast to the pure 2D matching approaches no positions, orientations, and scales but projective transformation matrices are returned. These can

be used to apply a projective transformation, e.g., to visualize the matching result by overlaying a structure of the reference image on the match in the search image. Note that different HALCON structures like pixels, regions, images, or XLD contours can be transformed that way (see also section 2.4.2.2 on page 31).

Depending on the selected matching approach, typically different structures are transformed. For the uncalibrated perspective deformable matching, e.g., the model contour is transformed to visualize the match, whereas for the uncalibrated descriptor-based matching the region that was used to create the model may be transformed and visualized.

An example for a projective transformation in the context of an uncalibrated **perspective deformable matching** is given in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Traffic-Monitoring\detect_road_signs.hdev` (see figure 2.26).

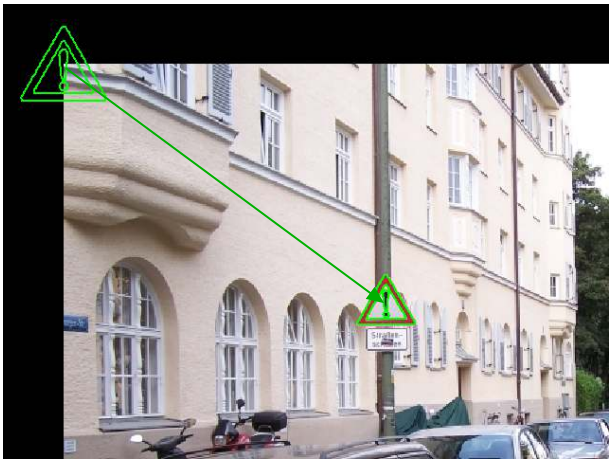


Figure 2.26: The contours of the model are projected into the search image.

Step 1: Find the perspective deformable model

There, the model of a road sign is searched in an image using `find_planar_uncalib_deformable_model`, which returns the projective transformation matrix `HomMat2D`.

```
find_planar_uncalib_deformable_model (ImageChannel, Models[Index2], 0, \
                                      0, ScaleRMin[Index2], \
                                      ScaleRMax[Index2], \
                                      ScaleCMin[Index2], \
                                      ScaleCMax[Index2], 0.85, 1, 0, 2, \
                                      0.4, [], [], HomMat2D, Score)
```

Step 2: Transform the model contours

After the matching, the model contours are queried from the model with `get_deformable_model_contours`. The returned contours are by default positioned at the origin of the reference image. Thus, to visualize them at the position of the match, the contour must be transformed using the returned projective transformation matrix. This transformation is applied with `projective_trans_contour_xld`.

```
get_deformable_model_contours (ModelContours, Models[Index2], 1)
projective_trans_contour_xld (ModelContours, ContoursProjTrans, \
                              HomMat2D)
dev_display (ContoursProjTrans)
```

An example for a projective transformation in the context of an uncalibrated **descriptor-based matching** is the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\detect_brochure_pages.hdev` (see figure 2.27).

Step 1: Find the descriptor model

There, the projective transformation matrix `HomMat2D` is returned by `find_uncalib_descriptor_model`.

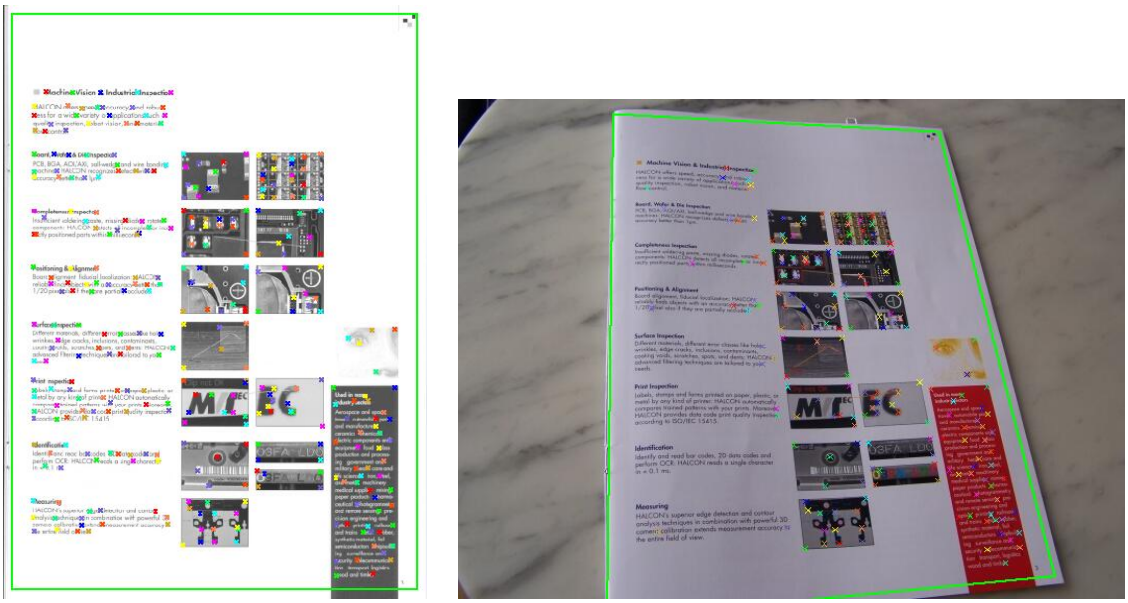


Figure 2.27: (left) Model brochure page with ROI and interest points; (right) search image with projected interest points and ROI.

```
find_uncalib_descriptor_model (ImageGray, ModelIDs[Index2], \
                               'threshold', 600, \
                               ['min_score_descr', \
                              'guided_matching'], [0.003,'on'], \
                               0.25, 1, 'num_points', HomMat2D, \
                               Score)
```

Step 2: Access and visualize the interest points

The interest points are queried with `get_descriptor_model_points`. In contrast to the contours of a perspective deformable model, which can only be queried for the model, the points of the descriptor model can also be queried for the specific match (`Set` set to `'search'`). Thus, they do not have to be transformed anymore.

```
get_descriptor_model_points (ModelIDs[Index2], 'search', 0, Row, \
                             Col)
gen_cross_contour_xld (Cross, Row, Col, 6, 0.785398)
```

Step 3: Transform the model region and its corner points

Projective transformations can be used if, e.g., the transformed region that was used for the creation of the model should be visualized for the match as well or if the transformed corner points of the region are needed to check the angle between edges of the region.

```
projective_trans_region (Rectangle, TransRegion, HomMat2D, \
                         'bilinear')
projective_trans_pixel (HomMat2D, RowRoi, ColRoi, RowTrans, \
                        ColTrans)
angle_ll (RowTrans[2], ColTrans[2], RowTrans[1], ColTrans[1], \
          RowTrans[1], ColTrans[1], RowTrans[0], ColTrans[0], \
          Angle)
Angle := deg(Angle)
if (Angle > 70 and Angle < 110)
    dev_display (TransRegion)
    dev_display (Cross)
endif
```

2.4.6 Use the Estimated 3D Pose

When applying one of the calibrated perspective matching approaches, 3D poses are returned that describe the relation between the model and the found model instance in world coordinates. To use such a pose, e.g., to visualize the matching result by overlaying a structure of the reference image on the match in the search image, a 3D affine transformation is needed (see also section 2.4.2.3 on page 32).

Depending on the selected matching approach, different structures from the reference image may be needed in the search image. For the calibrated perspective, deformable matching typically the contour of the model is used to visualize the match. That is, the contour must be transformed from the reference image to the search image. For the calibrated descriptor-based matching, the interest points of the model can be queried directly for the search result, i.e., there is no need for a transformation of the model representation. However, a transformation may be needed if the region that was used to create the model should be transformed as well.

An example for a 3D affine transformation in the context of a calibrated **perspective deformable matching** is given in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\locate_car_door.hdev` (see figure 2.28). There, a part of a car door is searched in an image using `find_planar_calib_deformable_model`, which returns the 3D pose `Pose`.

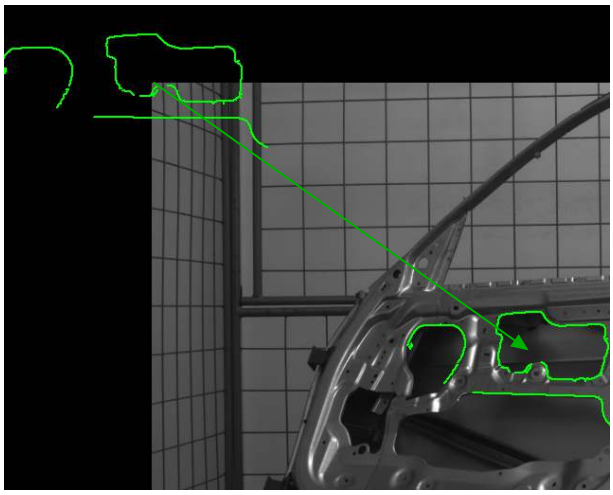


Figure 2.28: The contours of the model are projected into the search image.

```
find_planar_calib_deformable_model (ImageReducedSearch, ModelID, -0.1, \
                                    0.2, 0.6, 1.0, 0.6, 1.0, 0.7, 1, 1, \
                                    0, 0.7, [], [], Pose, CovPose, \
                                    Score)
```

For the visualization of the match, the model contour should be overlaid on the match in the search image. This transformation is realized in three steps.

Step 1: Preprocessing for visualization purposes

First, the operator `set_deformable_model_param` is used to set the coordinate system in which the contours are returned to `'world'`. When calling the operator `get_deformable_model_contours` the contours are returned in the world coordinate system (WCS). In this example, this step was realized before the actual matching.

```
set_deformable_model_param (ModelID, \
                            'get_deformable_model_contours_coord_system', \
                            'world')
get_deformable_model_contours (ModelContours, ModelID, 1)
```

Step 2: Apply a 3D affine transformation to the points of the contour

Within the WCS, a 3D affine transformation transforms the individual points of the contour according to the 3D pose that is returned by the matching. Before applying this transformation, the 3D pose must be converted with `pose_to_hom_mat3d` into a 3D homogeneous transformation matrix `HomMat3D`.

```
for Index1 := 0 to |Score| - 1 by 1
    tuple_select_range (Pose, Index1 * 7, ((Index1 + 1) * 7) - 1, \
                        PoseSelected)
    pose_to_hom_mat3d (PoseSelected, HomMat3D)
```

Additionally, as 3D affine transformations cannot be applied to 2D contours, the contour, which is now available in the x-y-plane of the WCS, must be split into points. This is realized by the operator `get_contour_xld`, which returns a tuple for the x coordinates and a tuple for the y coordinates. To obtain 3D coordinates, a tuple for the z coordinates with the same number of elements is created, for which all values are `0`.

```
    gen_empty_obj (FoundContour)
    for Index2 := 1 to NumberContour by 1
        select_obj (ModelContours, ObjectSelected, Index2)
        get_contour_xld (ObjectSelected, Y, X)
        Z := gen_tuple_const(|X|,0.0)
```

The 3D points are now transformed by the 3D affine transformation using `affine_trans_point_3d`.

```
        affine_trans_point_3d (HomMat3D, X, Y, Z, Xc, Yc, Zc)
```

Step 3: Project the points into the search image and reconstruct the contour

Then, the transformed 3D points are projected into the search image using `project_3d_point` and the contour is reconstructed with `gen_contour_polygon_xld`.

```
        project_3d_point (Xc, Yc, Zc, CamParam, R, C)
        gen_contour_polygon_xld (ModelWorld, R, C)
        concat_obj (FoundContour, ModelWorld, FoundContour)
    endfor
    dev_display (FoundContour)
endfor
```

An example for a 3D affine transformation in the context of a calibrated **descriptor-based matching** is given in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\locate_cookie_box.hdev` that is described in more detail in section 3.6.1 on page 108. There, selected image points, in particular the corner points of the rectangular ROI, which was used to build the descriptor model, are transformed into the WCS using `image_points_to_world_plane`.

2.4.7 About the Score

The scores returned by the different matching approaches have different meanings.

For the correlation-based matching, the score is determined by a normalized cross correlation between a pattern and the image.

For the approaches that use contours, in particular for the shape-based matching and the local or perspective deformable matching the score is a number between `0` and `1` that approximately shows how much of the model is visible in the search image. That is, if half of a model is occluded, the score can not be larger than `0.5`. Note that besides the pure number of corresponding contour points further influences on the score exist, which comprise, e.g., the orientation of the contour (see also section 2.1.2.2 on page 19 and section 3.2.4.3 on page 64).

For the component-based matching, the score has a similar meaning, but here, two types of scores have to be distinguished. On one hand, the score for the individual components is returned. Here, again the score shows primarily how much of the component is visible in the image. On the other hand, a score for the whole component model is returned. This is determined via the weighted mean of the score values for the individual components. The weighting is performed according to the number of model points within the respective component.

For the descriptor-based matching different meanings for the score value(s) are available, depending on the selected score type (see section 3.6.3.2 on page 111).

Chapter 3

The Individual Approaches

Now, we go deeper into the individual matching approaches, in particular, for

- the correlation-based matching (section 3.1),
- the shape-based matching (section 3.2 on page 52),
- the component-based matching (section 3.3 on page 74),
- the local deformable matching (section 3.4 on page 89),
- the perspective deformable matching (section 3.5 on page 99), and
- the descriptor-based matching (section 3.6 on page 107).

3.1 Correlation-Based Matching

Correlation-based matching is based on gray values. This approach uses a normalized cross correlation to evaluate the correspondence between a model and a search image. It is quite fast and can compensate both additive as well as multiplicative variations in illumination. In contrast to the shape-based matching, also objects with slightly changing shapes, lots of texture, or objects in blurred images (contours vanish in blurred images, e.g., because of defocus) can be found.

The following sections show

- a first example for a correlation-based matching (section 3.1.1),
- how to select an appropriate ROI to derive the template image from the reference image (section 3.1.2),
- how to create a suitable model (section 3.1.3 on page 49), and
- how to optimize the search (section 3.1.4 on page 50).

3.1.1 A First Example

In this section we give a quick overview of the matching process with correlation-based matching. To follow the example actively, start the HDevelop program `%HALCONEXAMPLES%\hdevelop\Matching\Correlation-Based\find_ncc_model_defocused.hdev`, which demonstrates the robustness of correlation-based matching against texture and defocus.

Step 1: Select the object in the reference image

First, inside the training image a region containing the object is created using `gen_rectangle1`. The center of this region is queried using `area_center`. It will be needed in a later step to overlay the results of the matching with the original region. Then, the image is reduced to the region of interest.

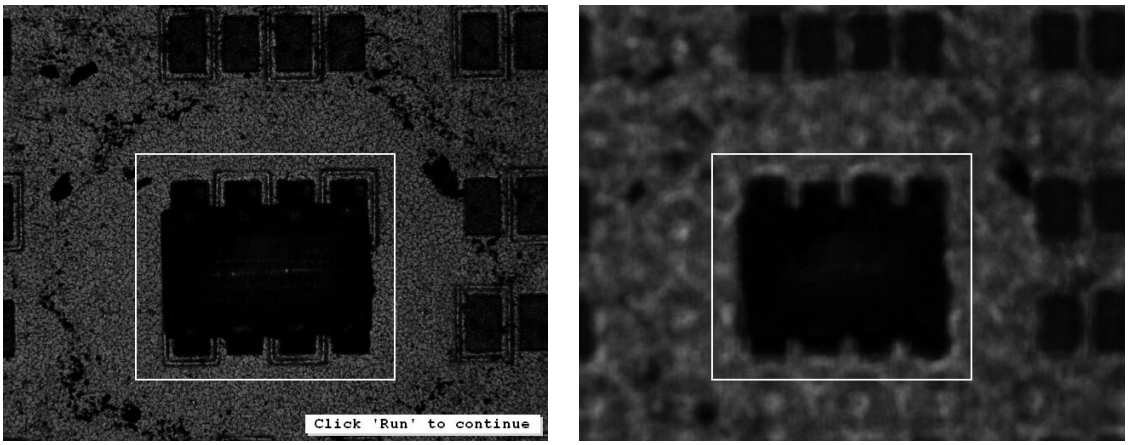


Figure 3.1: (left) reference image with the ROI that is used to create the model; (right) match of a defocused instance of the model.

```
read_image (Image, 'smd/smd_on_chip_05')
gen_rectangle1 (Rectangle, 175, 156, 440, 460)
area_center (Rectangle, Area, RowRef, ColumnRef)
reduce_domain (Image, Rectangle, ImageReduced)
```

Step 2: Create the model

The reduced image is used to create the NCC model with `create_ncc_model`. As a result, the operator returns a handle for the newly created model (`ModelID`), which can then be used to specify the model, e.g., in calls to the operators `find_ncc_model` or `find_ncc_models`.

```
create_ncc_model (ImageReduced, 'auto', 0, 0, 'auto', 'use_polarity', \
                  ModelID)
```

Step 3: Find the object again

Now, the search images are read in a loop and for each search image the NCC model is searched and overlaid by the region of the model using an affine transformation as described in section 2.4.2.1 on page 29. Note that the training was applied in a focused image and the search is applied in images with varying defocus. Nevertheless, the object instances are all found. Figure 3.1 shows the reference image and one of the defocused but found model instances.

```
for J := 1 to 11 by 1
    read_image (Image, 'smd/smd_on_chip_' + J$'02')
    find_ncc_model (Image, ModelID, 0, 0, 0.5, 1, 0.5, 'true', 0, Row, \
                    Column, Angle, Score)
    dev_display (Image)
    dev_display_ncc_matching_results (ModelID, 'green', Row, Column, Angle, \
                                      0)
endfor
```

3.1.2 Select the Model ROI

As a first step of the correlation-based matching, the region of interest that specifies the template image must be selected as described, e.g., in section 2.1.1 on page 17. This region can have an arbitrary shape, i.e., it can also contain holes or consist of several parts that are not connected. Thus, it is possible to create an ROI that contains only “robust” parts of the object. In most cases, the ROI must be selected so that it contains also some pixels outside of the object of interest, as the immediate surroundings (neighborhood) of the object are needed to obtain the model. But if the object itself contains enough structure to be recognized independently from its outline, it can also be selected smaller than the object. Then, the object can be found also in front of different backgrounds. Note that you can speed up the later search using a subsampling (see section 3.1.3.1). For that, the ROI should not be

too “thin”, because otherwise it vanishes at higher pyramid levels! As a rule of thumb, you are on the safe side if an ROI is $2^{NumLevels-1}$ pixels wide. That is, a width of 8 pixels allows to use 4 pyramid levels. After having selected a suitable ROI, the reduced image is used as template image for the creation of the model.

3.1.3 Create a Suitable NCC Model

Having derived the template image from the reference image, the NCC model can be created with the operator `create_ncc_model`. Here, we will take a closer look at how to adjust the corresponding parameters. In particular, you can

- use a subsampling to speed up the search by adjusting the parameter `NumLevels` (section 3.1.3.1),
- allow a specific range of orientation by adjusting the parameters `AngleExtent`, `AngleStart`, and `AngleStep` (section 3.1.3.2), and
- specify which pixels are compared with the model in the later search, i.e., specify the polarity of the object by adjusting the parameter `Metric` (section 3.1.3.3).

For the parameters `NumLevels` and `AngleStep`, you can let HALCON suggest values automatically. This can be done either by setting the parameters within `create_ncc_model` to the value `'auto'`; then, if you need to know the values, you can query them using `get_ncc_model_params`. Or you apply `determine_ncc_model_params` before you create the model. Then, you get an estimation of the automatically determined values as suggestion so that you can still modify them for the actual creation of the model. Note that both approaches return only approximately the same values and the values returned by `create_ncc_model` are more precise.

Note that after the creation of the model, the model can still be modified. In section 3.1.3.4 the possibilities for the inspection and modification of an already created model are shown.

3.1.3.1 Use Subsampling to Speed Up the Search (`NumLevels`)

To speed up the matching process, subsampling can be used (see also section 2.3.2 on page 25). There, an 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. You can specify how many pyramid levels are used via the parameter `NumLevels`. We recommend to **let HALCON select a suitable value itself** by specifying the value `'auto'`. You can then query the used value via the operator `get_ncc_model_params`.

3.1.3.2 Allow a Range of Orientation (`AngleExtent`, `AngleStart`, `AngleStep`)

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: radians). We recommend to limit the allowed range of rotation as much as possible in order to speed up the search process. During the matching process, the model is searched for in different angles within the allowed range, at steps specified with the parameter `AngleStep`. If you select the value `'auto'`, HALCON automatically chooses an optimal step size to obtain the highest possible accuracy by determining the smallest rotation that is still discernible in the image. In section 3.2.3.3 on page 58 tips for the selection of values for all three parameters are given for shape-based matching. These tips are also valid for correlation-based matching.

3.1.3.3 Specify how Gray Values are Compared with the Model (`Metric`)

The parameter `Metric` lets you specify whether the polarity, i.e., the “direction” of the contrast must be observed. If you select 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, e.g., 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 and reduced robustness.

3.1.3.4 Inspect and Modify the NCC Model

To inspect the current parameter values of the model, you query them with `get_ncc_model_params`. This may be necessary if during the creation of the model an automatic parameter selection was used or if the model was created within another program, saved to file with `write_ncc_model`, and read from this file in the current program with `read_ncc_model`. Additionally, you can query the coordinates of the origin of the model using `get_ncc_model_origin`.

After the creation of the model and before you search the object in a search image, you can further modify the model. In particular, you can apply `set_ncc_model_param` to change individual parameters and `set_ncc_model_origin` to change the origin of the model. The latter is not recommended because the accuracy of the matching result may decrease, which is shown in more detail for shape-based matching in section 3.2.4.8 on page 68.

3.1.4 Optimize the Search Process

The actual matching is performed by the operators `find_ncc_model` or `find_ncc_models`. In the following, we show how to select suitable parameters for these operators to adapt and optimize a matching task. In particular, we show how to

- restrict the search space to a region of interest (section 3.1.4.1),
- restrict the search space by restricting the range of orientation via the parameters `AngleStart` and `AngleExtent` (section 3.1.4.2),
- restrict the search space to a specific amount of deviations from the model for the object, i.e., specify the similarity of the object via the parameter `MinScore` (section 3.1.4.3),
- search for multiple instances of the model by adjusting the parameters `NumMatches` and `MaxOverlap` (section 3.1.4.4),
- search for multiple models with just one operator call (section 3.1.4.5),
- specify the accuracy that is needed for the results by adjusting the parameter `SubPixel` (section 3.1.4.6), and
- restrict the number of pyramid levels (`NumLevels`) for the search process (section 3.1.4.7).

3.1.4.1 Restrict the Search to a Region of Interest

The obvious way to restrict the search space and thus speed up the matching is to apply the operator `find_ncc_model` not to the whole image but only to an ROI. The corresponding procedure is explained in more detail for shape-based matching in section 3.2.4.1 on page 63. For correlation-based matching you simply have to replace `find_shape_model` by `find_ncc_model`.

3.1.4.2 Restrict the Range of Orientation (`AngleStart`, `AngleExtent`)

When creating the model you already specified the allowed range of orientation (see section 3.1.3.2 on page 49). When calling the operator `find_ncc_model` you can further **limit** the range with the parameters `AngleStart` and `AngleExtent`. 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.

3.1.4.3 Specify the Similarity of the Object (`MinScore`)

The parameter `MinScore` specifies the minimum score a potential match must have to be returned as match. The score is a value for the quality of a match, i.e., for the correspondence, or "similarity", between the model and the search image. For correlation-based matching the score is obtained using the normalized cross correlation between the pattern and the image (for the formula, see the description of `find_ncc_model` in the Reference Manual). To speed up the search, the value of `MinScore` should be chosen as large as possible, but of course still as small as necessary for the success of the search.

3.1.4.4 Search for Multiple Instances of the Object (NumMatches, MaxOverlap)

To find multiple instances of a model in the search image, the maximum number of returned matches is selected via the parameter NumMatches. Additionally, the parameter MaxOverlap specifies to which degree the instances may overlap. In section 3.2.4.6 on page 66 the corresponding parameters for shape-based matching are explained in more detail. There, tips for the parameter selection are given that are valid also for correlation-based matching. Note that if multiple instances of the object are searched and found, the parameters Row, Column, Angle, and Score contain tuples. How to access these results is exemplarily shown for shape-based matching in section 3.2.5.1 on page 72.

3.1.4.5 Search for Multiple Models Simultaneously (ModelIDs)

If you are searching for instances of multiple models in a single image, you can of course call the operator find_ncc_model multiple times. But a faster alternative is to use the operator find_ncc_models. This operator expects 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 3.1.4.4), 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 array.
- 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 parameter NumMatches. By passing a tuple of values, you can specify for each model individually how many instances are to be found. The tuple [3,0], e.g., specifies to return the best three 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.

3.1.4.6 Specify the Needed Accuracy (SubPixel)

The parameter SubPixel specifies if the position and orientation of a found model instance is returned with pixel accuracy (SubPixel set to 'false') or with subpixel accuracy (SubPixel set to 'true'). As the subpixel accuracy is almost as fast as the pixel accuracy, we recommend to set SubPixel to 'true'.

3.1.4.7 Restrict the Number of Pyramid Levels (NumLevels)

The parameter NumLevels, which you already specified when creating the model, allows you to use a different (in most cases a more restrictive) value in the search process. By using the value 0 for NumLevels, the value specified when creating the model is used.

Optionally, NumLevels can contain a second value, so that you can specify not only the highest but also the lowest pyramid level used for the search. If the search is finished on a pyramid level that is higher than the first pyramid level, which corresponds to the original, full-sized image, the search becomes faster. On the other hand, the search is then also less robust and less accurate.

3.1.4.8 Set the NCC Model Parameter 'timeout' via set_ncc_model_param

If your application demands that the search must be carried out within a specific time, you can set the parameter 'timeout' with the operator set_ncc_model_param to specify the maximum period of time after which the search is guaranteed to terminate, i.e., you can make the search interruptible. But note that for an interrupted search no result is returned. Additionally, when setting the timeout mechanism, the runtime of the search may be increased by up to 10%.

3.2 Shape-Based Matching

The shape-based matching does not use the gray values of pixels and their neighborhood as template but describes the model by the shapes of contours. The model can also be extended by further constraints regarding neighboring edges. The following sections show

- a first example for a shape-based matching (section 3.2.1),
- how to select an appropriate ROI to derive the template image from the reference image (section 3.2.2 on page 54),
- how to create a suitable model (section 3.2.3 on page 55),
- how to optimize the search (section 3.2.4 on page 61),
- how to deal with the results that are specific for shape-based matching (section 3.2.5 on page 71), and
- how to adapt to a changed camera orientation (section 3.2.6 on page 74).

Note that for shape-based matching, an HDevelop Assistant is available, which helps you to configure and test the matching process with a few mouse clicks and to optimize parameters interactively to get the maximum matching speed and recognition rate. Additionally, it can be used to prepare the reference image before the model ROI is selected. How to use the Matching Assistant is described in the HDevelop User's Guide, section 7.3 on page 217.

3.2.1 A First Example

In this section we give a quick overview of the matching process with shape-based matching. To follow the example actively, start the HDevelop program %HALCONEXAMPLES%\solution_guide\matching\first_example_shape_matching.hdev, which locates the print on an IC.

Step 1: Select the object in the reference image

After grabbing the reference image 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, i.e., the template image, is created using the operator reduce_domain. The result is shown in figure 3.2.

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

Step 2: Create the model

With the operator create_shape_model, the model is created. 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 (ImageROI, ShapeModelImages, ShapeModelRegions, 8, 30)
create_shape_model (ImageROI, NumLevels, 0, rad(360), 'auto', 'none', \
                    'use_polarity', 30, 10, ModelID)
```

inspect_shape_model shows the effect of two parameters, in particular the number of pyramid levels on which the model is created (NumLevels) and the minimum contrast that object points must have to be included in the model (Contrast). As a result, the operator inspect_shape_model returns the model points on the selected pyramid levels as shown in figure 3.2. 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. For example, you can restrict the range of angles the object can assume (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, which 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 3.2.3 on page 55.

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 .NET, or C++ interface and call the operator via the class HShapeModel, no handle is returned because the instance of the class itself acts as your handle.

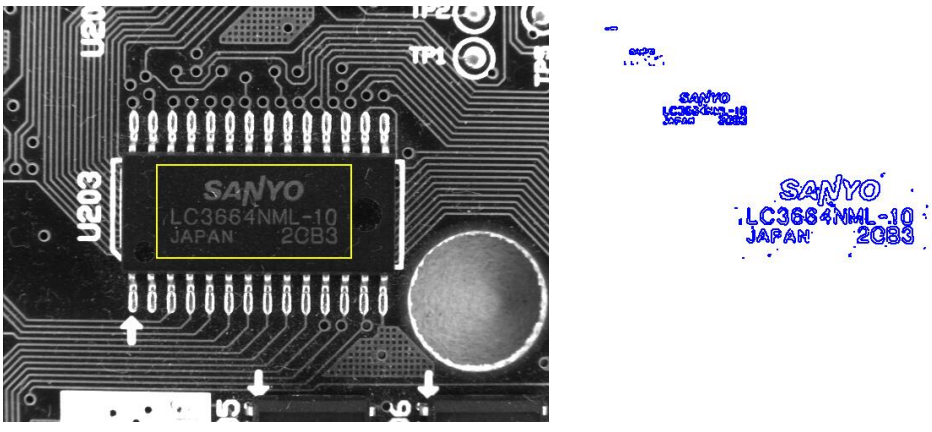
If not only the orientation but also the scale of the searched object is allowed to vary, you must use the operator create_scaled_shape_model or create_aniso_shape_model to create the model. Then, you can describe the allowed range of scale with parameters similar to those used for the range of angles.

Step 3: Find the object again

To find the object in a search image, all you need to do is call the operator find_shape_model. Figure 3.3 shows the result for one of the example images.

```
for i := 1 to 20 by 1
    grab_image (SearchImage, FGHandle)
    find_shape_model (SearchImage, ModelID, 0, rad(360), 0.7, 1, 0.5, \
                      'least_squares', 0, 0.7, RowCheck, ColumnCheck, \
                      AngleCheck, Score)
endfor
```

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 much of the model must be visible. A value of 0.5 approximately means that half of the contours of a 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,



① ②

Figure 3.2: (left) reference image with the ROI that specifies the object; (right) the internal model (4 pyramid levels).

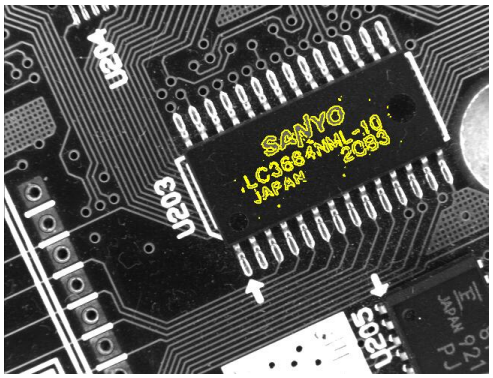


Figure 3.3: Finding the object in other images.

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 3.2.4 on page 61.

The operator `find_shape_model` returns the position and orientation of each found object instance in the parameters `Row`, `Column`, and `Angle`, and their corresponding `Score`.

If you use the operator `find_scaled_shape_model` or `find_aniso_shape_model` (after creating the model using `create_scaled_shape_model` or `create_aniso_shape_model`, respectively), additionally the scale of the found object is returned in `Scale` or `ScaleR` and `ScaleC`, respectively.

The following sections go deeper into the details of the individual steps of a shape-based matching and the parameters that have to be adjusted.

3.2.2 Select the Model ROI

As a first step of the shape-based matching, the region of interest that specifies the template image must be selected as described, e.g., in section 2.1.1 on page 17. This region can have an arbitrary shape, i.e., it can also contain holes or consist of several parts that are not connected. Thus, it is possible to create an ROI that contains only "robust" parts of the object. The ROI must be selected so that it contains all prominent structures of the object and also some pixels outside of them, as their immediate surroundings (neighborhood) are needed to obtain the model. Furthermore, you can speed up the later search using a subsampling (see section 3.2.3.2 on page 58). For that, the ROI should not be too "thin", because otherwise it vanishes at higher pyramid levels! As a rule of thumb, you are on the safe side if an ROI is $2^{NumLevels-1}$ pixels wide. That is, a width of 8 pixels allows to use 4 pyramid levels. After having selected a suitable ROI, the reduced image is used as template image for the creation of the model.

When using shape-based matching in the presence of clutter in the reference image, you can also use the operator `inspect_shape_model` to improve an interactively selected ROI by additional image processing. This is shown in the HDevelop example program `%HALCONEXAMPLES%\solution_guide\matching\process_shape_model.hdev`, which locates the arrows that are shown in Figure 3.4.

Step 1: Select the arrow

There, an initial ROI is created around the arrow, without trying to exclude clutter (see figure 3.4a).

```
gen_rectangle1 (ROI, 361, 131, 406, 171)
reduce_domain (ModelImage, ROI, ImageROI)
```

Step 2: Create a first model region

Then, this ROI is inspected via `inspect_shape_model`.

```
inspect_shape_model (ImageROI, ShapeModelImage, ShapeModelRegion, 1, 30)
```

Figure 3.4b 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.

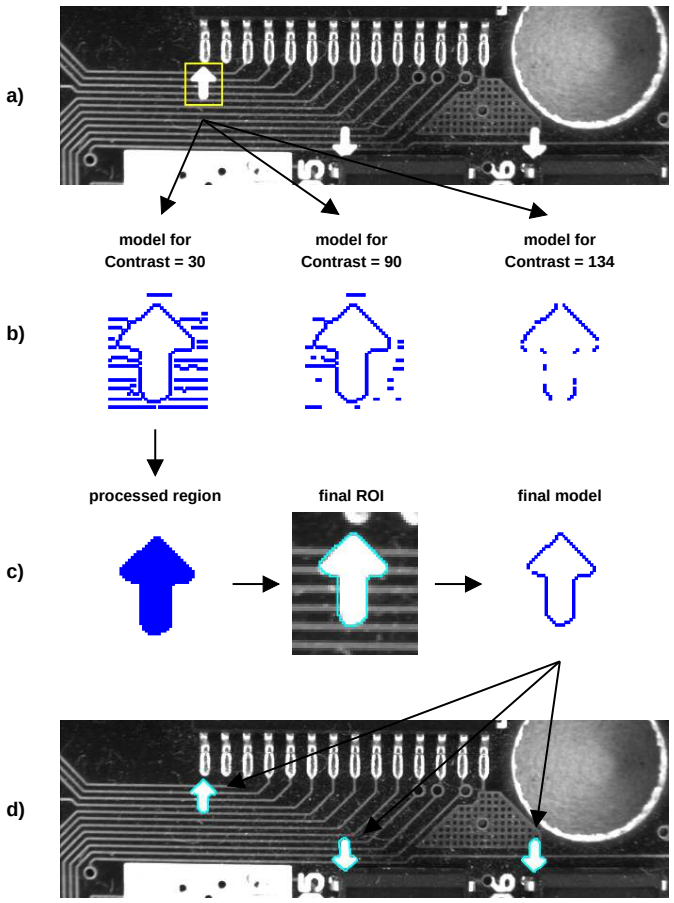


Figure 3.4: Processing the result of `inspect_shape_model`: a) interactive ROI; b) models for different values of `Contrast`; c) processed model region and corresponding ROI and model; d) result of the search.

Step 3: Process the model region

This problem can be solved 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 the 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.

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

Step 4: Create the final model

The obtained region is then used to create the model for a matching that locates all arrows successfully. Figure 3.4c shows the processed region, the corresponding region of interest, and the final model region.

```
create_shape_model (ImageROI, 3, 0, rad(360), 'auto', 'none', \
                    'use_polarity', 30, 15, ModelID)
```

3.2.3 Create a Suitable Shape Model

Having derived the template image from the reference image, the shape model can be created. Note that the shape-based matching consists of different methods to find the trained objects in images. Depending on the selected method, one of the following operators is used to create the model:

- `create_shape_model` creates a model for a simple shape-based matching that uses a template image to derive the model and which supports no scaling.

- create_shape_model_xld creates a model for a simple shape-based matching that uses an XLD contour to derive the model and which supports no scaling.
- create_scaled_shape_model creates a model for a shape-based matching that uses a template image to derive the model and which supports a uniform scaling.
- create_scaled_shape_model_xld creates a model for a shape-based matching that uses an XLD contour to derive the model and which supports a uniform scaling.
- create_aniso_shape_model creates a model for a shape-based matching that uses a template image to derive the model and which supports anisotropic scaling.
- create_aniso_shape_model_xld creates a model for a shape-based matching that uses an XLD contour to derive the model and which supports anisotropic scaling.

Note that if you derive your model from an XLD contour, after a first match, it is strongly recommended to determine the polarity information for the model with set_shape_model_metric (see section 3.2.3.5 on page 60 and section 2.1.3.2 on page 22 for details).

In the following, the described parameters belong to the operator create_scaled_shape_model if not stated otherwise.

As the name "shape-based 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 neighboring contrast exceeds a certain threshold. Typically, the points are part of the contours of the object (see, e.g., figure 3.2 on page 53). Here, we will take a closer look at how to adjust the corresponding parameters. In particular, you can

- specify which pixels are part of the model by adjusting the parameter Contrast (section 3.2.3.1),
- speed up the search by using a subsampling, i.e., by adjusting the parameter NumLevels, and by reducing the number of model points, i.e., by adjusting the parameter Optimization (section 3.2.3.2 on page 58),
- allow a specific range of orientation by adjusting the parameters AngleExtent, AngleStart, and AngleStep (section 3.2.3.3 on page 58),
- allow a specific range of scale by adjusting the parameters ScaleMin, ScaleMax, and ScaleStep or the corresponding parameters for anisotropic scaling (section 3.2.3.4 on page 59), and
- specify which pixels are compared with the model in the later search by adjusting the parameters MinContrast and Metric (section 3.2.3.5 on page 60).

Note that when adjusting the parameters you can also let HALCON assist you:

- Apply inspect_shape_model:

  Before creating the model, you can apply the operator inspect_shape_model to the template image to try different values for the parameters NumLevels and Contrast. The operator returns the resulting representation of a shape model with multiple pyramid levels and thus allows you to visually check if the object of interest is represented adequately by the model. If several combinations of parameter values do not lead to a satisfying representation of the object, maybe the template image, i.e., the model's ROI was not selected properly. How to select a suitable ROI is described in section 2.1 on page 17.

- Use automatic parameter suggestion:

  For many parameters you can let HALCON suggest suitable parameters. Then, you can either set the corresponding parameters to the value 'auto' within create_shape_model (or one of its equivalents), or you apply determine_shape_model_params to automatically determine parameters for a shape model from a template image and then decide individually if you use the suggested values for the creation of the shape model. Note that both approaches return only approximately the same values and create_shape_model (or one of its equivalents) returns the more precise values.

Note that after the creation of the model, the model can still be modified. In section 3.2.3.6 on page 61 the possibilities for the inspection and modification of an already created model are shown.

3.2.3.1 Specify Pixels that are Part of the Model (Contrast)

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 or one of its equivalents. 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., structures 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 3.5 shows an example. The task is to create a model for the outline of the pad. If the complete outline is selected, the model also contains clutter (figure 3.5a). If the clutter is removed, parts of the outline are missing (figure 3.5b).

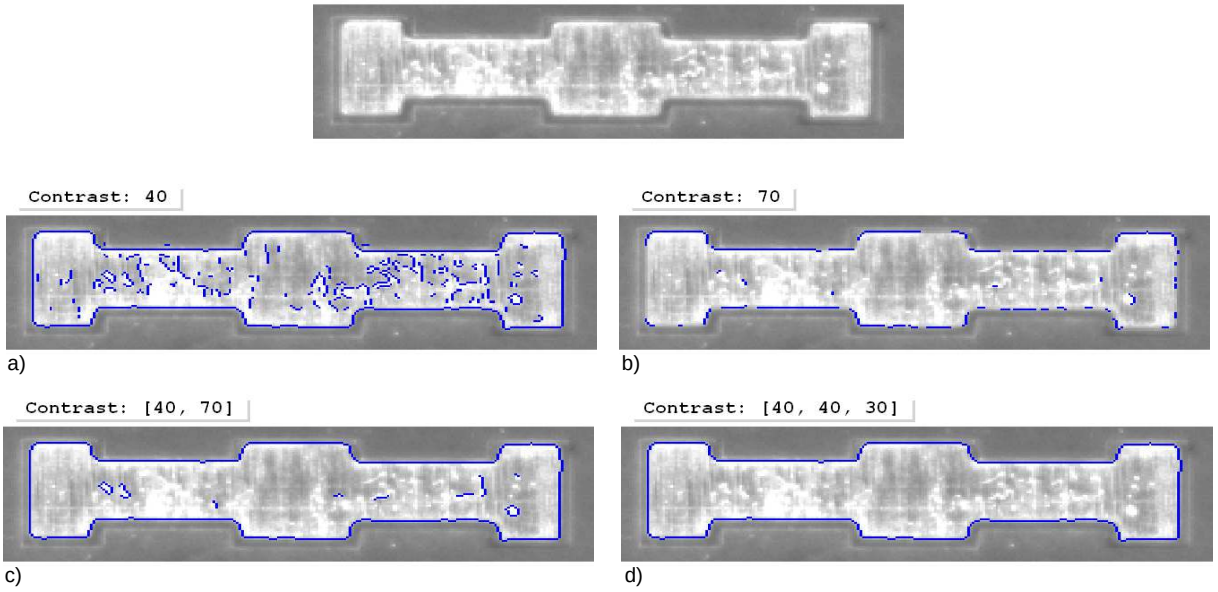


Figure 3.5: Selecting significant pixels via Contrast: a) complete object containing clutter; b) little clutter but incomplete object; c) hysteresis threshold; d) minimum contour size.

To solve this problem, the parameter Contrast provides two additional methods: a hysteresis thresholding and the 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 is locally low. Returning to the example with the pad: As you can see in figure 3.5c, with a hysteresis threshold you can create a model for the complete outline of the pad without clutter. The following line of code shows how to specify the two thresholds in a tuple:

```
inspect_shape_model (ImageReduced, ModelImages, ModelRegions, 2, [40,70])
```

The second method to remove clutter is to specify a minimum size, i.e., number of pixels, for the contour components. Figure 3.5d shows the result for the example task. The minimum size must be specified in the third element of the tuple. If you do not want to additionally use a hysteresis threshold, you have to set the first two elements to the same value:

```
inspect_shape_model (ImageReduced, ModelImages, ModelRegions, 2, [40,40,30])
```

Alternative methods to remove clutter are to modify the ROI as described in section 3.2.2 on page 54 or create a synthetic model (see section 2.1.3.1 on page 20).

Note that you can **let HALCON select suitable values itself** by specifying the value 'auto' for Contrast. If you want to specify some of the three contrast parameters and let HALCON determine the rest, please refer to the Reference Manual for detailed information.

3.2.3.2 Speed Up the Search using Subsampling and Point Reduction (NumLevels, Optimization)

To speed up the matching process, subsampling can be used (see also section 2.3.2 on page 25). There, an 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.

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.

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

A further reduction of model points can be enforced via the parameter Optimization. This may be useful to speed up the matching in the case of particularly large models. Again, we recommend to **specify the value 'auto' to let HALCON select a suitable value itself**. 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.

With an optional second value, you can specify whether the model is pregenerated completely for the allowed range of rotation and scale (see the following sections) or not. By default, the model is not pregenerated. You can pregenerate the model and thereby possibly speed up the matching process by passing 'pregeneration' as the second value of Optimization. Note, however, that if you allow large ranges of rotation and/or scaling, the memory requirements rise. Another effect is that the process of creating the model takes significantly more time. **In most cases, it is not recommended to pregenerate the model.**

Note that if you want to set the number of pyramid levels manually, you can inspect the template image pyramid using the operator inspect_shape_model, e.g., as shown in the HDevelop program %HALCONEXAMPLES%\solution_guide\matching\first_example_shape_matching.hdev. After the call to inspect_shape_model, 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 ▷ Zoom Window). 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.

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

Note that 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 parameter tuples start with the index 0!**

3.2.3.3 Allow a Range of Orientation (AngleExtent, AngleStart, AngleStep)

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: radians). Note that the range of rotation is defined relative to the reference image, i.e., a starting angle of 0 corresponds to the orientation the object has in the reference 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).

Note that you can further limit the allowed range when calling the operator `find_shape_model` or one of its equivalents during the search (see section 3.2.4.2 on page 64). Thus, if you want to reuse a model for different tasks requiring a different range of angles, you can 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° (see figure 3.6).

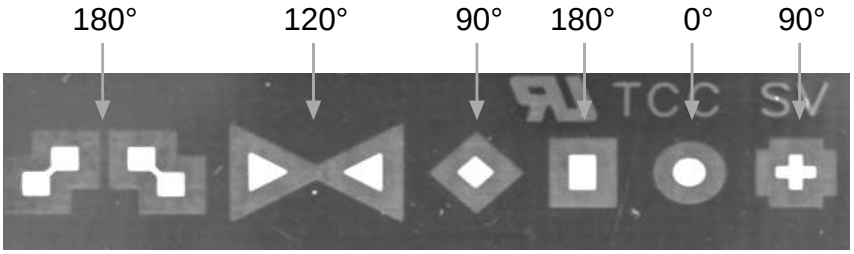


Figure 3.6: Suitable angle ranges for rotation symmetric objects.

If you pregenerate the model (see page 58), a large range of rotation also leads to high memory requirements. Thus, in this special case, it is recommended to limit the allowed range of rotation already during the creation of the model as much as possible in order to speed up the search process.

During the matching process, the model is searched for in different angles within the allowed range, at steps specified with the parameter `AngleStep`. If you select the value `'auto'`, 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 3.7: 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. For some models, the such estimated angle step size is still too large. In these cases, it is divided by 2 automatically.



Figure 3.7: 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** `'auto'`. 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!

3.2.3.4 Allow a Range of Scale (`Scale*Min`, `Scale*Max`, `Scale*Step`)

Similarly to the range of orientation, you can specify an allowed range of scale. You can allow for scaling in two forms:

- identical scaling in row and column direction (uniform scaling)
- different scaling in row and column direction (anisotropic scaling)

For a uniform scaling, you specify the range of scale with the parameters ScaleMin, ScaleMax, and ScaleStep of the operator create_scaled_shape_model. For anisotropic scaling, you use the operator create_aniso_shape_model instead, with six scale parameters instead of the three above.

Note that you can further limit the allowed range when calling the operator find_scaled_shape_model or find_aniso_shape_model (see section 3.2.4.2 on page 64). Thus, if you want to reuse a model for different tasks requiring a different range of scale, you can use a large range when creating the model and a smaller range for the search.

If you pregenerate the model (see page 58), a large range of scale also leads to high memory requirements. Thus, in this special case, it is recommended to limit the allowed range of scale already during the creation of the model as much as possible in order to speed up the search process.

☞ 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 'auto' for the parameter ScaleStep (or the equivalents for anisotropic scaling), 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 3.7 on page 59), 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. For some models, the such estimated scale step size is still too large. In these cases, it is divided by 2 automatically.

☞ The automatically determined scale step size is suitable for most applications. Therefore, **we recommend to select the value** 'auto'. 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!

3.2.3.5 Specify which Pixels are Compared with the Model (MinContrast, Metric)

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 **let HALCON select suitable values itself** by specifying the value 'auto' for MinContrast. As the value is estimated from the noise in the reference image, the automatic parameter selection is suitable only if the noise in the reference image is expected to be similar to the noise in the search images. If a synthetic model is used, which contains no noise, MinContrast is automatically set to 0 and must be set to a higher value manually. Note that you can change the value for MinContrast also in a later step using set_shape_model_param (see section 3.2.4.10 on page 69).

The parameter Metric lets you specify whether and how the polarity, i.e., the direction of the contrast must be observed (see figure 3.8). 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, e.g., 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

or brighter sub-objects lie. Please note however, that the recognition speed and the robustness may decrease dramatically in this mode, especially if you allowed a large range of rotation (see section 3.2.3.3 on page 58).

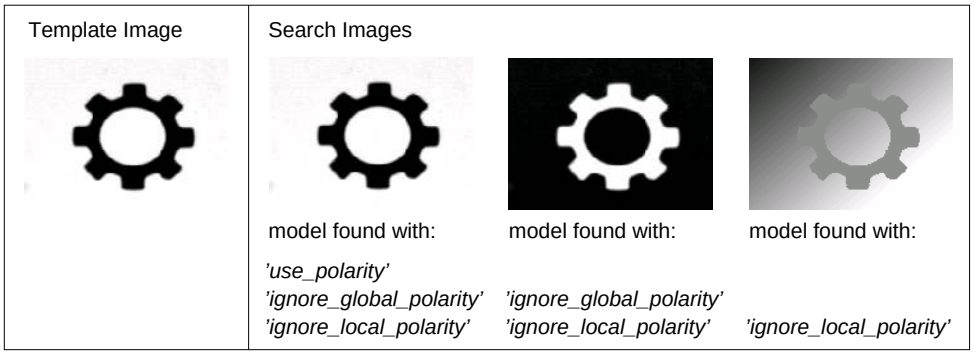


Figure 3.8: The parameter `Metric` specifies how to consider the polarity of the model.

If you select the value `'ignore_color_polarity'`, you can perform the matching in color images (or, more generally, in multi-channel images). An example is `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-2D\matching_multi_channel_yogurt.dev`.

If you created your **model from XLD contours**, there is no information about the polarity of the model available. Thus, when creating the model, the value of `Metric` must be set to `'ignore_local_polarity'`. When a first matching was successful, you can use the matching results to get the transformation parameters (using `vector_angle_to_rigid`) that are needed to project the contour onto the search image. Then, with the operator `set_shape_model_metric` you can determine the polarity of the first search image, which is used as training image for the polarity, and set the match metric to `'use_polarity'` or `'ignore_global_polarity'`. Using the value `'use_polarity'`, i.e., if the following search images have the same polarity as the training image, the search becomes faster and more robust. An example is `%HALCONEXAMPLES%\hdevelop\Matching\Shape-Based\create_shape_model_xld.dev`. **Note that `set_shape_model_metric` is only available for models that are created from XLD contours!**

3.2.3.6 Inspect and Modify the Shape Model

If you want to visually inspect an already created shape model, you can use `get_shape_model_contours` to get the XLD contours that represent the model in a specific pyramid level. Note that the XLD contour of the model is located at the origin of the image and thus a transformation may be needed for a proper visualization (see section 2.4.2 on page 29).

To inspect the current parameter values of a model, you query them with `get_shape_model_params`. This may be necessary if during the creation of the model an automatic parameter selection was used or if the model was created within another program, saved to file with `write_shape_model`, and read from this file in the current program with `read_shape_model`. Additionally, you can query the coordinates of the origin of the model using `get_shape_model_origin`.

After the creation of the model and before you search the object in a search image, you can further modify the model. In particular, you can apply `set_shape_model_param` to change individual parameters, `set_shape_model_clutter` to exclude matches containing edges in regions specified relative to the shape model contours, and `set_shape_model_origin` to change the origin of the model. Note that the latter expects not the absolute position of a new point of reference as parameters, but its *distance* to the default point of reference. That is, the value of the model's origin can also become negative, e.g., [-20, -40]. But please note that by modifying the point of reference, the accuracy of the estimated position may decrease (see section 3.2.4.8 on page 68). Thus, **if possible, the point of reference should not be changed.**

3.2.4 Optimize the Search Process

The actual matching is performed by one of the following operators:

- find_shape_model searches for instances of a single model and is used if the instances of the model may not vary in scale,
- find_shape_models simultaneously searches for instances of multiple models and is used if the instances of the model may not vary in scale,
- find_scaled_shape_model searches for instances of a single model and is used if a uniform scaling is allowed,
- find_scaled_shape_models simultaneously searches for instances of multiple models and is used if a uniform scaling is allowed,
- find_aniso_shape_model searches for instances of a single model and is used if different scaling factors in row and column direction are allowed, and
- find_aniso_shape_models simultaneously searches for instances of multiple models and is used if different scaling factors in row and column direction are allowed.

In the following, we show how to select suitable parameters for these operators to adapt and optimize a matching task. In particular, we show how to

- restrict the search space to a region of interest (section 3.2.4.1),
- restrict the search space by restricting the range of orientation and scale via the parameters AngleStart, AngleExtent, ScaleMin, and ScaleMax or the corresponding parameters for anisotropic scaling (section 3.2.4.2 on page 64),
- restrict the search space to a specific amount of allowed occlusions for the object, i.e., specify the visibility of the object via the parameter MinScore (section 3.2.4.3 on page 64),
- restrict the search to candidates that have no (or too faint) clutter edges within a region specified relative to the model contours (section 3.2.4.4 on page 65),
- specify the used search heuristics, i.e., trade thoroughness versus speed by adjusting the parameter Greediness (section 3.2.4.5 on page 65),
- search for multiple instances of the model by adjusting the parameters NumMatches and MaxOverlap (section 3.2.4.6 on page 66),
- search for multiple models simultaneously by adjusting the parameter ModelIDs (section 3.2.4.7 on page 66),
- specify the accuracy that is needed for the results by adjusting the parameter SubPixel (section 3.2.4.8 on page 68),
- restrict the number of pyramid levels (NumLevels) for the search process (section 3.2.4.9 on page 69),
- set the parameters 'timeout' and 'min_contrast' via the operator set_shape_model_param (section 3.2.4.10 on page 69), and
- optimize the matching speed (section 3.2.4.11 on page 70).

Note that many matching approaches can be used only to search a single instance of a single model in an image, whereas the shape-based matching additionally can be used to search for several instances of multiple models simultaneously.

3.2.4.1 Restrict the Search to a Region of Interest

The obvious way to restrict the search space is to apply the operator `find_shape_model` (or one of its equivalents) not to the whole image but only to an ROI. Figure 3.9 shows such an example. The reduction of the search space can be realized in a few lines of code.

Step 1: Create a region of interest

First, you create a region, e.g., with the operator `gen_rectangle1` (see section 2.1.1 on page 17 for more ways to create regions).

```
Row1 := 141
Column1 := 159
Row2 := 360
Column2 := 477
gen_rectangle1 (SearchROI, Row1, Column1, Row2, Column2)
```

Step 2: Restrict the search to the region of interest

Then, each search image is reduced to this ROI using the operator `reduce_domain`. In this example, the search speed is almost doubled using this method.

```
for i := 1 to 20 by 1
    grab_image (SearchImage, FGHandle)
    reduce_domain (SearchImage, SearchROI, SearchImageROI)
    find_shape_model (SearchImageROI, ModelID, 0, rad(360), 0.7, 1, 0.5, \
                      'interpolation', 0, 0.7, RowCheck, ColumnCheck, \
                      AngleCheck, Score)
endfor
```

Note that by restricting the search to an ROI 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 2.1.2 on page 18). 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$\times$30. In order to assure a correct boundary treatment on higher pyramid levels, we recommend to enlarge the ROI by $2^{NumLevels-1}$ pixels in each direction. Thus, if you specified `NumLevels` = 4, you can restrict the search to an ROI of the size 36$\times$46.

Please note that even if you modified the point of reference using `set_shape_model_origin` (which is not recommended), 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 point of reference.

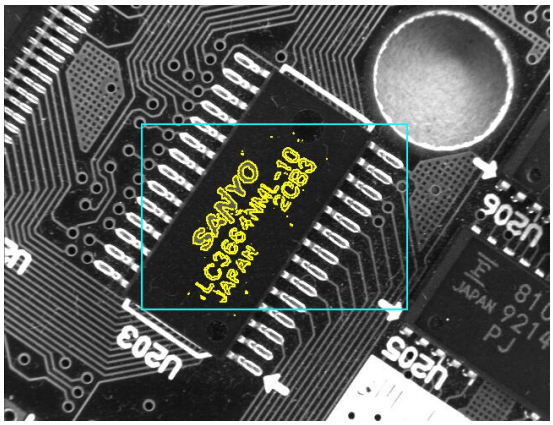


Figure 3.9: Searching in a region of interest.

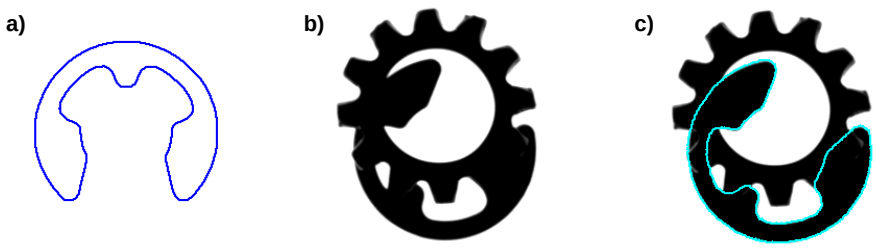


Figure 3.10: Searching for partly occluded objects: a) model of the security ring; b) search result for `MinScore` = 0.8; c) search result for `MinScore` = 0.7.

3.2.4.2 Restrict the Range of Orientation and Scale (`AngleStart`, `AngleExtent`, `ScaleMin`, `ScaleMax`)

When creating the model you already specified the allowed range of orientation and scale (see section 3.2.3.3 on page 58 and section 3.2.3.4 on page 59). When calling the operator `find_shape_model` or one of its equivalents you can further limit these ranges with the parameters `AngleStart`, `AngleExtent`, `ScaleMin`, and `ScaleMax` (or the corresponding scale parameters for anisotropic scaling). 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 as well.

Note that, if the scale range for the creation of the model is larger than the range used for the search, the model might be found in the search image even if it is slightly outside of the more restricted scale range.

3.2.4.3 Specify the Visibility of the Object (`MinScore`)

With the parameter `MinScore` you can specify how much of the model must be visible. A typical use of this mechanism is to allow a certain degree of occlusion as demonstrated in figure 3.10: 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 primarily 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:

- Parts of the object's contour are occluded, e.g., as depicted in figure 3.10.

  Please note that by default **objects are not found if they are clipped at the image border**. This behavior can be changed with `set_system('border_shape_models','true')`. An example is `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-2D\matching_image_border.dev`.

  Note that the runtime of the search will increase in this mode.

- Parts of the contour have a contrast lower than specified in the parameter `MinContrast` when creating the model (see section 3.2.3.5 on page 60).

- The polarity of the contrast changes globally or locally (see section 3.2.3.5 on page 60).

- If the object is deformed, which includes also the case that the camera observes the scene under an oblique angle, parts of the contour may be visible but appear at an incorrect position and therefore do not fit the model anymore. Note that deformed objects might be found if you set the parameter `'SubPixel'` to `'max_deformation'` (see section 3.2.4.8 on page 68). Additionally, deformed or defocused objects might be found if the increased tolerance mode is activated. For that the lowest pyramid level has to be specified negatively within `NumLevels`. Then, the matches on the lowest pyramid level that still provides matches are returned. How to handle the specific case of perspective deformations that occur because of an oblique camera view is described in section 3.2.6 on page 74.

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:

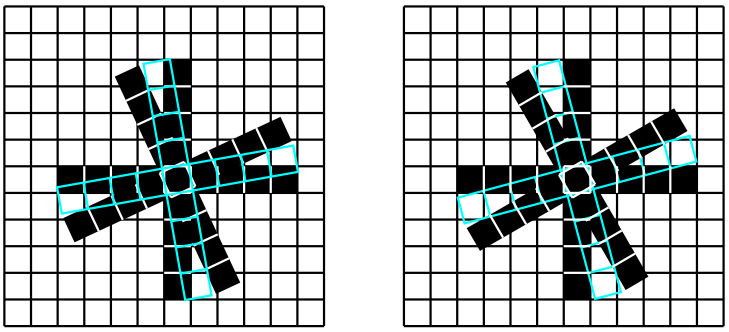


Figure 3.11: The effect of a large `AngleStep` on the matching.

- As described in section 3.2.3.3 on page 58, 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 3.11 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` and its variants for anisotropic scaling (see section 3.2.3.4 on page 59).

- Another stumbling block lies in the use of an image pyramid which was introduced in section 3.2.3.2 on page 58: 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`. Note that if the matches are not tracked to the lowest pyramid level it might happen that instances with a score slightly below `MinScore` are found.

Recommendation: The higher `MinScore`, the faster the search! ☜

3.2.4.4 Restrict the Search by Using Clutter Parameters

In some cases, although multiple matches with high `Score` values are found within a search image, not all of them are wanted as results. For example, in Figure 3.12, although only the structures consisting of eight dots are searched for, more instances are found. In order to get only the wanted matches, the instances have to be distinguished. This can be done considering the absence of edges in the neighborhood of the object as a distinctive feature.

To restrict the search to candidates that have no (or too faint) edges within a region specified relative to the model contours, you can use `set_shape_model_clutter`, which defines the clutter region and its respective parameters. For further information how to adjust the clutter parameters, please refer to the description of the operator in the Reference Manual.

3.2.4.5 Trade Thoroughness vs. Speed (`Greediness`)

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

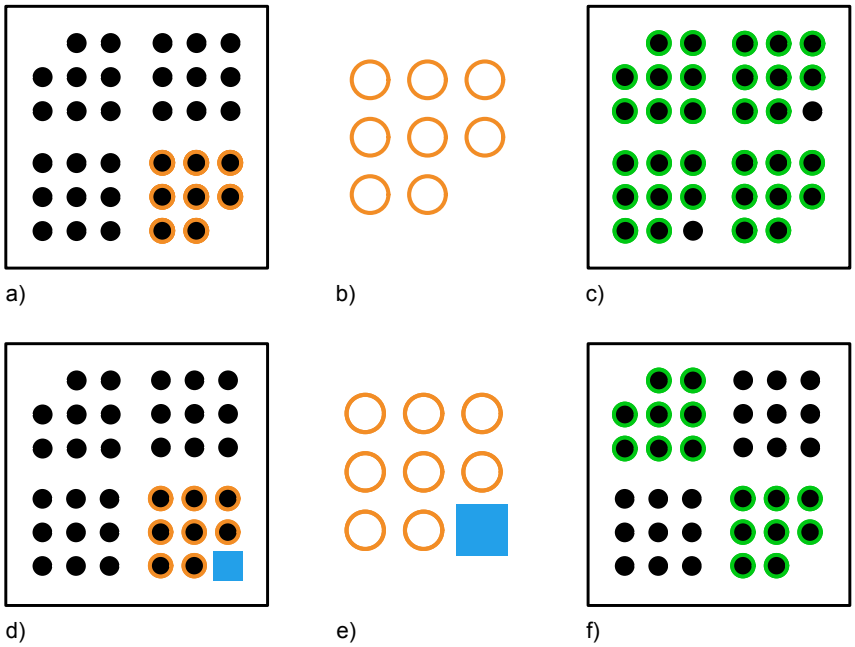


Figure 3.12: Extending a shape model by a clutter region to retrieve only the wanted results: a) select shape model (orange) from image; b) extracted shape model; c) matching result (green) without the use of clutter parameters; d) additionally define clutter region (blue); e) shape model (orange), extended by a clutter region (blue); f) matching result (green) with the use of clutter 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.

3.2.4.6 Search for Multiple Instances of the Object (`NumMatches`, `MaxOverlap`)

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 one of its equivalents) then returns the matching results as tuples in the parameters `Row`, `Column`, `Angle`, `Scale` (or its variants for anisotropic scaling), 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 3.13b, 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 neither for the models themselves nor for the model's ROI 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 3.13b and figure 3.13d).

Figure 3.13e 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.

3.2.4.7 Search for Multiple Models Simultaneously (`ModelIDs`)

If you are searching for instances of multiple models in a single image, you can of course call the operator `find_shape_model`, `find_scaled_shape_model`, or `find_aniso_shape_model` multiple times. But a much faster alternative is to use the operators `find_shape_models`, `find_scaled_shape_models`, or `find_aniso_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 3.2.4.6), the matching result parameters `Row` etc. return tuples of values.

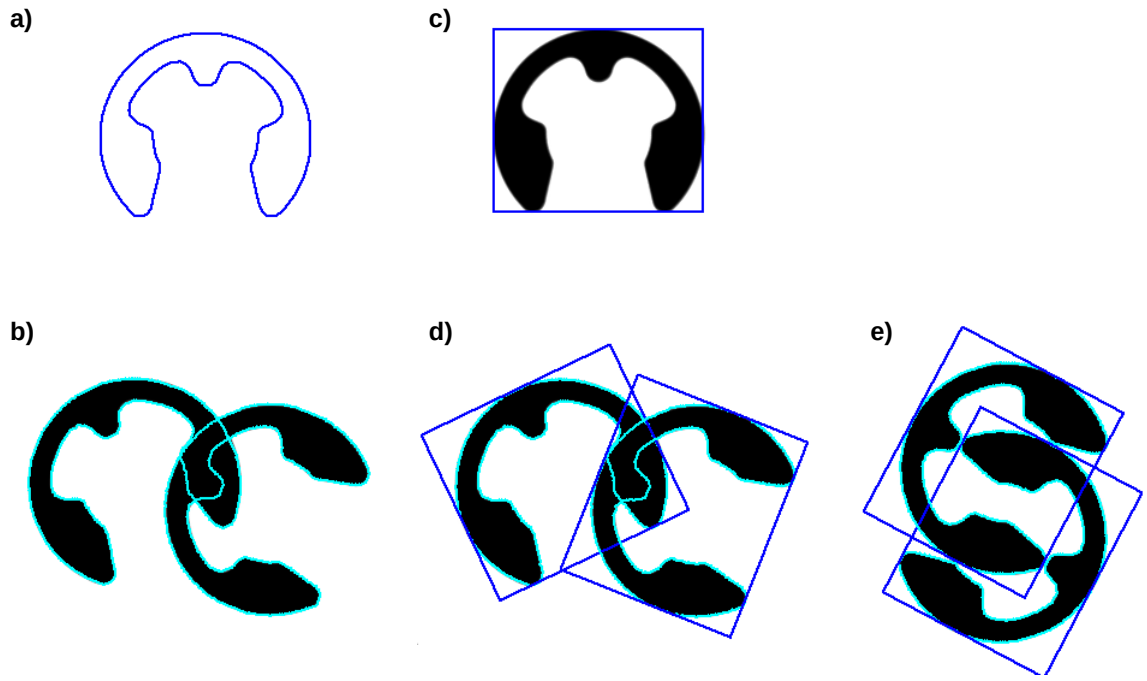


Figure 3.13: A closer look at overlapping matches: a) model of the security ring; b) model overlap; c) smallest rectangle surrounding the model; d) rectangle overlap; e) pathological case.

- 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 array (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 parameter `NumMatches`. By passing a tuple of values, you can specify for each model individually how many instances are to be found. The tuple `[3,0]`, e.g., specifies to return the best three 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 `%HALCONEXAMPLES%\solution_guide\matching\multiple_models.hdev` uses the operator `find_scaled_shape_models` to search simultaneously for the rings and nuts depicted in figure 3.14.

Step 1: Create the models

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`.

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

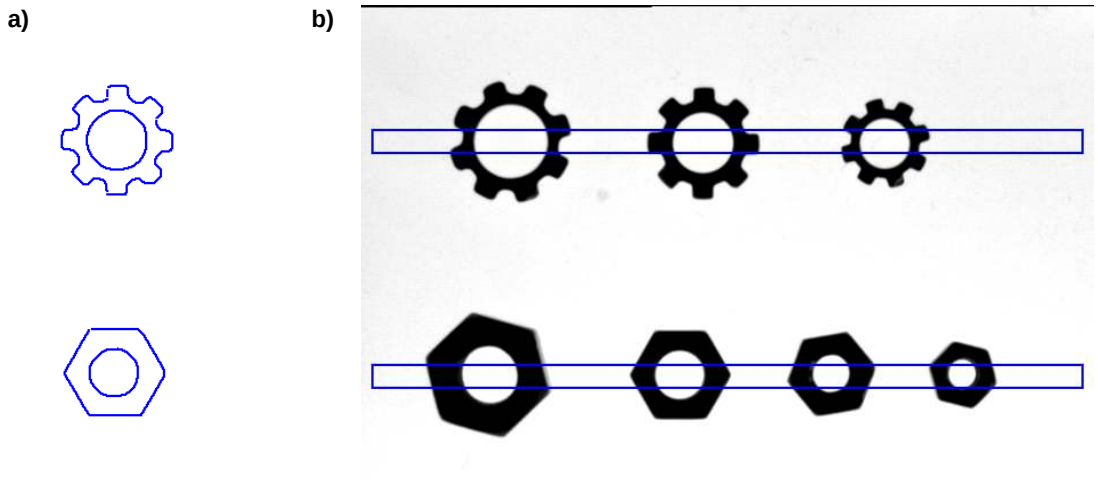


Figure 3.14: Searching for multiple models : a) models of ring and nut; b) search ROIs for the two models.

Step 2: Specify individual search ROIs

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 3.2.4.1 on page 63, a search ROI corresponds to the extent of the allowed movement. Thus, narrow horizontal ROIs can be used in the example (see figure 3.14b).

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`, i.e., in each region the corresponding gray values of the search image are "painted". 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 and the domain of the second image is restricted to the second ROI.

```
gen_rectangle1 (SearchROIRing, 110, 10, 130, Width - 10)
gen_rectangle1 (SearchROINut, 315, 10, 335, Width - 10)
concat_obj (SearchROIRing, SearchROINut, SearchROIs)
add_channels (SearchROIs, SearchImage, SearchImageReduced)
```

Step 3: Find all instances of the two models

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 valid for both models. Section 3.2.5.2 on page 72 shows how to access the matching results.

```
find_scaled_shape_models (SearchImageReduced, ModelIDs, [-rad(22.5), \
                          -rad(30)], [rad(45),rad(60)], [0.8,0.6], [1.2, \
                          1.4], 0.7, 0, 0, 'least_squares', 0, 0.8, \
                           RowCheck, ColumnCheck, AngleCheck, ScaleCheck, \
                           Score, ModelIndex)
```

3.2.4.8 Specify the Needed Accuracy (`SubPixel`)

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 resulting parameters `Row`, `Column`, `Angle`, and `Scale` (or the corresponding parameters for anisotropic scaling) 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 (or the corresponding parameters for anisotropic scaling), respectively, when creating the model (see section 3.2.3.3 on page 58 and section 3.2.3.4 on page 59).

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 can be achieved). 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 or the corresponding parameters for anisotropic scaling (see section 3.2.3.3 on page 58 and section 3.2.3.4 on page 59): 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 $\approx \frac{1}{10}°$.

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

When you set the parameter SubPixel to '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.

Sometimes objects are not found or found only with a low accuracy because they are slighly deformed compared to the model. If your object is most probably deformed, you can allow a maximum deformation of a few pixels for the model by setting SubPixel additionally to 'max_deformation ', which is followed by the number of pixels allowed for the deformation. For example, if you set SubPixel to 'max_deformation 2', the contour of the searched object may differ by up to two pixels from the shape of the model. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment. Note that high values for the maximal allowed deformation increase the runtime and the risk of finding wrong model instances, especially for small models. Thus, the value should be chosen as small as possible but as high as necessary.

Please note that the **accuracy of the estimated position may decrease if you modify the point of reference** using set_shape_model_origin! This effect is visualized in figure 3.15: 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 point of reference 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 3.15).

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 point of reference.

For maximum accuracy in case the point of reference 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 point of reference.

3.2.4.9 Restrict the Number of Pyramid Levels (NumLevels)

The parameter NumLevels, which you already specified when creating the model, allows you to use a different (in most cases a more restrictive) value in the search process. By using the value 0 for NumLevels, the value specified when creating the model is used.

Optionally, NumLevels can contain a second value, so that you can specify not only the highest but also the lowest pyramid level used for the search. If the search is finished on a pyramid level that is higher than the first pyramid level, which corresponds to the original, full-sized image, the search becomes faster. On the other hand, the search is then also less robust and less accurate. If objects should be found also in images of poor quality, e.g., if the object is defocused, deformed, or noisy, you can activate the increased tolerance mode by specifying the second value negatively. Then, the matches on the lowest pyramid level that still provides matches are returned.

In case high noise is affecting the matching results you can also use adapt_shape_model_high_noise in order to estimate the lowest pyramid level for the shape model.

3.2.4.10 Set the Parameters 'min_contrast' and 'timeout' via set_shape_model_param

Most of the parameters are set during the creation of the shape model. Two parameters can be set also at a later time using the operator set_shape_model_param. In particular, you can set the parameters 'min_contrast'

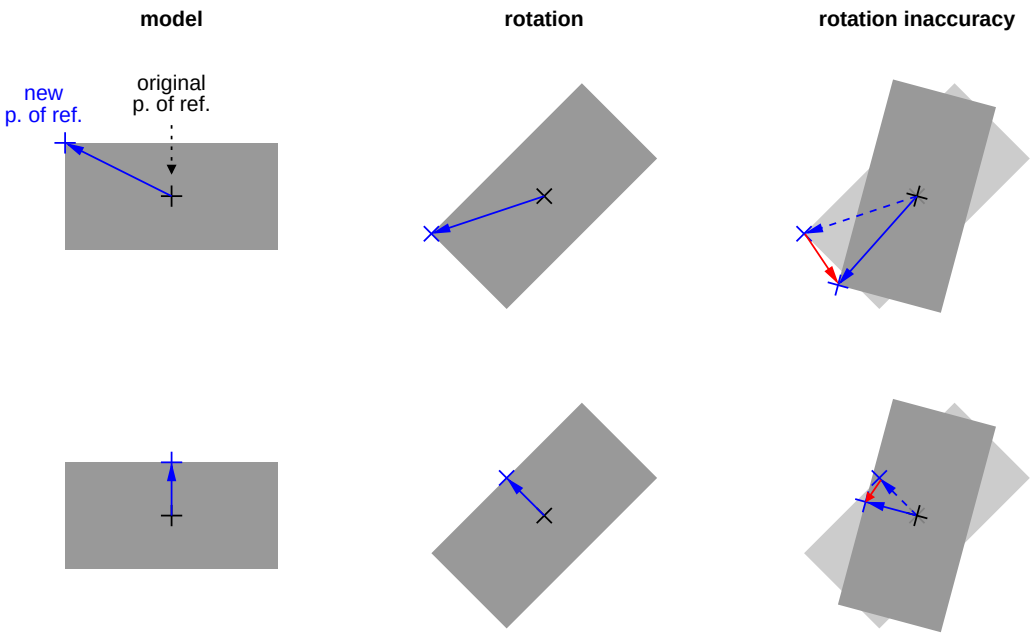


Figure 3.15: Two examples for the effect of the inaccuracy of the estimated orientation on a moved point of reference.

and 'timeout'. 'min_contrast' can be changed, e.g., if the contrast for some search images is too low. 'timeout' can be set only with set_shape_model_param but not during the creation of the model. With it, you can specify a maximum period of time after which the search is guaranteed to terminate, i.e., you can make the search interruptible. But note that for an interrupted search no result is returned. Additionally, when setting the timeout mechanism, the runtime of the search may be increased by up to 10%. The example %HALCONEXAMPLES%\hdevelop\Matching\Shape-Based\set_shape_model_timeout.hdev shows how to use the timeout mechanism.

3.2.4.11 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?

Set set_system('border_shape_models','true') (see section 3.2.4.3 on page 64).

? Is the search algorithm "too greedy"?

As described in section 3.2.4.5 on page 65, 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 3.2.4.3 on page 64, 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 or one of its equivalents. Alternatively, reduce MinScore.

? Does the object have a low contrast?

If the object should be recognized in this state nevertheless, reduce the parameter `MinContrast` (see section 3.2.3.5 on page 60).

? 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 3.2.3.5 on page 60). If only a small part of the object is affected, it may be better to reduce `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 3.2.4.6 on page 66).

? Are multiple matches found on the same object?

If the object is almost symmetric, restrict the allowed range of rotation as described in section 3.2.3.3 on page 58 or decrease the parameter `MaxOverlap` (see section 3.2.4.6 on page 66).

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 `MinScore` as far as possible, i.e., as long as the matching succeeds.
- Now, increase `Greediness` until the matching fails. Try reducing `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 3.2.3.3 on page 58 and section 3.2.3.4 on page 59. Alternatively, adjust the corresponding parameters when calling `find_shape_model`, `find_scaled_shape_model`, or `find_aniso_shape_model`.
- Restrict the search to a region of interest as described in section 3.2.4.1 on page 63.
- Make sure that the model consists of many contour points and the contours have dominant structures that are discriminable clearly from other structures in the image. That is, already when taking the images and preparing the model, you can significantly influence the speed of the search by preferring larger models with dominant structures to smaller models with weak structures (see also section 2.1.2.2 on page 19 and section 2.3.2 on page 25).

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

- Increase `MinContrast` as long as the matching succeeds.
- If you are searching for a particularly large object, it typically helps to select a higher point reduction with the parameter `Optimization` (see section 3.2.3.2 on page 58).
- Increase `AngleStep` (and `ScaleStep` or the corresponding parameters for anisotropic scaling) as long as the matching succeeds.
- If the speed of the matching is more important than its robustness and accuracy, terminate the search on a higher pyramid level as described in section 3.2.4.9 on page 69.

Note that for many of the above described optimizations you can also use the HDevelop Matching Assistant that guides you through the different steps of the complete matching process. How to use the Matching Assistant is described in the HDevelop User’s Guide, section 7.3 on page 217.

3.2.5 Use the Specific Results of Shape-Based Matching

The results of shape-based matching can be used for all tasks introduced in section 2.4 on page 28 for general matching. Furthermore, shape-based matching can return multiple instances of a model and additionally can deal with multiple models simultaneously. How to use these specific results is described in section 3.2.5.1 and section 3.2.5.2.

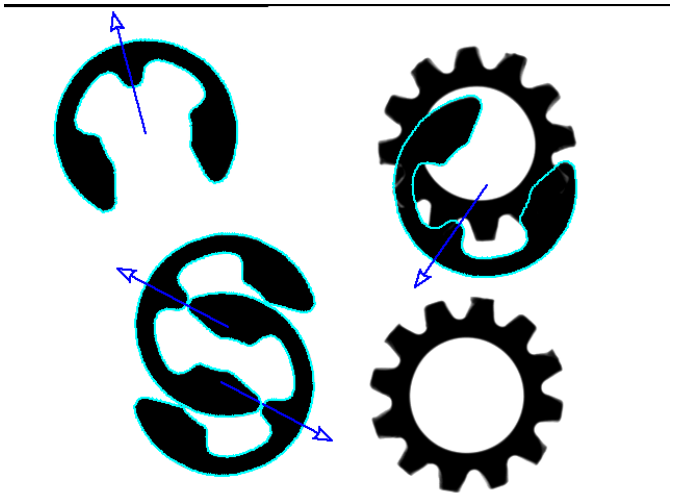


Figure 3.16: Displaying multiple matches; the used model is depicted in figure 3.10a on page 64.

3.2.5.1 Deal 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 `%HALCONEXAMPLES%\solution_guide\matching\multiple_objects.hdev` shows how to access these results in a loop.

Step 1: Determine the affine transformation

The transformation corresponding to the movement of the match is determined as described in section 2.4.3.1 on page 33. In contrast to a search for single matches, here, the position of the match is extracted from the tuple via the loop variable.

```
find_shape_model (SearchImage, ModelID, 0, rad(360), 0.6, 0, 0.55, \
                  'least_squares', 0, 0.8, RowCheck, ColumnCheck, \
                  AngleCheck, Score)
for j := 0 to |Score| - 1 by 1
    vector_angle_to_rigid (0, 0, 0, RowCheck[j], ColumnCheck[j], \
                           AngleCheck[j], MovementOfObject)
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition, \
                              MovementOfObject)
```

Step 2: Use the transformation

In this example, the transformation is also used to display an arrow that visualizes the orientation (see figure 3.16). For this, the position of the arrow head is transformed using `affine_trans_pixel` with the same transformation matrix that was used for the XLD model. As remarked in section 2.4.3.2 on page 34, **you must use** `affine_trans_pixel` and not `affine_trans_point_2d`.

```
    affine_trans_pixel (MovementOfObject, -120, 0, RowArrowHead, \
                        ColumnArrowHead)
    disp_arrow (WindowHandle, RowCheck[j], ColumnCheck[j], RowArrowHead, \
                ColumnArrowHead, 2)
```

3.2.5.2 Deal with Multiple Models

When searching for multiple models simultaneously as described in section 3.2.4.7 on page 66, it is useful to store the information about the models, i.e., the XLD models, in tuples. The following example code stems from the already partly described HDevelop program `%HALCONEXAMPLES%\solution_guide\matching\multiple_models.hdev`, which uses the operator `find_scaled_shape_models` to search simultaneously for the rings and nuts depicted in figure 3.14 on page 68.

Step 1: Access the XLD models

The XLD contours corresponding to the two models are accessed with the operator get_shape_model_contours.

```
create_scaled_shape_model (ImageROIRing, 'auto', -rad(22.5), rad(45), \
                           'auto', 0.8, 1.2, 'auto', 'none', \
                           'use_polarity', 60, 10, ModelIDRing)
inspect_shape_model (ImageROIRing, PyramidImage, ModelRegionRing, 1, 30)
get_shape_model_contours (ShapeModelRing, ModelIDRing, 1)
create_scaled_shape_model (ImageROINut, 'auto', -rad(30), rad(60), 'auto', \
                           0.6, 1.4, 'auto', 'none', 'use_polarity', 60, \
                           10, ModelIDNut)
inspect_shape_model (ImageROINut, PyramidImage, ModelRegionNut, 1, 30)
get_shape_model_contours (ShapeModelNut, ModelIDNut, 1)
```

Step 2: Save the information about the models in tuples

To facilitate the access to the shape models later, the XLD contours are saved in tuples in analogy to the model IDs (see section 3.2.4.7 on page 66). 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.

```
count_obj (ShapeModelRing, NumContoursRing)
count_obj (ShapeModelNut, NumContoursNut)
ModelIDs := [ModelIDRing,ModelIDNut]
concat_obj (ShapeModelRing, ShapeModelNut, ShapeModels)
StartContoursInTuple := [1,NumContoursRing + 1]
NumContoursInTuple := [NumContoursRing,NumContoursNut]
```

Step 3: Access the found instances

As described in section 3.2.5.1 on page 72, 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.

```
find_scaled_shape_models (SearchImageReduced, ModelIDs, [-rad(22.5), \
                         -rad(30)], [rad(45),rad(60)], [0.8,0.6], [1.2, \
                         1.4], 0.7, 0, 0, 'least_squares', 0, 0.8, \
                          RowCheck, ColumnCheck, AngleCheck, ScaleCheck, \
                          Score, ModelIndex)
for i := 0 to |Score| - 1 by 1
    Model := ModelIndex[i]
    get_hom_mat2d_from_matching_result (RowCheck[i], ColumnCheck[i], \
                                        AngleCheck[i], ScaleCheck[i], \
                                        ScaleCheck[i], \
                                        MoveAndScalingOfObject)
    copy_obj (ShapeModels, ShapeModel, StartContoursInTuple[Model], \
              NumContoursInTuple[Model])
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition, \
                              MoveAndScalingOfObject)
    dev_display (ModelAtNewPosition)
endfor
```

3.2.6 Adapt 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 perpendicularly 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 you can on the one hand 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 `calibrate_cameras` (see Solution Guide III-C, section 3.2 on page 59). 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` (see Solution Guide III-C, section 3.4 on page 78).

On the other hand, you can also use the uncalibrated perspective deformable matching. It works similar to the shape-based matching but already considers perspective deformations of the model and returns a projective transformation matrix (2D homography) instead of a 2D pose consisting of a position and an orientation. Additionally, a calibrated perspective deformable matching is provided for which a camera calibration has to be applied and which results in the 3D pose of the object. For further information on perspective deformable matching please refer to section 3.5 on page 99 or to the Solution Guide I, chapter 10 on page 99 for the uncalibrated case and Solution Guide III-C, section 4.6 on page 112 for the calibrated case.

Note that if the same perspective view is used for all images, rectifying the images before applying the matching is faster and more accurate than using the perspective deformable matching. But if different perspective views are needed, you must use the perspective matching.

3.3 Component-Based Matching

The component-based matching is an extension of the shape-based matching. Like shape-based matching, the component-based matching extracts contours and matches their shapes against the shapes of previously created models. But in contrast to shape-based matching, a component model consists of several components that can change their relations, i.e., they can move and rotate relative to each other. The possible relations have to be determined or specified when creating the model. Then, the actual matching returns the individual relations for the found model instances. Note that in contrast to shape-based matching, for component-based matching no scaling in size is possible.

The task of locating components that can move relative to each other is a little bit more complex than the process needed for a shape-based matching. For example, instead of a single ROI several ROIs (containing the initial components) have to be selected or extracted. Additionally, the relations, i.e., the possible movements between the model components have to be determined. For an overview, figure 3.17 illustrates the main steps needed for a component-based matching.

For detailed information, the following sections show

- a first example for a component-based matching (section 3.3.1),
- how to extract the initial components of a component model (section 3.3.2 on page 76),
- how to create a suitable component model (section 3.3.3 on page 78),
- how to apply the search (section 3.3.4 on page 85), and
- how to deal with the results that are specific for component-based matching (section 3.3.5 on page 88).

3.3.1 A First Example

In this section we give a quick overview of the matching process with component-based matching. To follow the example actively, start the HDevelop program `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-2D\cbm_bin_switch.hdev`, which locates instances of a switch that consists of two components and determines if the setting of each found switch is 'on' or 'off'.

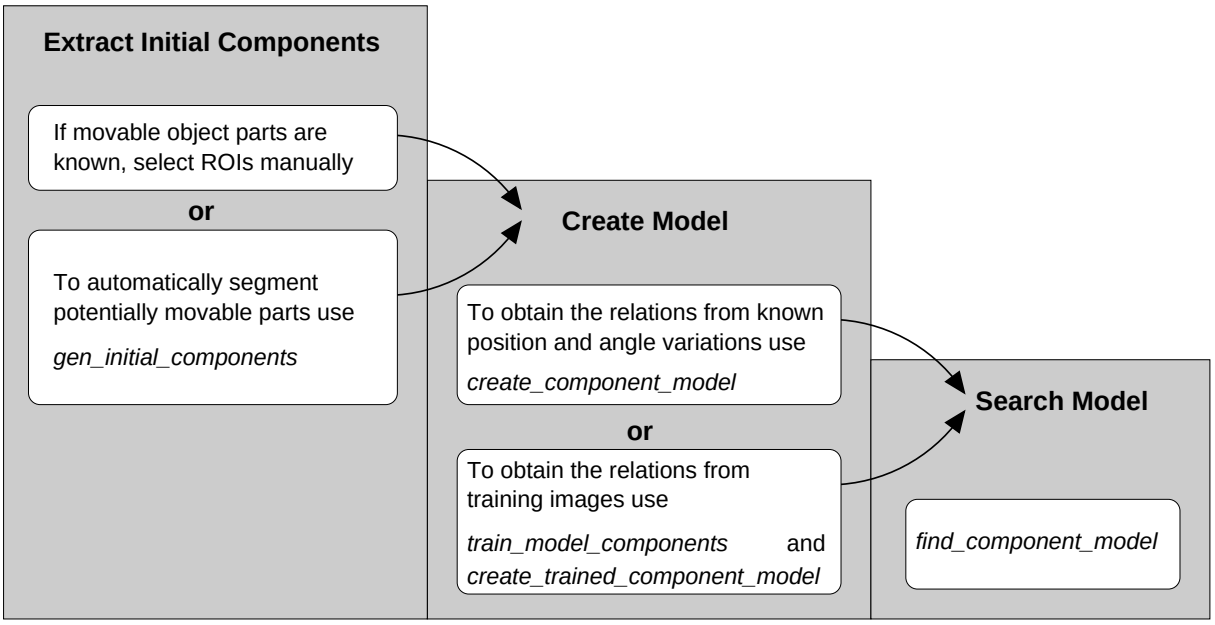


Figure 3.17: Main steps of component-based matching.

Step 1: Extract the initial components

First, the reference image is read. It shows a switch with the setting 'on' (see figure 3.18, left). As it consists of two parts that can move relative to each other, two ROIs are created. Concatenated into a tuple (`InitialComponents`), the regions build the initial components.

```
read_image (ModelImage, 'bin_switch/bin_switch_model')
gen_rectangle1 (Region1, 78, 196, 190, 359)
gen_rectangle1 (Sub1, 150, 196, 190, 321)
difference (Region1, Sub1, InitialComponents)
gen_rectangle1 (Region2, 197, 204, 305, 339)
gen_rectangle1 (Sub2, 205, 232, 285, 314)
difference (Region2, Sub2, InitialComponent)
concat_obj (InitialComponents, InitialComponent, InitialComponents)
dev_display (ModelImage)
dev_display (InitialComponents)
```



Figure 3.18: (left) reference image with the two initial components (switch is 'on'); (right) training image (switch is 'off').

Step 2: Train the possible relations between the components

Then, the relations between the components have to be determined. Here, they are obtained by a training. Another method that is very common in practice is to manually define the possible relations (see section 3.3.3 on page 78). For the training, a training image is read (see figure 3.18, right). As only two different relations of the initial components are valid (one for the setting 'on' and one for the setting 'off'), a single training image showing the setting 'off' together with the reference image showing the setting 'on' are sufficient for the training. If more than two relations were valid, more training images would be needed. Using the reference image, the initial components, the training image, and several parameters that control the training, the training is applied with `train_model_components`.

```
read_image (TrainingImage, 'bin_switch/bin_switch_training_1')
train_model_components (ModelImage, InitialComponents, TrainingImage, \
                        ModelComponents, 30, 30, 20, 0.7, -1, -1, rad(25), \
                        'speed', 'rigidity', 0.2, 0.5, ComponentTrainingID)
```

Step 3: Create the component model

If the result of the training is satisfying, the component model can be created. To make the model less strict, before the creation small tolerances are added to the obtained relations using `modify_component_model`.

```
modify_component_relations (ComponentTrainingID, 'all', 'all', 1, rad(1))
create_trained_component_model (ComponentTrainingID, 0, rad(360), 10, 0.7, \
                                'auto', 'auto', 'none', 'use_polarity', \
                                'false', ComponentModelID, RootRanking)
```

Step 4: Find the component model and derive the corresponding relations

Now, the search images are read and instances of the component model are searched with `find_component_model`. For each match, the individual components and their relations are queried using `get_found_component_model`. Dependent on the angle between the components, the procedure `visualize_bin_switch_match` visualizes the setting of the found switch.

```
    read_image (SearchImage, 'bin_switch/bin_switch_' + ImgNo)
    find_component_model (SearchImage, ComponentModelID, 1, 0, rad(360), 0, \
                          0, 1, 'stop_search', 'prune_branch', 'none', 0.6, \
                          'least_squares', 0, 0.85, ModelStart, ModelEnd, \
                          Score, RowComp, ColumnComp, AngleComp, ScoreComp, \
                          ModelComp)
    dev_display (SearchImage)
    for Match := 0 to |ModelStart| - 1 by 1
        get_found_component_model (FoundComponents, ComponentModelID, \
                                   ModelStart, ModelEnd, RowComp, \
                                   ColumnComp, AngleComp, ScoreComp, \
                                   ModelComp, Match, 'false', RowCompInst, \
                                   ColumnCompInst, AngleCompInst, \
                                   ScoreCompInst)
        dev_display (FoundComponents)
        visualize_bin_switch_match (AngleCompInst, Match, WindowHandle)
    endfor
```

The following sections go deeper into the details of the individual steps of a component-based matching and the parameters that have to be adjusted.

3.3.2 Extract the Initial Components

In contrast to shape-based matching, a model is not generated from a single ROI but from several concatenated regions that contain the initial components of the model, i.e., the parts of the model that can move and rotate relative to each other. The initial components can be extracted by different means:

- If the components are approximately known, the corresponding ROIs can be selected manually and are concatenated into a tuple (section 3.3.2.1).

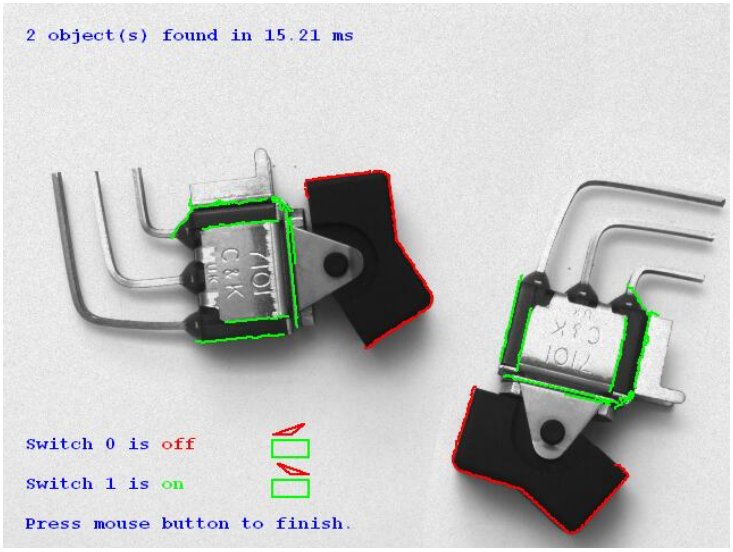


Figure 3.19: Instances of the component model in a search image and their determined switch settings.

- If the initial components are not known, potential candidates can be derived using the operator gen_initial_components (section 3.3.2.2).

3.3.2.1 Manual Selection of Initial Components

If the initial components are approximately known, for each component, i.e., for each movable part of the object of interest, an ROI is selected manually (see section 2.1.1 on page 17 for the selection of ROIs). Then, the ROIs of all initial components are concatenated into a tuple. An example for the manual selection of the initial components was already shown in the example in section 3.3.1 on page 74. There, the initial components were built by the ROIs that were selected for the two parts of a switch that are allowed to change their relation.

3.3.2.2 Automatic Extraction of Initial Components

If the initial components are not known, i.e., if it is not yet clear which parts of an object may move relative to each other, the potential ROIs have to be derived automatically. This is shown, e.g., in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Matching\Component-Based\cbm_label_simple.hdev. There, a 'best before' label has to be located and the relations of its components have to be determined. The individual components are not yet known, so in a first step only an ROI containing the complete label is selected (see figure 3.20, left) and the domain of the image is reduced to this region to obtain a template image. From this template image, the initial components are derived using gen_initial_components (see figure 3.20, right).

```
read_image (Image, 'label/label_model')
gen_rectangle1 (ModelRegion, 119, 106, 330, 537)
reduce_domain (Image, ModelRegion, ModelImage)
gen_initial_components (ModelImage, InitialComponents, 40, 40, 20, \
                        'connection', [], [])
dev_display (Image)
dev_display (InitialComponents)
```

Besides the template image the operator expects values for several parameters that control the segmentation of the components, in particular parameters describing the minimum and maximum contrast needed to extract contours from the image and the minimum size a connected contour must have to be returned. These parameters are similar to the parameters that are used for shape-based matching (see section 3.2.3 on page 55). Additionally, some generic parameters that control the internal image processing can be adjusted. The influence of different values for the control parameters is exemplarily shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Matching\Component-Based\cbm_param_visual.hdev. In figure 3.21, e.g., a result of gen_initial_components with a too low contrast is shown. Note that for component-based matching it is even

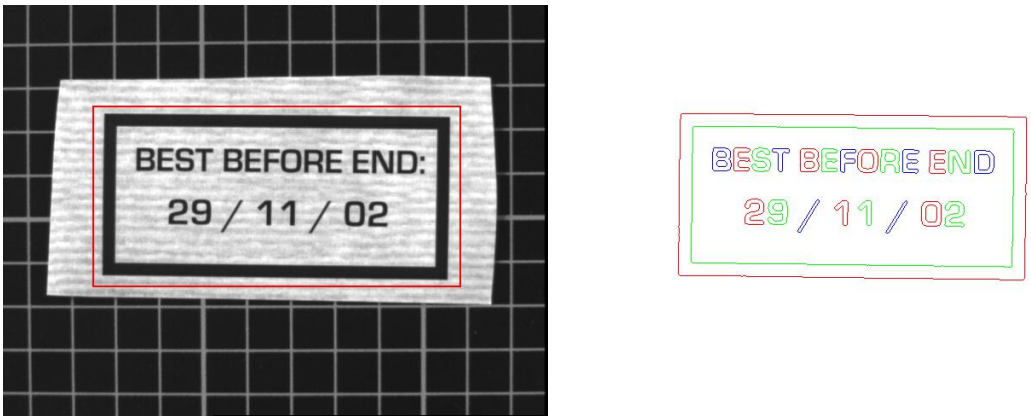


Figure 3.20: (left) ROI that is used to create a template image; (right) automatically derived initial components.

more important than for the other matching approaches that only those structures are contained in the model that belong to the object. Otherwise, the contours derived by the noise become initial components as well.

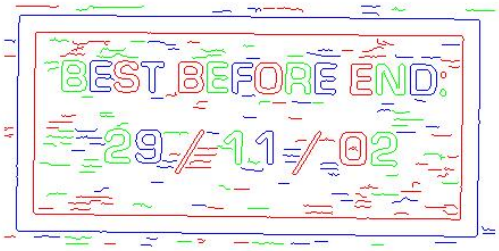


Figure 3.21: Segmentation of initial components with a too low contrast.

3.3.3 Create a Suitable Component Model

When the initial components are selected or extracted from the reference image, in a second step, the component model can be created. For the creation of the model the relations between the components have to be known. Thus, the creation can be realized by different means:

- If the possible relations between the components are already known, the component model can be directly created using `create_component_model` (section 3.3.3.1). The relations between the model components are discussed in more detail in section 3.3.3.2 on page 80.
- If the possible relations are not known, they have to be derived from a set of training images that show all possible variations of relations (section 3.3.3.3 on page 81).

If the model is created using a training, between the training and the final creation of the model it may be suitable to

- visualize different training results (section 3.3.3.4 on page 82) or
- modify the training results (section 3.3.3.5 on page 84).

After creating the model, you may want to

- store the created model to file so that you can reuse it in another application (section 3.3.3.6 on page 84), or
- query information from the already existing model (section 3.3.3.7 on page 85).

3.3.3.1 Create Model from Known Relations

If the relations, i.e., the possible movements between the model components are known, you can create the component model directly using `create_component_model`, which creates a component model for a matching based on explicitly specified components and relations. An example is `%HALCONEXAMPLES%\hdevelop\Matching\Component-Based\cbm_modules_simple.hdev`. There, the relations between the individual modules of a compound object that are shown in figure 3.22 are known by the user.

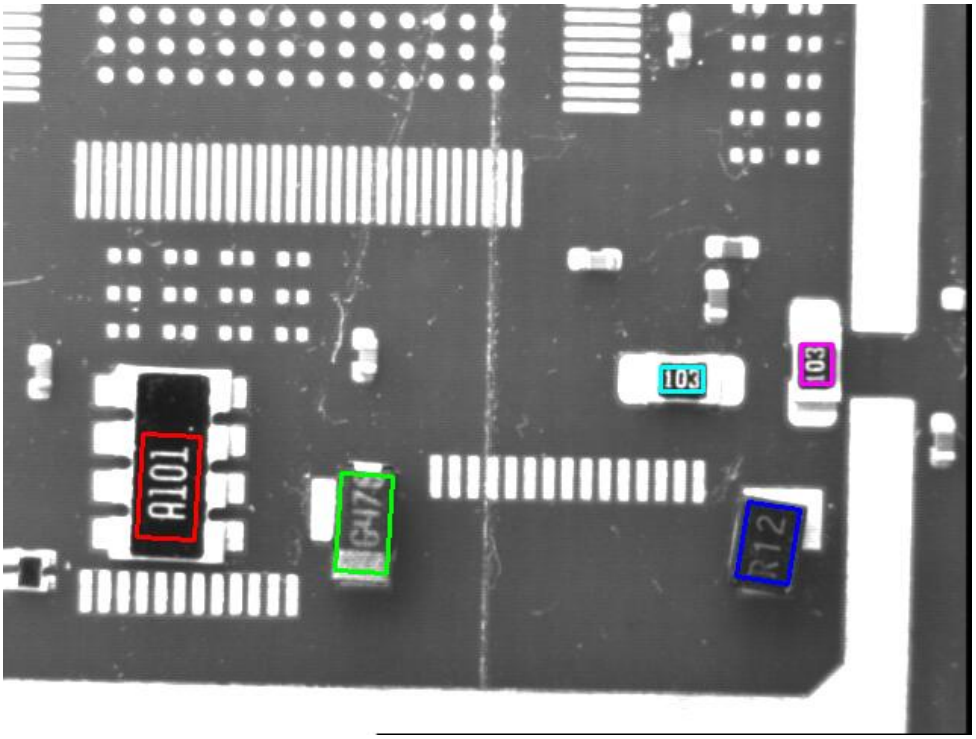


Figure 3.22: Initial components of a compound object with known relations.

After the ROIs of the components were selected and concatenated the operator `create_component_model` is applied.

```
read_image (ModelImage, 'modules/modules_model')
gen_rectangle2 (ComponentRegions, 318, 109, -1.62, 34, 19)
gen_rectangle2 (Rectangle2, 342, 238, -1.63, 32, 17)
gen_rectangle2 (Rectangle3, 355, 505, 1.41, 25, 17)
gen_rectangle2 (Rectangle4, 247, 448, 0, 14, 8)
gen_rectangle2 (Rectangle5, 237, 537, -1.57, 13, 10)
concat_obj (ComponentRegions, Rectangle2, ComponentRegions)
concat_obj (ComponentRegions, Rectangle3, ComponentRegions)
concat_obj (ComponentRegions, Rectangle4, ComponentRegions)
concat_obj (ComponentRegions, Rectangle5, ComponentRegions)
create_component_model (ModelImage, ComponentRegions, 20, 20, rad(25), 0, \
                        rad(360), 15, 40, 15, 10, 0.8, [4,3,3,3,3], 0, \
                        'none', 'use_polarity', 'true', ComponentModelID, \
                        RootRanking)
```

Note that instead of the explicit relations the variations of the positions and the angles of the model components are passed to the operator. These describe the movements of the components independently from each other but relative to their positions in the reference image. That is, if a search image would be transformed so that the complete compound object is positioned and oriented approximately as in the reference image, the individual components may differ from their corresponding model components for `VariationRow`/2 pixels in row direction, `VariationColumn`/2 pixels in column direction, and `VariationAngle`/2 in their orientation. If single values are passed to the variation parameters, they are valid for all components. If tuples of values are passed, they must contain a value for each component. In figure 3.23 the variations for the components of figure 3.22 are illustrated. In particular,

the gray arrows show the range of movement allowed for the point of reference of the individual components and the gray rectangles show the allowed rotation. The operator `create_component_model` automatically derives the relations between the components from the variations and uses them to create the component model.

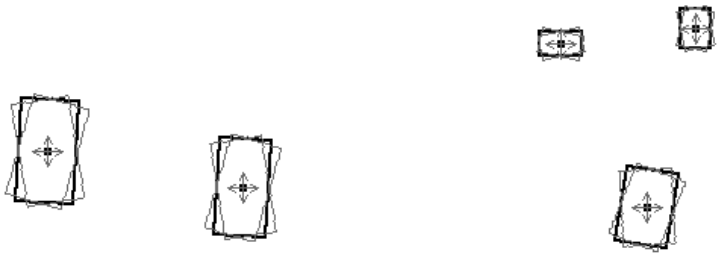


Figure 3.23: Variations (gray) of the model components (black) that are used to automatically derive the relations between the model components (`VariationRow`=20, `VariationAngle`=rad(25)).

3.3.3.2 Search Tree and Relations Between Model Components

Whereas the manually selected variations show the allowed movements and rotations of the individual components relative to their position in the template image, the relations show the allowed movements and rotations of the components relative to a reference component.

The relations can be queried together with the search tree using `get_component_model_tree`. The search tree specifies in which sequence the components are searched and how they are searched relative to each other. It depends on the selected root component, i.e., the component that is searched first (how to select the root component is described in section 3.3.4.2 on page 86). The reference component used to calculate the relations of an individual component is the component's predecessor in the search tree. For example, in the search tree that is shown in figure 3.24 for the compound model of the example program, the leftmost component is the root component and the components are searched in the direction indicated by the dim gray lines.

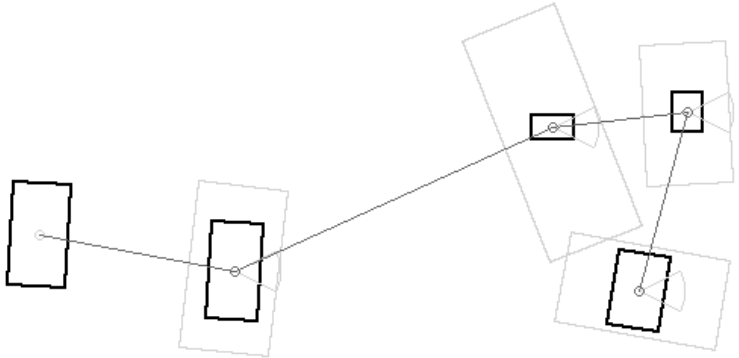


Figure 3.24: Relations (light gray) and search tree (dim gray) for the model components (black) relative to the automatically derived root component.

If a component model is obtained by a training with multiple training images (see section 3.3.3.3), you can also use `get_component_relations` to query the relations between the components that are determined by the training. Then, no search tree is used as basis for the calculations. Instead, the reference component used to calculate the relations of the components is always the selected root component. Thus, the relations returned by both operators differ significantly. Note that `get_component_relations` is mainly used to evaluate the success of a training before creating the corresponding component model, whereas `get_component_model_tree` is applied after the

creation of the model to visualize the search space used for the actual matching (see also section 3.3.3.4 on page 82 and figure 3.28 on page 84).

In both cases, the relations are returned as regions and as numerical values. In particular, the following information is obtained:

- The positions of the points of reference for all components are represented as small circles. The corresponding numerical values are the coordinates of the positions that are returned in the parameters `Row` and `Column`.
- The orientation relations for all components except the root component, i.e., the variations of the orientations relative to the respective reference components are represented by circle sectors with a fixed radius that originate at the mean relative positions of the components. The corresponding numerical values are the angles that are returned in the parameter `Phi`.
- The position relations for all components except the root component, i.e., the movements of the reference points of the specific components relative to the reference point of their reference component are represented as rectangles. The corresponding numerical values are the parameters of the rectangle `Length1`, `Length2`, `AngleStart`, and `AngleExtent`.

If the model is created by manually selected variations, the derived relations show certain regularities. In particular, each rectangle is approximately orthogonal to the view from the predecessor to the searched component and the width of the rectangle orthogonally to this view depends on the angle variation and the distance between the two components. That is, with an increasing distance between the components the width of the rectangle increases, too (see figure 3.24 on page 80). For models that are obtained by a training as described in the following section, the behavior is different, because the relations are then obtained by arbitrary relations of the components in the training images instead of regular variations. Thus, also arbitrary orientations and sizes of the rectangles are possible (see figure 3.28 on page 84).

3.3.3.3 Create Model from Training Images

If the relations are not explicitly known, you have to determine them using training images that show the needed variety of relations.

For example, in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Matching\Component-Based\cbm_label_simple.hdev` several training images are read and concatenated into a tuple (`TrainingImages`). These training images show the already introduced 'best before' label with different relations between its individual parts (see also section 3.3.2.2 on page 77).

```
gen_empty_obj (TrainingImages)
for Index := 1 to 5 by 1
    read_image (TrainingImage, 'label/label_training_' + Index)
    concat_obj (TrainingImages, TrainingImage, TrainingImages)
endfor
```

For the training of the components and relations the operator `train_model_components` is applied. The training is controlled by a set of different parameters, which are described in detail in the Reference Manual. During the training, from the initial components the model components are derived by a clustering. For example, if you compare the initial components in figure 3.20 on page 78 (right) with the model components in figure 3.25 (right), you see that those initial components that have the same relations in all training images are merged.

```
train_model_components (ModelImage, InitialComponents, TrainingImages, \
                        ModelComponents, 40, 40, 20, 0.85, -1, -1, rad(15), \
                        'reliability', 'rigidity', 0.2, 0.5, \
                        ComponentTrainingID)
dev_display (Image)
dev_display (ModelComponents)
```

After the training, the component model that is based on the trained components is created with `create_trained_component_model`. Before the creation, `modify_component_relations` is applied to add small tolerance values to the relations and thus make the model less strict.

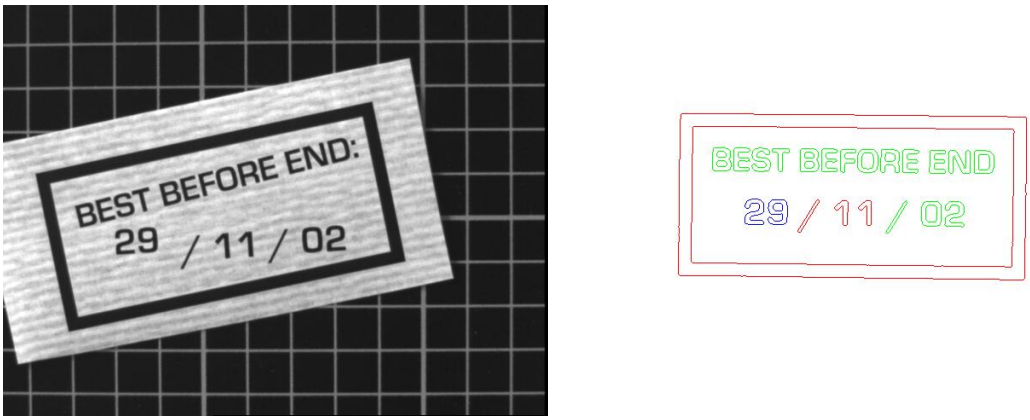


Figure 3.25: (left) one of several training images; (right) the model components for the component model.

```
modify_component_relations (ComponentTrainingID, 'all', 'all', 15, rad(5))
create_trained_component_model (ComponentTrainingID, -rad(30), rad(60), 10, \
                                0.8, 'auto', 'auto', 'none', \
                                'use_polarity', 'false', ComponentModelID, \
                                RootRanking)
```

Note that `gen_initial_components`, which was introduced in section 3.3.2.2 on page 77 for the automatic derivation of the initial components for a model, can be used also to test different values for the control parameters `ContrastLow`, `ContrastHigh`, and `MinSize` that have to be adjusted for the training of the model components. Thus, for the training of a component model the operator can be used similar as `inspect_shape_model` is used for shape-based matching (see section 3.2.3 on page 55).

3.3.3.4 Visualize the Training Results

The training with `train_model_components` returns the final model components and trains the relations between them. Different results of the training can be visualized. In particular, you can visualize

- the initial components or the final model components in a specific image,
- the relations between the components, and
- the search tree.

To visualize all final model components, all initial components, or a specific initial component for a specific image, you apply the operator `get_training_components`. There, you specify amongst others if the model components, the initial components, or a specific initial component should be queried and in which image the components should be searched. Besides the numerical results (`Row`, `Column`, `Angle`, and `Score`), the contours of the initial components or model components are returned and can be visualized.

```
get_training_components (ModelComponents, ComponentTrainingID, \
                         'model_components', 'model_image', 'false', \
                         Row, Column, Angle, Score)
dev_display (Image)
dev_display (ModelComponents)
```

The HDevelop example program `%HALCONEXAMPLES%\hdevelop\Matching\Component-Based\cbm_param_visual.hdev` shows the use of different visualization and analysis tools provided for the component-based matching. For example, in figure 3.26 a specific initial component (the first 'B' of the 'best before' label) is obtained from the template image, whereas in figure 3.27 it is obtained from a training image. The result differs, because within the training image the ambiguities are not yet solved.



Figure 3.26: (left) Template image; (right) component '2' in the template image.

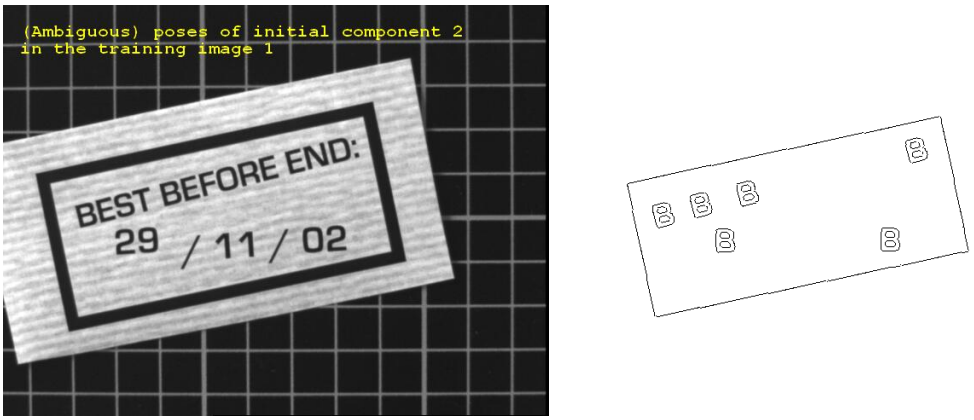


Figure 3.27: (left) Training image; (right) component '2' in the training image.

To check the quality of a training, you can use `get_component_relations` to query the relations of the components that are returned by the training. How to interpret the returned regions and numerical values is shown in section 3.3.3.2 on page 80.

```
count_obj (ModelComponents, NumComp)
for I := 0 to NumComp - 1 by 1
    get_component_relations (Relations, ComponentTrainingID, I, \
                             'model_image', Row, Column, Phi, Length1, \
                             Length2, AngleStart, AngleExtent)
    dev_display (Relations)
endfor
```

If the result of the training is not satisfying you can repeat the training with different conditions (different parameters, different training images etc.). If the training is satisfying, you can create the component model. After creating the component model, you can visualize the search tree and the corresponding relations using `get_component_model_tree`. That is, you can visualize the search space that is actually used for the later search (assuming that you select the same root component for the visualization of the search tree and for the search).

```
create_trained_component_model (ComponentTrainingID, -rad(30), rad(60), 10, \
                                0.8, 'auto', 'auto', 'none', \
                                'use_polarity', 'false', ComponentModelID, \
                                RootRanking)
for I := 0 to |RootRanking| - 1 by 1
    get_component_model_tree (Tree, Relations, ComponentModelID, \
                              RootRanking[I], 'model_image', StartNode, \
                              EndNode, Row, Column, Phi, Length1, Length2, \
                              AngleStart, AngleExtent)
    dev_display (Tree)
    dev_display (Relations)
endfor
```

The difference between the relations obtained for the training result with `get_component_relations` and those obtained for the created model with `get_component_model_tree` is illustrated in figure 3.28. In particular,

get_component_relations calculates the relations of all components relative to the root component, whereas get_component_model_tree calculates the relations of the individual components relative to their predecessor in the search tree.

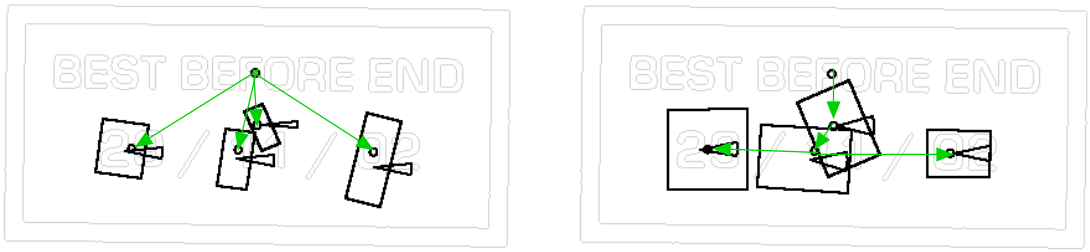


Figure 3.28: (left) Result of get_component_relations: relations of the model components relative to component 1 ("BEST BEFORE END"); (right) result of get_component_model_tree: relations within the search tree using component 1 as root component.

3.3.3.5 Modify the Training Results

Sometimes, the training returns results that are not exactly the desired ones and thus have to be modified. Besides a new training with different parameters or more suitable training images, there are two cases in which a further modification is possible after the training. In particular, you can modify

- the clustering that was used to derive the model components from the initial components and
- the relations between the model components.

During the training the rigid model components were derived from the initial components by a clustering that is controlled by the parameters AmbiguityCriterion, MaxContourOverlap, and ClusterThreshold. After a first training you can apply inspect_clustered_components to test different values for these parameters. If a suitable set of values is found that does not correspond to the values used for the first training, the new set of values can be added to the training result with cluster_model_components as is shown, e.g., in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Matching\Component-Based\cbm_param_visual.hdev.

Besides the modification of the model components, it is sometimes suitable to change the relations between the model components. This modification can be applied with modify_component_relations and is often used to add small tolerance values to the relations (see figure 3.29) to make the matching less strict. Examples were already shown in section 3.3.1 on page 74 and section 3.3.3.3 on page 81.

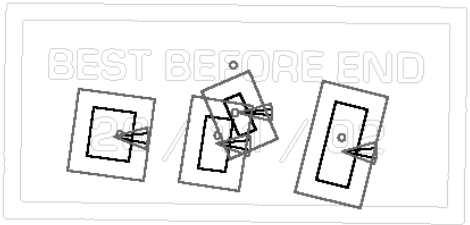


Figure 3.29: (black) Component relations obtained by the training; (gray) component relations after modification.

3.3.3.6 Reuse the Model

Often, it is useful to split the creation of the model (offline part) from the search of the model (online part). Then, a created component model can be stored to file using write_component_model. With read_component_model

it can be read from the file again for the online process. Furthermore, write_training_components can be used to store training results to file and read_training_components can be used to read them from file again.

3.3.3.7 Query Information from the Model

At different steps of a matching application it may be necessary or suitable to query information from a model.

After the training, the different training results can be queried as introduced in section 3.3.3.4 on page 82. In particular, you can query the initial components or the model components in a specific image with get_training_components and the relations between the model components that are contained in a training result with get_component_relations.

Additionally, the rigid model components obtained by the training can be inspected with inspect_clustered_components as introduced in section 3.3.3.5 on page 84.

After the creation of a model, respectively when reading an already existing component model from file, you can query the search tree and the associated relations of the component model with get_component_model_tree. Additionally, you can query the parameters of the model with get_component_model_params. The returned parameters describe the minimum score the returned instances must have (MinScoreComp), a ranking of the model components concerning their suitability to be a root component (RootRanking), and the handles of the shape models of the individual components (ShapeModelIDs). The RootRanking is useful to select a suitable root component for the later search (see section 3.3.4.2). The handles of the individual shape models can be used, e.g., to query the parameters for the shape models of each component using get_shape_model_params (see section 3.2.3 on page 55).

3.3.4 Search for Model Instances

The actual matching is performed by the operator find_component_model. In the following, we show how to select suitable parameters for this operator to adapt and optimize a matching task. In particular, we show how to

- restrict the search space to a region of interest (section 3.3.4.1),
- specify the root component via the parameter RootComponent (section 3.3.4.2),
- restrict the range of orientation for the search of the root component via the parameters AngleStartRoot and AngleExtentRoot (section 3.3.4.3),
- specify the visibility of the object via the parameter MinScore (section 3.3.4.4),
- search for multiple instances of the model by adjusting the parameters NumMatches and MaxOverlap (section 3.3.4.5),
- adjust the behavior of the search for the case that components are not found via the parameters IfRootNotFound, IfComponentNotFound, and PosePrediction (section 3.3.4.6), and
- adjust the internally applied shape-based matching, which is used to search for the individual components, via the parameters MinScoreComp, SubPixelComp, NumLevelsComp, and GreedinessComp (section 3.3.4.7 on page 87).

3.3.4.1 Restrict the Search to a Region of Interest

Similar to correlation-based matching or shape-based matching, you can restrict the search space for the root component and thus speed up the matching by applying the operator find_component_model not to the whole image but only to an ROI. The corresponding procedure is explained in more detail for a shape-based matching in section 3.2.4.1 on page 63. For component-based matching you simply have to replace find_shape_model by find_component_model. Note that although component-based matching is based on shape-based matching, you must not modify the point of reference of the model's components using set_shape_model_origin.

3.3.4.2 Specify the Root Component (RootComponent)

The components of a component model are organized in a search tree that specifies the sequence in which the individual components of a model are searched (see also section 3.3.3.1 on page 79). That is, first the root component, which is specified in the parameter RootComponent, is searched in all allowed positions and orientations. Then, the other components are searched in the sequence that is given by the search tree. In doing so, each component is searched relative to the component with the previous position in the search tree. Thus, the complete search space is needed only for the root component, whereas the other components are searched recursively.

As root component a component should be chosen that most probably can be found in the search image. An alternative criterion is the runtime of the search that is expected for each potential root component. To get the root component with the best expected runtime, you simply pass the root ranking RootRanking that was returned during the creation of the model to the parameter RootComponent.

```
find_component_model (SearchImage, ComponentModelID, RootRanking, \
                      -rad(30), rad(60), 0.5, 0, 0.5, 'stop_search', \
                      'prune_branch', 'none', 0.6, 'least_squares', 4, \
                      0.9, ModelStart, ModelEnd, Score, RowComp, \
                      ColumnComp, AngleComp, ScoreComp, ModelComp)
```

3.3.4.3 Restrict the Range of Orientation for the Root Component (AngleStartRoot, AngleExtentRoot)

The root component is searched in the allowed range of orientation that is specified by AngleStartRoot and AngleExtentRoot. We recommend to limit the allowed range of rotation as much as possible in order to speed up the search process. If necessary, the range of orientation is clipped to the range that was specified when creating the model.

3.3.4.4 Specify the Visibility of the Object (MinScore)

With the parameter MinScore you can specify how much of the component model must be visible to be returned as a match. This is similar to the corresponding parameter of shape-based matching (see section 3.2.4.3 on page 64). As the visibility of the component model depends on the visibility of the contained components (see section 3.3.4.7), please refer to section 3.3.4.6 for the case that individual components are not found.

3.3.4.5 Search for Multiple Instances of the Object (NumMatches, MaxOverlap)

With the parameter NumMatches you can specify the maximum number of instances of a component model that should be returned. With the parameter MaxOverlap you can specify to which degree instances may overlap. This parameter is suitable, e.g., to reject instances of the component model that differ only in the poses of one or a few components from an instance that was found with a higher score. Both parameters are similar to the corresponding parameters of shape-based matching (see section 3.2.4.6 on page 66).

3.3.4.6 Adjust the Behavior for the Case that Components are not found (IfRootNotFound, IfComponentNotFound, PosePrediction)

Sometimes a component is not found, e.g., because of occlusions. For example, in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-2D\cbm_dip_switch.hdev in one of the search images two instances of the model of a dip switch are found, but for one instance some components are occluded (see figure 3.30). That is, when calling find_component_model, the number of elements returned in the parameters RowComp, ColumnComp, AngleComp, ScoreComp, and ModelComp does not correspond to the number of expected components (taking the number of found instances into account). How to examine which specific components of which found instance are missing is described in section 3.3.5 on page 88.

Here, we show how to adjust the behavior of the search for the case that components are not found. In particular, you can

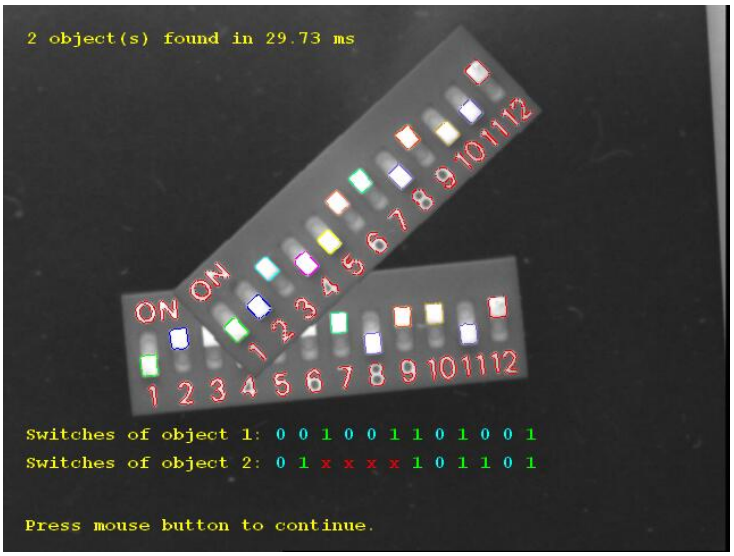


Figure 3.30: For one of two found instances of a component model some components are occluded.

- adjust the behavior of the search for the case that the root component is not found,
- adjust the behavior of the search for the case that any other component is not found, and
- adjust if the position and orientation of a not found component should be ignored or if it should be estimated based on the positions and orientations of the found components.

For the case that a root component is not found in a search image, the parameter `IfRootNotFound` must be adjusted. If the parameter is set to `'stop_search'` the search is aborted, i.e., for this instance of the model no other components are searched for. If it is set to `'select_new_root'` the other components are tested in the sequence that is given by `RootRanking`. Note that the latter proceeding is significantly slower than aborting the search.

For the case that at least one of the other components is not found, the parameter `IfComponentNotFound` must be adjusted. It controls how to handle a component that is searched relative to a component that has not been found. If `IfComponentNotFound` is set to `'prune_branch'` the component is not searched and therefore also not found. If it is set to `'search_from_upper'` the component is searched relative to another component. In particular, instead of the component with the previous position in the search tree, the component that is previous to the not found component is used as reference for the relative search. If `IfComponentNotFound` is set to `'search_from_best'` the component is searched relative to the component for which the runtime for the relative search is expected to be minimal.

For the case that not all components are found, you can further select if the 2D poses, i.e., the positions and orientations, are returned only for the found components (`PosePrediction` set to `'none'`) or if the positions and orientations of the not found components should be returned as an estimation that is based on the neighboring components (`PosePrediction` set to `'from_neighbors'`) or on all found components (`PosePrediction` set to `'from_all'`). Note that if an estimated position and orientation is returned, the corresponding return value of `ScoreComp` is `0.0`.

3.3.4.7 Adjust the Shape-Based Matching that is Used to Find the Components (`MinScoreComp`, `SubPixelComp`,`NumLevelsComp`, `GreedinessComp`)

The internal search for the individual components is based on shape-based matching. Therefore, the parameters `MinScoreComp`, `SubPixelComp`, `NumLevelsComp`, and `GreedinessComp` correspond to the parameters `MinScore`, `SubPixel`, `NumLevels`, and `Greediness` that are used for shape-based matching (see section 3.2.4 on page 61). Each of the parameters must contain either a single value that is used for all components or the number of values must correspond to the number of components. The number of values for `NumLevelsComp` can also be two or twice the number of components. For further information, please refer to the Reference Manual entry of `find_component_model`.

3.3.5 Use the Specific Results of Component-Based Matching

Each instance of a component model consists of several components that can change their spatial relations. Therefore, the result of the component-based matching is a bit more complex than that of a shape-based matching. In particular, the positions and orientations that are returned do not consist of a single value for each found instance of the model, but consist of values for all returned components. Parameters that return values for all components are `RowComp`, `ColumnComp`, `AngleComp`, and `ScoreComp`. The latter returns the scores of the individual components. From these, the score for each found instance of the component model (`Score`) is derived.

To know to which component of which found model instance a specific index position of the tuples `RowComp`, `ColumnComp`, `AngleComp`, and `ScoreComp` refers, the parameters `ModelStart`, `ModelEnd`, and `ModelComp` are provided. `ModelStart` and `ModelEnd` are needed to know how many instances are returned and to which instance the returned components belong. `ModelComp` is needed to know which specific components are returned and therefore also which specific components are not found.

In particular, the number of returned instances of the component model can be derived from the length of the tuples `ModelStart` and `ModelEnd`. In the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-2D\cbm_dip_switch.hdev`, e.g., for the search image shown in figure 3.30 on page 87, two model instances have been found, i.e., the tuples `ModelStart` and `ModelEnd` both consist of two values.

The actual values of the tuples indicate the intervals of the index positions of the tuples `RowComp`, `ColumnComp`, `AngleComp`, and `ScoreComp` that refer to the same instance of the component model. In the example program, `ModelStart` is [0,13] and `ModelEnd` is [12,21]. That is, the values with the index positions 0 to 12 refer to the first instance of the component model and the values with the index positions 13 to 21 refer to the second model instance. As the component model consists of 13 initial components, we see that for the first model instance all components have been found, whereas for the second model instance only 9 of 13 components have been found

Now we know the number of returned instances and for each instance we know the number of returned components and their positions, orientations, and scores. But we do not yet know, which of the 13 initial components of the model are missing for the second model instance. For this knowledge, we need the parameter `ModelComp` that indicates which specific components are found (see figure 3.31). In the example, `ModelComp` is [0,1,2,3,4,5,6,7,8,9,10,11,12,0,1,2,7,8,9,10,11,12]. That is, for the first model instance the components 0 to 12, i.e., all components of the component model have been found. For the second model instance, the components 0 to 2 and the components 7 to 12 have been found. Thus, the components 3 to 6 are missing.

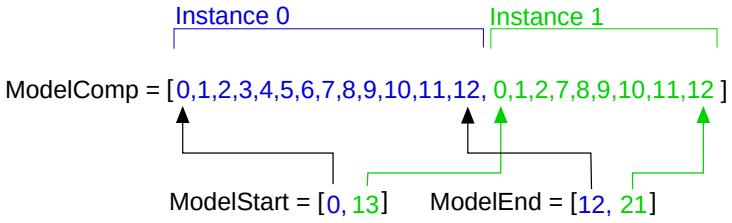


Figure 3.31: Meaning of `ModelStart`, `ModelEnd`, and `ModelComp`.

After the matching the components of each found model instance can be queried with `get_found_component_model`, so that the result can be visualized as is shown, e.g., for the dip switches in figure 3.30 on page 87 and figure 3.32. Besides the visualization of the found components by overlaying their regions on the search image, it is sometimes suitable to apply additional application-specific visualization steps. Here, e.g., a procedure for the textual interpretation of the result is applied (`visualize_bin_switch_match`).

```
NumFound := |ModelStart|
for Match := 0 to |ModelStart| - 1 by 1
    get_found_component_model (FoundComponents, ComponentModelID, \
                               ModelStart, ModelEnd, RowComp, \
                               ColumnComp, AngleComp, ScoreComp, \
                               ModelComp, Match, 'false', RowCompInst, \
                               ColumnCompInst, AngleCompInst, \
                               ScoreCompInst)
    dev_display (FoundComponents)
    visualize_dip_switch_match (RowCompInst, ColumnCompInst, \
                                AngleCompInst, RowRef, ColumnRef, \
                                AngleRef, WindowHandle, Match)
endfor
```

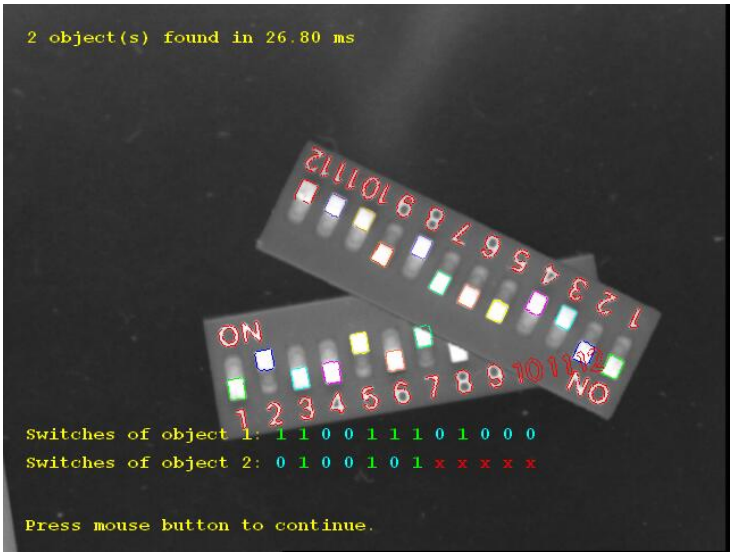


Figure 3.32: Result of component-based matching: two instances of a component model showing a dip switch are found, although some components are occluded.

3.4 Local Deformable Matching

Like shape-based matching, local deformable matching extracts contours and matches their shapes against the shapes of previously created models. But in contrast to shape-based matching, slightly deformed contours are not only found but the deformations are returned as result. Note that the actual location of the object is restricted to determining its position, whereas the orientation and scale are interpreted as part of the deformation.

The following sections show

- a first example for a local deformable matching (section 3.4.1),
- how to select an appropriate ROI to derive the template image from the reference image (section 3.4.2 on page 92),
- how to create a suitable model (section 3.4.3 on page 93),
- how to optimize the search (section 3.4.4 on page 95), and
- how to deal with the results that are specific for the local deformable matching (section 3.4.5 on page 97).



Figure 3.33: (left) "MVTec" logo used for the model creation; (right) deformed logo instance overlaid by the model contours.

3.4.1 A First Example

In this section we give a quick overview of the matching process with local deformable matching. To follow the example actively, start the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Matching\Deformable\find_local_deformable_model.hdev`, which locates differently deformed "MVTec" logos (see figure 3.33).

Step 1: Prepare the template

First, the template is prepared. In particular, the procedure `create_mvtec_logo_broadened` extracts the "MVTec" logo from an image and creates a synthetical image with a slightly broader logo. The obtained colored image (figure 3.33, left) is transformed into a gray value image from which an ROI, i.e., the template image (figure 3.34 on page 91, left) is derived.

```
create_mvtec_logo_broadened (LogoImage, 0, 200, Width, Height)
rgb1_to_gray (LogoImage, GrayImage)
gen_rectangle1 (Rectangle, 82, 17, 177, 235)
reduce_domain (GrayImage, Rectangle, ImageReduced)
```

Step 2: Create the model

The template image is then used to create a model of the logo using `create_local_deformable_model`. The contours of the model can be queried with `get_deformable_model_contours`, e.g., to overlay a later match by it to visually inspect the deformations (see figure 3.33, right).

```
create_local_deformable_model (ImageReduced, 'auto', [], [], 'auto', 1, [], \
                               'auto', 1, [], 'auto', 'none', \
                               'use_polarity', 'auto', 'auto', [], [], \
                               ModelID)
get_deformable_model_contours (ModelContours, ModelID, 1)
```

Step 3: Find the object again

The created model is used to find instances of the logo in search images. For demonstration purposes, the example uses synthetic search images that show the logo with various random deformations. After transforming the colored images again into the corresponding gray value images (see, e.g., figure 3.34, middle), the matching is applied with `find_local_deformable_model`.



Figure 3.34: From left to right: template image, corresponding part of the search image, rectified image.

```
rgb1_to_gray (SearchImage, GrayImage)
find_local_deformable_model (GrayImage, ImageRectified, VectorField, \
                             DeformedContours, ModelID, 0, 0, 1, 1, 1, 1, \
                             0.5, 1, 1, 4, 0.9, ['image_rectified', \
                            'vector_field','deformed_contours'], \
                             ['deformation_smoothness','expand_border', \
                            'subpixel'], [Smoothness,0,1], Score, Row, \
                             Column)
```

By default, the operator returns the position of the object in the image. With the parameter `ResultType`, you can additionally specify some iconic objects that are returned. In the example program, all available iconic objects, i.e., a rectified version of the part of the search image that corresponds to the bounding box of the ROI that was used to create the model (see figure 3.34, right), the vector field that describes the deformations of the matched model instance, and the contours of the deformed model instance are queried.

Step 4: Visualize the deformations

To visualize the deformations of a found model instance, the procedure `gen_warped_mesh` creates a regular grid and deforms it with the information that is contained in the returned vector field. For that, the vector field is first converted into the two real-valued images `DRow` and `DCol` using `vector_field_to_real`. For each point of the model, or more precisely of the bounding box that surrounds the template image, `DRow` contains the corresponding row coordinates and `DCol` the corresponding column coordinates of the search image. Then, a regular grid with the size of the model is created. The horizontal and vertical grid lines are created in separate loops using the operators `tuple_gen_sequence` and `tuple_gen_const`. Using the images `DRow` and `DCol`, for each point of a grid line `get_grayval_interpolated` queries the corresponding "deformed" point, i.e., the corresponding coordinates of the point in the search image. These coordinates are used to create polygons that represent the deformed horizontal and vertical grid lines.

```
procedure gen_warped_mesh (VectorField, WarpedMesh, Step)
gen_empty_obj (WarpedMesh)
count_obj (VectorField, Number)
for Index := 1 to Number by 1
    select_obj (VectorField, ObjectSelected, Index)
    vector_field_to_real (ObjectSelected, DRow, DCol)
    get_image_size (VectorField, Width, Height)
    for ContR := 0.5 to Height[0] - 1 by Step
        Col1 := [0.5:Width[0] - 1]
        tuple_gen_const (Width[0] - 1, ContR, Row1)
        get_grayval_interpolated (DRow, Row1, Col1, 'bilinear', GrayRow)
        get_grayval_interpolated (DCol, Row1, Col1, 'bilinear', GrayCol)
        gen_contour_polygon_xld (Contour, GrayRow, GrayCol)
        concat_obj (WarpedMesh, Contour, WarpedMesh)
    endfor
    for ContC := 0.5 to Width[0] - 1 by Step
        Row1 := [0.5:Height[0] - 1]
        tuple_gen_const (Height[0] - 1, ContC, Col1)
        get_grayval_interpolated (DRow, Row1, Col1, 'bilinear', GrayRow)
        get_grayval_interpolated (DCol, Row1, Col1, 'bilinear', GrayCol)
        gen_contour_polygon_xld (Contour, GrayRow, GrayCol)
        concat_obj (WarpedMesh, Contour, WarpedMesh)
    endfor
endfor
return ()
```

The grid is displayed together with the returned contours of the deformed logo as shown in figure 3.35.

```
dev_display (SearchImage)
dev_display (WarpedMesh)
dev_display (DeformedContours)
```

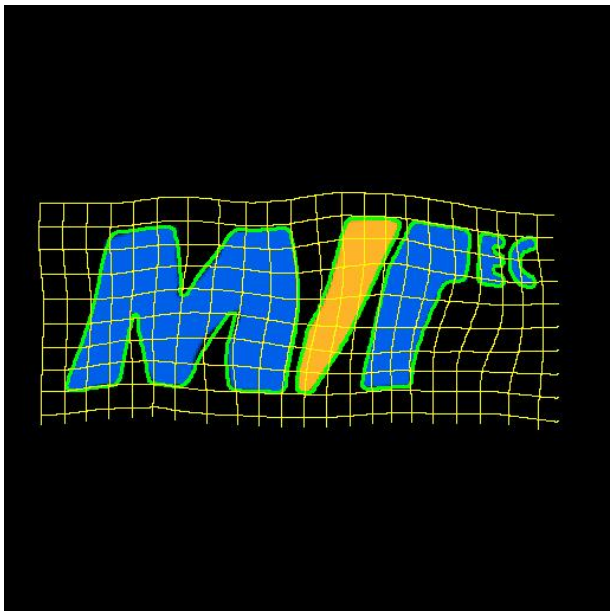


Figure 3.35: Search image (before transforming it into a gray value image) with the grid that illustrates the deformations of the matched “MVTec” logo.

The difference between the returned rectified image and the template image is visualized as shown in figure 3.36.

```
crop_domain (ImageReduced, ImagePart)
abs_diff_image (ImagePart, ImageRectified, ImageAbsDiff1, 1)
dev_display (ImageAbsDiff1)
```



Figure 3.36: Difference between the rectified image and the template image.

The following sections go deeper into the details of the individual steps of a local deformable matching and the parameters that have to be adjusted.

3.4.2 Select the Model ROI

As a first step of the local deformable matching, the region of interest that specifies the template image must be selected as described, e.g., in section 2.1.1 on page 17. This region can have an arbitrary shape, i.e., it can also contain holes or consist of several parts that are not connected. Thus, it is possible to create an ROI that contains only “robust” parts of the object. The ROI must be selected so that it contains all prominent structures of the object and also some pixels outside of them, as their immediate surroundings (neighborhood) are needed to obtain the

model. Furthermore, you can speed up the later search using a subsampling (see section 3.4.3.2 on page 94). For that, the ROI should not be too "thin", because otherwise it vanishes at higher pyramid levels! As a rule of thumb, you are on the safe side if an ROI is $2^{NumLevels-1}$ pixels wide. That is, a width of 8 pixels allows to use 4 pyramid levels. After having selected a suitable ROI, the reduced image is used as template image for the creation of the model.

3.4.3 Create a Suitable Local Deformable Model

Having derived the template image from the reference image, the local deformable model can be created. Note that the local deformable matching consists of different methods to find the trained objects in images. Depending on the selected method, one of the following operators is used to create the model:

- `create_local_deformable_model` creates a model for a local deformable matching that uses a template image to derive the model.
- `create_local_deformable_model_xld` creates a model for a local deformable matching that uses XLD contours to derive the model. Note that after a first match, it is strongly recommended to determine the polarity information for the model with `set_local_deformable_model_metric` (see section 2.1.3.2 on page 22 for details).

Here, we will take a closer look at how to adjust the corresponding parameters. In particular, you can

- specify which pixels are part of the model by adjusting the parameter `Contrast` (section 3.4.3.1),
- speed up the search by using a subsampling, i.e., by adjusting the parameter `NumLevels`, and by reducing the number of model points, i.e., by adjusting the parameter `Optimization` (section 3.4.3.2),
- specify which pixels are compared with the model in the later search by adjusting the parameters `MinContrast` and `Metric` (section 3.4.3.4), and
- adjust some additional (generic) parameters within `GenParamName` and `GenParamValue`, which is needed only in very rare cases (section 3.4.3.5).

Note that when adjusting the parameters you can also let HALCON assist you:

- Use automatic parameter suggestion:

  You can let HALCON suggest suitable values for many of these parameters by either setting the corresponding parameters to the value `'auto'` within the operator that is used to create the model or by applying `determine_deformable_model_params` to automatically determine values for a local deformable model from a template image and then decide individually whether you use the suggested values for the creation of the model. Note that both approaches return only approximately the same values and the values that are returned by the operators that are used to create the model are more precise.

- Apply `inspect_shape_model`:

  As the local deformable matching uses XLD contours to build the model, which is similar to shape-based matching, you can use `inspect_shape_model` to try different values for the parameters `NumLevels` and `Contrast`. The operator returns the resulting representation of a shape model with multiple pyramid levels and thus allows you to visually check if the object of interest is represented adequately by the model. If several combinations of parameter values do not lead to a satisfying representation of the object, maybe the template image, i.e., the model's ROI was not selected properly. How to select a suitable ROI is described in section 2.1 on page 17.

Note that after the creation of the model, the model can still be modified. In section 3.4.3.6 on page 95 the possibilities for the inspection and modification of an already created model are shown.

3.4.3.1 Specify Pixels that are Part of the Model (Contrast)

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_local_deformable_model. The parameter Contrast is similar to the corresponding parameter of shape-based matching that is described in more detail in section 3.2.3.1 on page 57. The only difference is that here small structures are not suppressed within Contrast but with the separate generic parameter 'min_size' that is set via GenParamName and GenParamValue as described in section 3.4.3.5.

3.4.3.2 Speed Up the Search using Subsampling and Point Reduction (NumLevels, Optimization)

To speed up the matching process, subsampling can be used (see also section 2.3.2 on page 25). There, an 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.

You can specify how many pyramid levels are used via the parameter NumLevels. A further reduction of model points can be enforced via the parameter Optimization. This may be useful to speed up the matching in the case of particularly large models. Both parameters are similar to the corresponding parameters of shape-based matching that are described in more detail in section 3.2.3.2 on page 58.

3.4.3.3 Allow a Range of Orientation (AngleExtent, AngleStart, AngleStep) and Scale (Scale*Min, Scale*Max, Scale*Step)

Note that AngleStart, AngleExtent, ScaleRMax, and ScaleCMax are not used by this operator. Instead, they have to be specified within the operator find_local_deformable_model.

ScaleRMin and ScaleCMin can be relevant if you expect matches that are smaller than in the original image Template. Then, if NumLevels is set to *'auto'*, the number of used pyramid levels may decrease.

The parameters AngleStep, ScaleRStep, and ScaleCStep can be determined automatically by setting their values to 'auto'. Generally, this should lead to good results.

For more information about how these parameters work, please refer to the operator find_local_deformable_model.

3.4.3.4 Specify which Pixels are Compared with the Model (MinContrast, Metric)

The parameter MinContrast lets you specify which contrast a point in a search image must at least have in order to be compared with the model, whereas Metric lets you specify whether and how the polarity, i.e., the direction of the contrast must be observed.

The parameters are similar to the corresponding parameters of shape-based matching that are described in section 3.2.3.5 on page 60. The only difference is that for local deformable matching a further value for Metric is available that allows to observe the polarity in sub-parts of the model. A sub-part is a set of model points that are adjacent. For different calculations of the local deformable matching, the model is divided into a set of sub-parts that have approximately the same size (see also section 3.5.3.5 on page 103). When setting Metric to 'ignore_part_polarity', the different sub-parts may have different polarities as long as the polarities of the points within the individual sub-parts are the same. 'ignore_part_polarity' is a trade-off between 'ignore_global_polarity', which does not work with locally changing polarities, and 'ignore_local_polarity', which requires a very fine structured inspection and thus slows down the search significantly.

3.4.3.5 Adjust Generic Parameters (GenParamName, GenParamValue)

Typically, there is no need to adjust the generic parameters that are set via GenParamName and GenParamValue. But in rare cases, it might be helpful to adjust the splitting of the model into sub-parts, which are needed for different calculations of the local deformable matching, or to suppress small connected components of the model contours.

The size of the sub-parts is adjusted setting GenParamName to 'part_size' and GenParamValue to 'small', 'medium', or 'big'. Small connected components of the model contours can be supressed by setting GenParamName to 'min_size'. The corresponding numeric value that is specified in GenParamValue describes the number of points a connected part of the object must at least contain to be considered for the following calculations. The effect corresponds to the suppression of small structures that can be applied for shape-based matching within the parameter Contrast, which is described in section 3.2.3.1 on page 57.

3.4.3.6 Inspect and Modify the Local Deformable Model

If you want to visually inspect an already created deformable model, you can use get_deformable_model_contours to get the XLD contours that represent the model in a specific pyramid level. Note that the XLD contour of the model is located at the origin of the image and thus a transformation may be needed for a proper visualization (see section 2.4.2 on page 29).

To inspect the current parameter values of a model, you query them with get_deformable_model_params. This may be necessary if during the creation of the model an automatic parameter selection was used or if the model was created within another program, saved to file with write_deformable_model, and read from this file in the current program with read_deformable_model. Additionally, you can query the coordinates of the origin of the model using get_deformable_model_origin.

After the creation of the model and before you search the object in a search image, you can further modify the model. In particular, you can apply set_deformable_model_origin to change its point of reference. But similar to shape-based matching, when modifying the point of reference, the accuracy of the estimated position may decrease (see section 3.2.4.8 on page 68). Therefore, **if possible, the point of reference should not be changed.** Instead, a suitable ROI for the model creation should be selected right from the start (see section 2.1.2 on page 18).



3.4.4 Optimize the Search Process

The actual matching is applied with find_local_deformable_model. In the following, we show how to select suitable parameters for it to adapt and optimize a matching task. In particular, we show how to

- restrict the search space to a region of interest (section 3.4.4.1),
- restrict the search space by restricting the range of orientation and scale via the parameters AngleStart, AngleExtent, ScaleMin, and ScaleMax (section 3.4.4.2),
- restrict the search space to a specific amount of allowed occlusions for the object, i.e., specify the visibility of the object via the parameter MinScore (section 3.4.4.3),
- specify the used search heuristics, i.e., trade thoroughness versus speed by adjusting the parameter Greediness (section 3.4.4.4),
- search for multiple instances of the model by adjusting the parameters NumMatches and MaxOverlap (section 3.4.4.5),
- restrict the number of pyramid levels (NumLevels) for the search process (section 3.4.4.6),
- specify the types of iconic results the matching should return (ResultType, section 3.4.4.6), and
- adjust some additional (generic) parameters within GenParamName and GenParamValue (section 3.4.4.8).

3.4.4.1 Restrict the Search to a Region of Interest

The obvious way to restrict the search space and thus speed up the matching is to apply the operator find_local_deformable_model not to the whole image but only to an ROI. The corresponding procedure is explained in more detail for shape-based matching in section 3.2.4.1 on page 63. For local deformable matching you simply have to replace find_shape_model by find_local_deformable_model.

3.4.4.2 Restrict the Range of Orientation and Scale (AngleStart, AngleExtent, Scale*Min, Scale*Max)

When calling find_local_deformable_model you can limit the orientation and scale 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.

3.4.4.3 Specify the Visibility of the Object (MinScore)

With the parameter MinScore you can specify how much of the model must be visible. A typical use of this mechanism is to allow a certain degree of occlusion. The parameter is similar to the corresponding parameter for shape-based matching that is described in more detail in section 3.2.4.3 on page 64.

3.4.4.4 Trade Thoroughness vs. Speed (Greediness)

With the parameter Greediness you can influence the search algorithm itself and thereby trade thoroughness against speed. The parameter is similar to the corresponding parameter for shape-based matching that is described in more detail in section 3.2.4.5 on page 65.

3.4.4.5 Search for Multiple Instances of the Object (NumMatches, MaxOverlap)

All you have to do to search for more than one instance of the object is to set the parameter NumMatches to the maximum number of instances you want to find. If you select the value 0, all matches are returned.

Locating more than one instance, the operator find_local_deformable_model returns the results for the individual model instances concatenated in the output tuples (see also section 3.4.5.4 on page 98). Note that a search for multiple objects is only slighly slower than a search for a single object.

A second parameter, MaxOverlap, lets you specify how much two matches may overlap (as a fraction). This parameter is similar to the corresponding parameter of shape-based matching that is described in section 3.2.4.6 on page 66.

3.4.4.6 Restrict the Number of Pyramid Levels (NumLevels)

The parameter NumLevels, which you already specified when creating the model, allows you to use a different (in most cases a more restrictive) value in the search process. By using the value 0 for NumLevels, the value specified when creating the model is used.

3.4.4.7 Specify the Iconic Objects Returned by the Matching (ResultType)

By default, a position and a score are returned for each match. If you additionally want to obtain some iconic results, you can set the parameter ResultType to 'image_rectified', 'vector_field', and 'deformed_contours'. With 'image_rectified' you get a rectified version of the part of the search image that corresponds to the bounding box of the ROI that was used to create the model (RectifiedImage), with 'vector_field' you get the vector field that describes the deformations of the matched model instance (VectorField), and with 'deformed_contours' you get the contours of the deformed model instance (DeformedContours). The individual result types are introduced in more detail in section 3.4.5.

3.4.4.8 Adjust Generic Parameters (GenParamName, GenParamValue)

Besides the parameters that correspond more or less to those used for shape-based matching, local deformable matching allows to adjust some generic parameters that are set via GenParamName and GenParamValue. Typically, there is no need to adjust them. But if, e.g., a low accuracy of the matching is sufficient, you can speed up the matching by a reduction or deactivation of the subpixel precision or by changing the discretization steps for the orientation or scale that were set during the creation of the model.

Additionally, you can adjust the expected "smoothness" of the deformation. In particular, if the deformations vary locally, i.e., if the deformations are expected to be rather different within a small neighborhood, the value of `'deformation_smoothness'` may be set to a smaller value, whereas for an object with an expected "smooth" transition between the deformations the parameter may be set to a higher value.

Furthermore, you can expand the border of the optionally returned rectified image (see section 3.4.4.7 on page 96). By default, the section of the search image that is rectified corresponds to the bounding box of the ROI that was used to create the model. If a larger section is needed, e.g., if neighboring parts that are visible in the search image are needed as well for a further processing of the image, you can expand the border of the rectified image with the parameters `'expand_border'`, `'expand_border_top'`, `'expand_border_bottom'`, `'expand_border_left'`, or `'expand_border_right'`, respectively. If, e.g., in the example program only the letters "MVT" would have been contained in the model, but the letters "ec" would have been needed as well, the parameter `'expand_border_right'` could have been set to 50 to expand the image section by 50 pixels. Note that the rectification is reliable only for the image section that corresponds to the model ROI, as for the neighboring parts of the search image an extrapolation is applied. Thus, the larger the expanded border is, the less accurate is the result towards the image border.

For further information, please refer to the description of `find_local_deformable_model` in the Reference Manual.

3.4.5 Use the Specific Results of Local Deformable Matching

The results of local deformable matching differ from those of the other matching approaches. Though it returns also a position and a score, it does not return an orientation or even a scale. Instead, a set of iconic objects can be returned that helps to inspect the deformations of the found object instance. Depending on the selected values of `ResultType`, the following iconic objects are returned:

- `RectifiedImage`: the rectified part of the search image that corresponds to the bounding box of the ROI that was used to create the model (section 3.4.5.1).
- `VectorField`: the vector field describing the deformations of the found model instance (section 3.4.5.2).
- `DeformedContours`: the contours of the deformed model instance (section 3.4.5.3).

Similar to shape-based matching, local deformable matching can return multiple instances of a model. How to use these multiple instances is shown in section 3.5.5.1 on page 107.

3.4.5.1 The Rectified Image

The first of the iconic objects that can be returned by the local deformable matching is the rectified part of the search image that corresponds to the bounding box of the ROI that was used to create the model (see figure 3.37).

It can be used, e.g., as was introduced in the first example to show the difference between the rectified image and the corresponding part of the reference image (see section 3.4.1 on page 90). Another use may be that the rectified image is needed for a further image processing task. Note that if a larger section of the search image is needed for the further processing, it is possible to expand the borders of `ImageRectified` as is described in section 3.4.4.8 on page 96.

3.4.5.2 The Vector Field

Another iconic object that is returned by the matching is a vector field that can be used, e.g., to visualize the deformations. For that, `VectorField` can be transformed into a grid as was shown in the first example (page 91). Note that to directly display the vector field is not reasonable, as in contrast to a vector field that is returned, e.g., by optical flow, no relative but absolute coordinates are returned. In particular, the vector field that is returned, e.g., by `optical_flow_mg` describes the relative movement of the points from one image to another image of the same size. The components of the vector field that can be queried with `vector_field_to_real` contain for each point the relative movement in row and column direction. In contrast, the vector field that is returned by local deformable matching describes the movement of points from the model, or more precisely from the bounding box that surrounds the template image, to a search image that typically is larger than the model. Thus, instead



Figure 3.37: (left) search image; (right) rectified image part.

of the relative movement of each point, the vector field contains for each model point the corresponding absolute coordinates of the search image.

Besides visualization purposes, the vector field can be used to get a rectification map that can be used for other applications as well. If, e.g., the deformations of a model instance do not stem from a deformed object but from significant camera distortions, you may use the returned vector field to compensate for the camera distortions in other images. For that, the model either must be a synthetic model (see section 2.1.3 on page 20) or it must be created from an image that was made with a camera without significant distortions. The vector field that is returned by the matching in a search image that is made by the distorted camera can then be transformed into a map with `convert_map_type`. This map can then be used to rectify other images that were made with the same camera.

3.4.5.3 The Deformed Contours

At last, the deformed contours can be returned. Similar to the model contours the `DeformedContours` can be used for visualization purposes (see figure 3.38) or for a visual inspection of the difference between the model and the found model instance. For the latter, you can either overlay the matching result with the model contours or overlay the model with the deformed contours. Whereas the model contours by default are located at the origin of the image and thus must be transformed for a proper visualization as described in section 2.4.3.1 on page 33, the deformed contours are located already at the position of the match.

3.4.5.4 Deal with Multiple Matches

If multiple instances of the object are searched and found, the results are concatenated in tuples. For example, `Score`, which is a single value for a single model instance, is now a tuple for which the number of elements corresponds to the number of found model instances. The access of the value for a specific instance is similar to the proceeding described in section 3.2.4.6 on page 66 for shape-based matching.

The iconic output object `DeformedContours` is already returned as a tuple for a single model instance. For multiple instances, the values are simply concatenated. To access them for a specific match, you must know the number of elements returned for a single model instance. This can be obtained by calling `count_obj` for the model contours that can be queried with `get_deformable_model_contours` after the model creation. If, e.g., the model consists of five contours, in the tuple returned for the deformed contours, the first five values belong to the first instance, the next five values belong to the second instance, etc. An example for the access of the values for each instance is described in more detail for calibrated perspective deformable matching in section 3.5.4.5 on page 106.



Figure 3.38: (left) search image; (right) search image overlaid by deformed contours.

3.5 Perspective Deformable Matching

Like shape-based matching, perspective deformable matching extracts contours and matches their shapes against the shapes of previously created models. But in contrast to shape-based matching, also perspectively deformed contours can be found. For the perspective deformable matching, an uncalibrated as well as a calibrated version is provided. The following sections show

- a first example for a perspective deformable matching (section 3.5.1),
- how to select an appropriate ROI to derive the template image from the reference image (section 3.5.2 on page 101),
- how to create a suitable model (section 3.5.3 on page 101),
- how to optimize the search (section 3.5.4 on page 104), and
- how to deal with the results that are specific for the perspective deformable matching (section 3.5.5 on page 107).

3.5.1 A First Example

In this section we give a quick overview of the matching process with perspective deformable matching. To follow the example actively, start the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Traffic-Monitoring\detect_road_signs.hdev`, which locates road signs independently from their direction. Actually, the program searches an attention sign and a dead end road sign. In the following, we show the proceeding exemplarily for the attention sign.

Step 1: Select the object in the reference image

First, the colored reference image is read and a channel that clearly shows the attention sign (see figure 3.39, left) is accessed with `access_channel`. As the attention sign in the search images is expected to be much smaller than in the reference image, the reference image is scaled with `zoom_image_factor` to better fit the expected size (see figure 3.39, middle). The creation of perspective deformable models is based on XLD contours, which is similar to shape-based matching. Thus, `inspect_shape_model` (see section 3.2.3 on page 55) can be used to get suitable values for the parameters `NumLevels` and `Contrast`, which are needed for the creation of shape models as well as for the creation of perspective deformable models. The returned representation of a shape model with multiple pyramid levels can be used to visually check a potential model (see figure 3.39, right).

```
read_image (ImageAttentionSign, 'road_signs/attention_road_sign')
access_channel (ImageAttentionSign, Image, Channel[0])
zoom_image_factor (Image, ImageZoomed, 0.1, 0.1, 'weighted')
inspect_shape_model (ImageZoomed, ModelImages, ModelRegions, 3, 20)
```

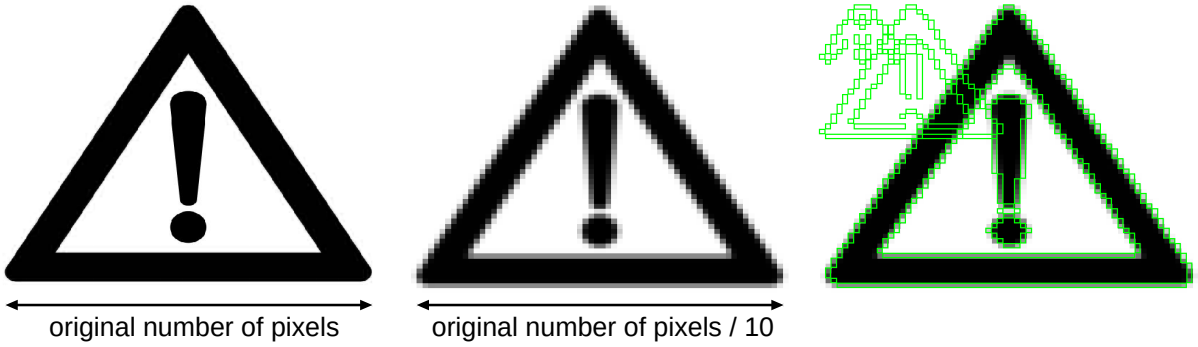


Figure 3.39: From left to right: specific channel of colored reference image, scaled image, inspection of the shape model.

Step 2: Create the model

How to obtain a proper template image from a reference image is described in section 2.1 on page 17. In this case, a further reduction to a more restricted ROI is not needed. The model is obtained from the template image with `create_planar_uncalib_deformable_model`.

```
create_planar_uncalib_deformable_model (ImageZoomed, 3, [], [], 0.1, \
                                        ScaleRMin[0], [], 0.05, \
                                        ScaleCMin[0], [], 0.5, 'none', \
                                        'use_polarity', 'auto', 'auto', [], \
                                        [], ModelID)
```

Step 3: Find the object again

To speed up the matching, the search space is restricted first to a rectangular region and then, within this region, to an ROI consisting of a set of blobs (see figure 3.40, left). This set of blobs is obtained by the procedure `determine_area_of_interest`, which exploits the available color information and applies a blob analysis.

```
gen_rectangle1 (Rectangle1, 28, 71, 69, 97)
for Index := 1 to 16 by 1
    read_image (Image, 'road_signs/street_' + Index$'.02')
    determine_area_of_interest (Image, Rectangle, AreaOfInterest)
    reduce_domain (Image, AreaOfInterest, ImageReduced)
```

Within the resulting ROI, the actual matching is applied using the operator `find_planar_uncalib_deformable_model`. Note that the search is applied in the same image channel that was already used for the model creation.

```
    for Index2 := 0 to |Models| - 1 by 1
        access_channel (ImageReduced, ImageChannel, Channel[Index2])
        find_planar_uncalib_deformable_model (ImageChannel, Models[Index2], \
                                              0, 0, ScaleRMin[Index2], \
                                              ScaleRMax[Index2], \
                                              ScaleCMin[Index2], \
                                              ScaleCMax[Index2], 0.85, 1, \
                                              0, 2, 0.4, [], [], HomMat2D, \
                                              Score)
```

The result of the matching for each found model instance is a 2D projective transformation matrix (homography) and a score that represents the quality of the matching. Using the homographies, the result is visualized for the

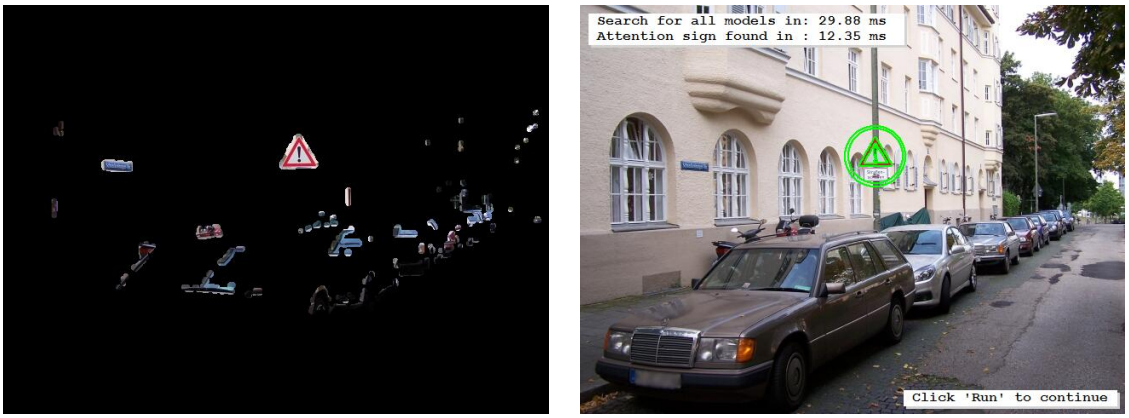


Figure 3.40: (left) ROI derived from the color image; (right) found match for the model of the attention sign.

successful matches. In particular, the contours of the model are queried and projected into the search image as described in more detail in section 2.4.5 on page 42 (see figure 3.40, right).

```
        if (|HomMat2D|)
            get_deformable_model_contours (ModelContours, Models[Index2], 1)
            projective_trans_contour_xld (ModelContours, ContoursProjTrans, \
                                          HomMat2D)
        endif
    endfor
endfor
```

The following sections go deeper into the details of the individual steps of a perspective deformable matching and the parameters that have to be adjusted.

3.5.2 Select the Model ROI

As a first step of the perspective deformable matching, the region of interest that specifies the template image must be selected as described, e.g., in section 2.1.1 on page 17. This region can have an arbitrary shape, i.e., it can also contain holes or consist of several parts that are not connected. Thus, it is possible to create an ROI that contains only "robust" parts of the object. The ROI must be selected so that it contains all prominent structures of the object and also some pixels outside of them, as their immediate surroundings (neighborhood) are needed to obtain the model. Furthermore, you can speed up the later search using a subsampling (see section 3.5.3.2 on page 103). For that, the ROI should not be too "thin", because otherwise it vanishes at higher pyramid levels! As a rule of thumb, you are on the safe side if an ROI is $2^{NumLevels-1}$ pixels wide. That is, a width of 8 pixels allows to use 4 pyramid levels. After having selected a suitable ROI, the reduced image is used as template image for the creation of the model.

3.5.3 Create a Suitable Perspective Deformable Model

Having derived the template image from the reference image, the perspective deformable model can be created. Note that the perspective deformable matching consists of different methods to find the trained objects in images. Depending on the selected method, one of the following operators is used to create the model:

- `create_planar_uncalib_deformable_model` creates a model for an uncalibrated perspective deformable matching that uses a template image to derive the model.
- `create_planar_uncalib_deformable_model_xld` creates a model for an uncalibrated perspective deformable matching that uses XLD contours to derive the model. Note that after a first match, it is strongly recommended to determine the polarity information for the model with `set_planar_uncalib_deformable_model_metric` (see section 2.1.3.2 on page 22 for details).

- create_planar_calib_deformable_model creates a model for a calibrated perspective deformable matching that uses a template image to derive the model.
- create_planar_calib_deformable_model_xld creates a model for a calibrated perspective deformable matching that uses XLD contours to derive the model. Note that after a first match, it is strongly recommended to determine the polarity information for the model with set_planar_calib_deformable_model_metric (see section 2.1.3.2 on page 22 for details).

Here, we will take a closer look at how to adjust the corresponding parameters. In particular, you can

- specify which pixels are part of the model by adjusting the parameter Contrast (section 3.5.3.1),
- speed up the search by using a subsampling, i.e., by adjusting the parameter NumLevels, and by reducing the number of model points, i.e., by adjusting the parameter Optimization (section 3.5.3.2),
- specify which pixels are compared with the model in the later search by adjusting the parameters MinContrast and Metric (section 3.5.3.4),
- adjust some additional (generic) parameters within GenParamName and GenParamValue, which is needed only in very rare cases (section 3.5.3.5), and
- specify the camera parameters and the reference pose that are needed only for the calibrated matching (CamParam and ReferencePose, see section 3.6.3.3 on page 112).

Note that when adjusting the parameters you can also let HALCON assist you:

- Use automatic parameter suggestion:

  You can let HALCON suggest suitable values for many of these parameters by either setting the corresponding parameters to the value 'auto' within the operator that is used to create the model or by applying determine_deformable_model_params to automatically determine values for a perspective deformable model from a template image and then decide individually if you use the suggested values for the creation of the model. Note that both approaches return only approximately the same values and the values that are returned by the operators that are used to create the model are more precise.

- Apply inspect_shape_model:

  As the perspective deformable matching uses XLD contours to build the model, which is similar to shape-based matching, you can use inspect_shape_model to try different values for the parameters NumLevels and Contrast. The operator returns the resulting representation of a shape model with multiple pyramid levels and thus allows you to visually check if the object of interest is represented adequately by the model. If several combinations of parameter values do not lead to a satisfying representation of the object, maybe the template image, i.e., the model's ROI was not selected properly. How to select a suitable ROI is described in section 2.1 on page 17.

Note that after the creation of the model, the model can still be modified. In section 3.5.3.7 on page 104 the possibilities for the inspection and modification of an already created model are shown.

3.5.3.1 Specify Pixels that are Part of the Model (Contrast)

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_planar_uncalib_deformable_model or create_planar_calib_deformable_model. The parameter Contrast is similar to the corresponding parameter of shape-based matching that is described in more detail in section 3.2.3.1 on page 57. The only difference is that here small structures are not suppressed within Contrast but with the separate generic parameter 'min_size' that is set via GenParamName and GenParamValue as described in section 3.5.3.5.

3.5.3.2 Speed Up the Search using Subsampling and Point Reduction (NumLevels, Optimization)

To speed up the matching process, subsampling can be used (see also section 2.3.2 on page 25). There, an 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.

You can specify how many pyramid levels are used via the parameter NumLevels. A further reduction of model points can be enforced via the parameter Optimization. This may be useful to speed up the matching in the case of particularly large models. Both parameters are similar to the corresponding parameters of shape-based matching that are described in more detail in section 3.2.3.2 on page 58.

3.5.3.3 Allow a Range of Orientation (AngleExtent, AngleStart, AngleStep) and Scale (Scale*Min, Scale*Max, Scale*Step)

Note that AngleStart, AngleExtent, ScaleRMax, and ScaleCMax are not used by this operator. Instead, they have to be specified within the operator find_planar_uncalib_deformable_model.

ScaleRMin and ScaleCMin can be relevant if you expect matches that are smaller than in the original image Template. Then, if NumLevels is set to *'auto'*, the number of used pyramid levels may decrease.

The parameters AngleStep, ScaleRStep, and ScaleCStep can be determined automatically by setting their values to 'auto'. Generally, this should lead to good results.

For more information about how these parameters work, please refer to the operator find_planar_uncalib_deformable_model.

3.5.3.4 Specify which Pixels are Compared with the Model (MinContrast, Metric)

The parameter MinContrast lets you specify which contrast a point in a search image must at least have in order to be compared with the model, whereas Metric lets you specify whether and how the polarity, i.e., the direction of the contrast must be observed.

The parameters are similar to the corresponding parameters of shape-based matching that are described in section 3.2.3.5 on page 60. The only difference is that for perspective deformable matching a further value for Metric is available that allows to observe the polarity in sub-parts of the model. A sub-part is a set of model points that are adjacent. For different calculations of the perspective deformable matching, the model is divided into a set of sub-parts that have approximately the same size (see also section 3.5.3.5). When setting Metric to 'ignore_part_polarity', the different sub-parts may have different polarities as long as the polarities of the points within the individual sub-parts are the same. This is helpful, e.g., if due to the movement in the 3D space different reflections on the object are expected. As these reflections typically are not too small, 'ignore_part_polarity' is a trade-off between 'ignore_global_polarity', which does not work with locally changing polarities, and 'ignore_local_polarity', which requires a very fine structured inspection and thus slows down the search significantly.

3.5.3.5 Adjust Generic Parameters (GenParamName, GenParamValue)

Typically, there is no need to adjust the generic parameters that are set via GenParamName and GenParamValue. But in rare cases, it might be helpful to adjust the splitting of the model into sub-parts, which are needed for different calculations of the perspective deformable matching, or to suppress small connected components of the model contours.

The size of the sub-parts is adjusted setting GenParamName to 'part_size' and GenParamValue to 'small', 'medium', or 'big'.

Small connected components of the model contours can be supressed by setting GenParamName to 'min_size'. The corresponding numeric value that is specified in GenParamValue describes the number of points a connected part of the object must at least contain to be considered for the following calculations. The effect corresponds to the suppression of small structures that can be applied for shape-based matching within the parameter Contrast, which is described in section 3.2.3.1 on page 57.

3.5.3.6 Specify the Camera Parameters and the Reference Pose (CamParam, ReferencePose)

For the calibrated matching, the internal camera parameters (CamParam) and a reference pose (ReferencePose) have to be specified. We recommend to obtain both using a camera calibration as described in the Solution Guide III-C, section 3.2 on page 59. Other approaches for determining the pose of the object plane comprise a manual measurement of the extents of the model, which is rather intricate and often inaccurate, stereo vision (see Solution Guide III-C, chapter 5 on page 115), or 3D laser triangulation, e.g., using sheet of light (see Solution Guide III-C, chapter 6 on page 145).

3.5.3.7 Inspect and Modify the Perspective Deformable Model

If you want to visually inspect an already created deformable model, you can use get_deformable_model_contours to get the XLD contours that represent the model in a specific pyramid level. In case that the model was generated by create_planar_calib_deformable_model_xld, the contours by default are returned in the world coordinate system in metric units. Here, the contours must be transformed by the returned pose for visualizing a match. In all other cases, the contours of the model by default are returned in the image coordinate system in pixel units. For the calibrated matching you can specify the coordinate system in which the contours are returned when calling get_deformable_model_contours with the operator set_deformable_model_param.

To inspect the current parameter values of a model, you query them with get_deformable_model_params. This may be necessary if during the creation of the model an automatic parameter selection was used or if the model was created within another program, saved to file with write_deformable_model, and read from this file in the current program with read_deformable_model. Additionally, you can query the coordinates of the origin of the model using get_deformable_model_origin.

After the creation of the model and before you search the object in a search image, you can further modify the model. In particular, you can apply set_deformable_model_origin to change its point of reference and thus, in case of a calibrated matching, also its reference pose. But similar to shape-based matching, when modifying the point of reference, the accuracy of the estimated position may decrease (see section 3.2.4.8 on page 68). Therefore, **if possible, the point of reference should not be changed.** Instead, a suitable ROI for the model creation should be selected right from the start (see section 2.1.2 on page 18).

For the case that you nevertheless need to modify the point of reference, e.g., if the origin should be placed on a specific part of the object, e.g., a corner of the object or a drill hole, and you want to apply a calibrated matching, we here shortly describe the relation between the reference pose, the model pose, and an offset that is manually applied to the point of reference. In particular, when creating the template model for a calibrated matching, the reference pose, which was obtained, e.g., by a camera calibration, is automatically modified by an offset so that its origin corresponds to the projection of the center of gravity of a rectified version of the template image (i.e., the ROI) onto the reference plane and the axes of the obtained model coordinate system are parallel to those of the initial reference pose (see figure 3.41). If you use set_deformable_model_origin to additionally apply an offset to the origin of the thus obtained model pose, the offset is set in image coordinates for the internally rectified template image. To determine the necessary number of pixels for the row and column direction it is convenient to query the rectified contour of the object from the template model with get_deformable_model_contours. After setting a manual offset, the obtained image point is automatically projected to the object plane as origin of the modified pose and the pose is adapted accordingly for all following operations.

3.5.4 Optimize the Search Process

The actual matching is applied by one of the following operators:

- find_planar_uncalib_deformable_model searches for the best matches of an uncalibrated perspective deformable model. It returns a 2D projective transformation matrix (homography) and the score describing the quality of the match.
- find_planar_calib_deformable_model searches for the best matches of a calibrated perspective deformable model. It returns the 3D pose of the object, the six mean square deviations, respectively the 36 covariances of the pose parameters, and the score describing the quality of the match.

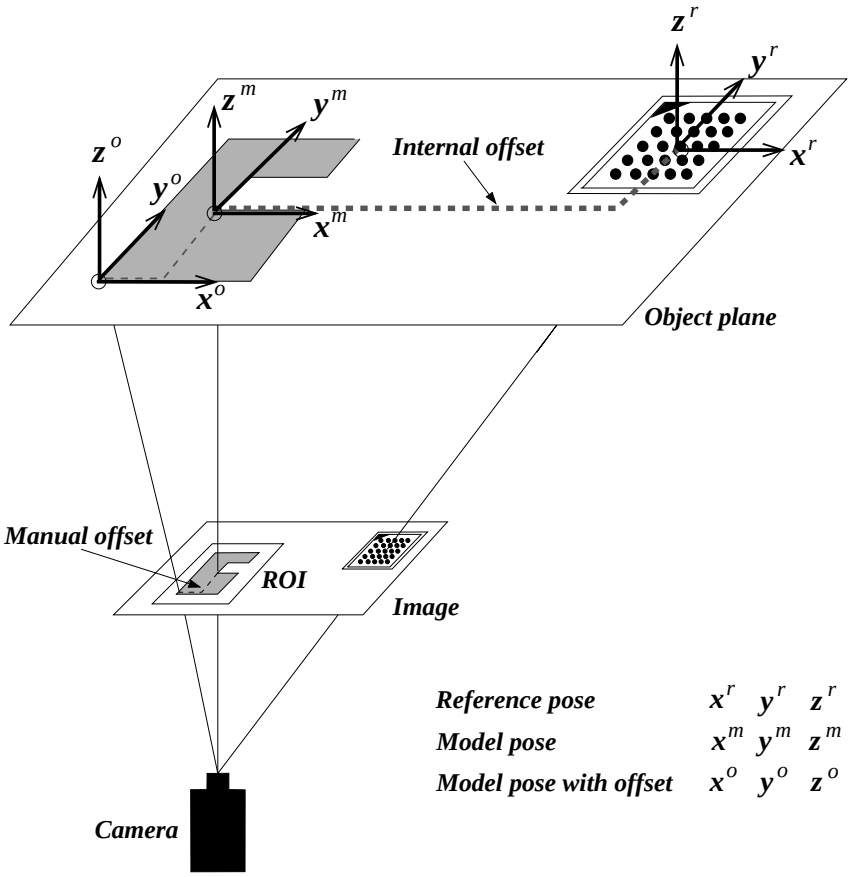


Figure 3.41: Internal and manual offset for the calibrated perspective matching: the offset from the reference pose to the model pose is calculated automatically from the reference pose, the internal camera parameters, and the template image. An additional offset can be applied manually and is defined in image coordinates.

In the following, we show how to select suitable parameters for these operators to adapt and optimize a matching task. In particular, we show how to

- restrict the search space to a region of interest (section 3.5.4.1),
- restrict the search space by restricting the range of orientation and scale via the parameters `AngleStart`, `AngleExtent`, `ScaleMin`, and `ScaleMax` (section 3.5.4.2),
- restrict the search space to a specific amount of allowed occlusions for the object, i.e., specify the visibility of the object via the parameter `MinScore` (section 3.5.4.3),
- specify the used search heuristics, i.e., trade thoroughness versus speed by adjusting the parameter `Greediness` (section 3.5.4.4),
- search for multiple instances of the model by adjusting the parameters `NumMatches` and `MaxOverlap` (section 3.5.4.5),
- restrict the number of pyramid levels (`NumLevels`) for the search process (section 3.5.4.6), and
- adjust some additional (generic) parameters within `GenParamName` and `GenParamValue` (section 3.5.4.7 on page 107).

3.5.4.1 Restrict the Search to a Region of Interest

The obvious way to restrict the search space and thus speed up the matching is to apply the operator `find_planar_uncalib_deformable_model` or `find_planar_calib_deformable_model` not to the whole image but only to an ROI. The corresponding procedure is explained in more detail for shape-based matching in

section 3.2.4.1 on page 63. For perspective deformable matching you simply have to replace `find_shape_model` by `find_planar_uncalib_deformable_model` or `find_planar_calib_deformable_model`, respectively.

3.5.4.2 Restrict the Range of Orientation and Scale (`AngleStart`, `AngleExtent`, `Scale*Min`, `Scale*Max`)

When calling `find_planar_uncalib_deformable_model` or `find_planar_calib_deformable_model` you can limit the orientation and scale 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.

3.5.4.3 Specify the Visibility of the Object (`MinScore`)

With the parameter `MinScore` you can specify how much of the model must be visible. A typical use of this mechanism is to allow a certain degree of occlusion. The parameter is similar to the corresponding parameter for shape-based matching that is described in more detail in section 3.2.4.3 on page 64.

3.5.4.4 Trade Thoroughness vs. Speed (`Greediness`)

With the parameter `Greediness` you can influence the search algorithm itself and thereby trade thoroughness against speed. The parameter is similar to the corresponding parameter for shape-based matching that is described in more detail in section 3.2.4.5 on page 65.

3.5.4.5 Search for Multiple Instances of the Object (`NumMatches`, `MaxOverlap`)

All you have to do to search for more than one instance of the object is to set the parameter `NumMatches` to the maximum number of instances you want to find. If you select the value `0`, all matches are returned.

Locating more than one instance, the operators `find_planar_uncalib_deformable_model` and `find_planar_calib_deformable_model` return the results for the individual model instances concatenated in the output tuples. That is, the `Score`, which is a single value for a single model instance, is now returned as a tuple for which the number of elements corresponds to the number of found model instances. For the results that are already returned as tuples for a single model instance, the number of elements multiplies by the number of found instances. These results comprise the projective transformation matrix (`HomMat2D`) for the uncalibrated matching or the 3D pose (`Pose`) and the standard deviations or covariances (`CovPose`) for the calibrated matching. How to access the values for a specific match is shown in section 3.5.5.1. Note that a search for multiple objects is only slighly slower than a search for a single object.

A second parameter, `MaxOverlap`, lets you specify how much two matches may overlap (as a fraction). This parameter is similar to the corresponding parameter of shape-based matching that is described in section 3.2.4.6 on page 66.

3.5.4.6 Restrict the Number of Pyramid Levels (`NumLevels`)

The parameter `NumLevels`, which you already specified when creating the model, allows you to use a different (in most cases a more restrictive) value in the search process. By using the value `0` for `NumLevels`, the value specified when creating the model is used.

Optionally, `NumLevels` can contain a second value, so that you can specify not only the highest but also the lowest pyramid level used for the search. If the search is aborted on a pyramid level that is higher than the first pyramid level, which corresponds to the original, full-sized image, the search becomes faster. On the other hand, the search is then also less robust and less accurate. If objects should be found also in images of poor quality, e.g., if the object is defocused or noisy, you can activate the increased tolerance mode by specifying the second value negatively. Then, the matches on the lowest pyramid level that still provides matches are returned.

3.5.4.7 Adjust Generic Parameters (GenParamName, GenParamValue)

Besides the parameters that correspond more or less to those used for shape-based matching, perspective deformable matching allows to adjust some generic parameters that are set via GenParamName and GenParamValue. Typically, there is not need to adjust them. But if, e.g., a low accuracy of the matching is sufficient, you can speed up the matching by a reduction or deactivation of the subpixel precision or by changing the discretization steps for the orientation or scale that were set during the creation of the model. To avoid false positive matches, you can also restrict the distortions of the angles and scales. For further information, please refer to the description of find_planar_uncalib_deformable_model in the Reference Manual.

3.5.5 Use the Specific Results of Perspective Deformable Matching

The perspective deformable matching returns for the uncalibrated case a 2D projective transformation matrix (HomMat2D), for the calibrated case a 3D pose (Pose) and either the standard deviations or the covariances (CovPose), and for both cases a score (Score) that evaluates the quality of the returned object location. The 2D projective transformation matrix and the 3D pose can be used, e.g., to transform a structure of the reference image into the search image as described in section 2.4.5 on page 42 for the 2D projective transformation matrix and in section 2.4.6 on page 45 for the 3D pose. Similar to the shape-based matching, perspective deformable matching can return multiple instances of a model. How to use these multiple instances is shown in section 3.5.5.1.

3.5.5.1 Deal with Multiple Matches

If multiple instances of the object are searched and found, the results are concatenated in tuples. In particular, Score, which is a single value for a single model instance, is now a tuple for which the number of elements corresponds to the number of found model instances. The access of the value for a specific instance is similar to the proceeding described in section 3.2.4.6 on page 66 for shape-based matching.

The parameters HomMat2D, Pose, and CovPose are already returned as tuples for a single model instance. For multiple instances, the values are simply concatenated. To access them for a specific match, you must know the number of elements returned for a single model instance, which is nine for a single projective transformation matrix, seven for a single 3D pose, and either six ('default') or 36 elements (generic parameter 'cov_pose_mode' set to 'covariances') for the single instance's standard deviations or covariances, respectively. Then, e.g., in the tuple containing the 3D poses that are returned by a calibrated matching, the first seven values belong to the first instance, the next seven values belong to the second instance, etc. An example for the access of the values for each instance is the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\locate_engine_parts.hdev, which locates the engine parts shown in figure 3.42. In the example, the 3D poses of the individual model instances are accessed by selecting specific subsets of the tuples using tuple_select_range.

```
find_planar_calib_deformable_model (Image, ModelID, rad(0), rad(360), 1, \
                                    1, 1, 1, 0.65, 0, 0, 3, 0.75, [], \
                                    [], Pose, CovPose, Score)
for Index1 := 0 to |Score| - 1 by 1
    tuple_select_range (Pose, Index1 * 7, ((Index1 + 1) * 7) - 1, \
                        PoseSelected)
    pose_to_hom_mat3d (PoseSelected, HomMat3D)
endfor
```

3.6 Descriptor-Based Matching

Similar to the perspective deformable matching, the descriptor-based matching is able to find objects even if they are perspectively deformed. Again, the matching can be applied either for a calibrated camera or for an uncalibrated camera. In contrast to the perspective deformable matching, the template is not built by the shapes of contours but by a set of so-called interest points. These points are first extracted by a detector and then are described, i.e., classified according to their location and their local gray value neighborhood, by a descriptor.

The following sections show

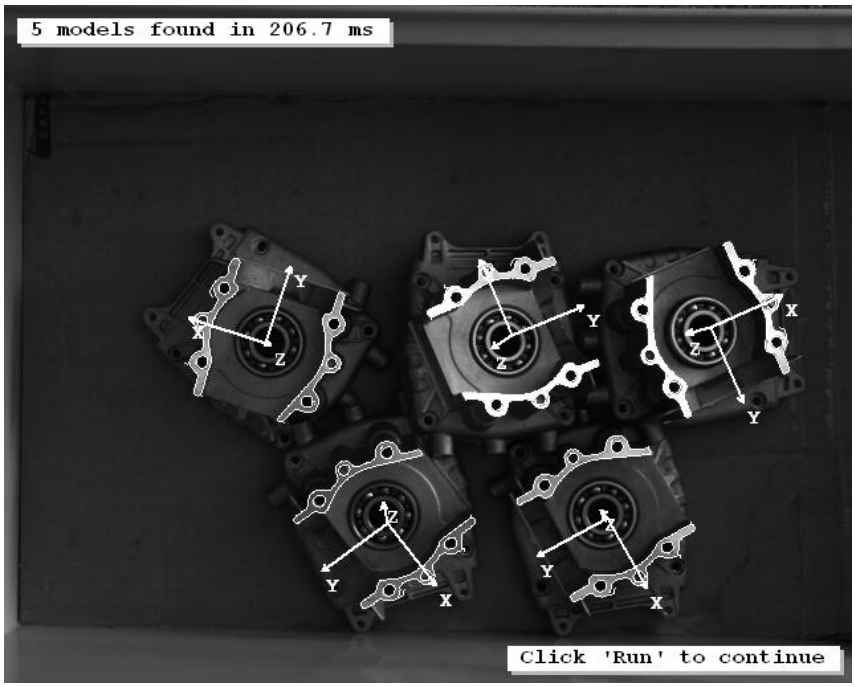


Figure 3.42: 3D poses of engine parts obtained by a calibrated perspective deformable matching.

- a first example for a descriptor-based matching (section 3.6.1),
- how to select an appropriate ROI to derive the template image from the reference image (section 3.6.2 on page 110),
- how to create a suitable model (section 3.6.3 on page 110),
- how to optimize the search (section 3.6.4 on page 112), and
- how to deal with the results that are specific for descriptor-based matching (section 3.6.5 on page 115).

3.6.1 A First Example

In this section we give a quick overview of the matching process with (calibrated) descriptor-based matching. To follow the example actively, start the HDevelop program `%HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\locate_cookie_box.hdev`, which locates a cookie box label that is aligned in different directions.

Step 1: Select the object in the reference image

First, the reference image is read and the image is reduced to a rectangular ROI. That is, a template image is derived that contains only the label of one specific side of a cookie box.

```
read_image (Image, 'packaging/cookie_box_01')
gen_rectangle1 (Rectangle, 224, 115, 406, 540)
reduce_domain (Image, Rectangle, ImageReduced)
```

As a calibrated matching is performed, the camera parameters and the reference pose of the label relative to the camera are needed. For a precise matching, these should be obtained, e.g., by a camera calibration as described in the Solution Guide III-C, section 3.2 on page 59. In the example, the pose is obtained by corresponding points, in particular by the image coordinates and the estimated world coordinates of the corner points of the rectangle from which the ROI was created.

Step 2: Create the model

The camera parameters and the reference pose are input to `create_calib_descriptor_model`, which is applied to create the calibrated descriptor model. Within the operator, the detector is selected and parameters for the

detector as well as for the descriptor are specified. In this case, the parameters for the detector are specified with [], i.e., the default values are selected. To be able to reuse the created descriptor model, it is stored to file with write_descriptor_file.

```
create_calib_descriptor_model (ImageReduced, CamParam, Pose, \
                               'harris_binomial', [], [], ['depth', \
                              'number_ferns','patch_size','min_scale', \
                              'max_scale'], [11,30,17,0.4,1.2], 42, \
                               ModelID)
write_descriptor_model (ModelID, 'cookie_box_model.dsm')
```

Step 3: Find the object again

find_calib_descriptor_model now locates the cookie box label in the search images. The returned 3D poses describe the relation between the world coordinates of the model and the world coordinates of the found matches. Figure 3.43 shows a search image with the visualized result.



Figure 3.43: 3D pose of cookie box label obtained by calibrated descriptor-based matching.

```
for Index := 1 to 10 by 1
    read_image (Image, 'packaging/cookie_box_' + Index$'.02')
    find_calib_descriptor_model (Image, ModelID, [], [], [], [], 0.25, 1, \
                                 CamParam, 'num_points', Pose, Score)
endfor
```

Step 4: Visualize the match

The result is visualized by different means. First, the interest points of the found model instance are queried with get_descriptor_model_points and can be immediately displayed as crosses using gen_cross_contour_xld.

```
        get_descriptor_model_points (ModelID, 'search', 0, Row, Col)
        gen_cross_contour_xld (Cross1, Row, Col, 6, 0.785398)
```

Then, the 3D coordinate system of the 3D pose is visualized by the procedure disp_3d_coord_system.

```
        disp_3d_coord_system (WindowHandle, CamParam, Pose, 0.07)
```

Finally, the rectangle that presents the outline of the cookie box label is displayed. For that, the rectangle must pass through different transformation steps. In particular, a transformation from the reference image to the world coordinate system (WCS) is followed by a 3D affine transformation within the WCS, which is followed by a transformation from the WCS to the search image.

As regions in HALCON cannot be transformed by a 3D transformation, the corner points of the rectangle must be transformed instead. Here, the world coordinates of the rectangle's corner points are obtained with image_points_to_world_plane.

```
image_points_to_world_plane (CamParam, Pose, RowsRoi, ColumnsRoi, 'm', \
                             XOuterBox, YOuterBox)
```

To apply the following 3D affine transformation with affine_trans_point_3d, the 3D pose that was returned by the matching must be converted into a 3D homogeneous transformation matrix using pose_to_hom_mat3d.

```
pose_to_hom_mat3d (Pose, HomMat3D)
affine_trans_point_3d (HomMat3D, XOuterBox, YOuterBox, [0,0,0,0], \
                       XTrans, YTrans, ZTrans)
```

The 3D coordinates obtained by the 3D affine transformation are then projected with project_3d_point into the search image, so that the rectangle can be reconstructed and displayed.

```
project_3d_point (XTrans, YTrans, ZTrans, CamParam, RowTrans, \
                  ColTrans)
gen_contour_polygon_xld (Contour, RowTrans, ColTrans)
close_contours_xld (Contour, Contour)
dev_display (Contour)
```

The following sections go deeper into the details of the individual steps of a descriptor-based matching and the parameters that have to be adjusted.

3.6.2 Select the Model ROI

As a first step of the descriptor-based matching, the region of interest that specifies the template image must be selected as described, e.g., in section 2.1.1 on page 17. This region can have an arbitrary shape, i.e., it can also contain holes or consist of several parts that are not connected. Thus, it is possible to create an ROI that contains only "robust" parts of the object. The ROI must be selected so that potentially important interest points are not directly at the border of the region, as the immediate surroundings (neighborhood) of the interest points are needed to obtain the model. Additionally, the part of the object that is visible in the region of interest needs to be planar. After having selected a suitable ROI, the reduced image is used as template image for the creation of the model.

3.6.3 Create a Suitable Descriptor Model

Having derived the template image from the reference image, the descriptor model can be created using

- create_uncalib_descriptor_model for an uncalibrated descriptor-based matching or
- create_calib_descriptor_model for a calibrated descriptor-based matching.

Except for the camera parameters and the reference pose, which are needed only for the calibrated case, the parameters that have to be adjusted are the same for both operators. Here, we will take a closer look at how to adjust them. In particular, we show how to

- select and adjust the detector (section 3.6.3.1),
- adjust the descriptor (section 3.6.3.2), and
- specify the camera parameters and the reference pose that are needed for the calibrated matching (section 3.6.3.3 on page 112).

Note that after the creation of the model, the model can still be modified. In section 3.6.3.4 on page 112 the possibilities for the inspection and modification of an already created model are shown.

3.6.3.1 Select and Adjust the Detector (DetectorType, DetectorParamName, DetectorParamValue)

The interest points that build the model are extracted from the image by the so-called detector. The type of detector is selected via the parameter DetectorType. Available types are 'lepetit', 'harris', and 'harris_binomial', which correspond to the HALCON point operators points_lepetit, points_harris, and points_harris_binomial. 'lepetit' can be used for a very fast extraction of significant points, but the obtained points are not as robust as those obtained with 'harris'. Especially, if a template or search image is very dark or has got a low contrast, 'lepetit' is not recommendend. 'harris_binomial' is a good compromise between 'lepetit' and 'harris', because it is faster than 'harris' and more robust than 'lepetit'.

For each detector type a set of generic parameters is available that can be adjusted with the parameters DetectorParamName and DetectorParamValue. The first one is used to specify the names of the generic parameters and the second one is used to specify the corresponding values. We recommend to apply tests with the selected point operator before creating the model. That is, you apply the corresponding point operator to the template image and visualize the returned points using, e.g., gen_cross_contour_xld. For a good result, about 50 to 450 points should be uniformly distributed within the template image. If you have found the appropriate parameter setting for the selected point operator, you can set these parameters also for the model in create_calib_descriptor_model or create_uncalib_descriptor_model, respectively. Note that in most cases the default values of the detectors (DetectorParamName and DetectorParamValue set to []) are sufficient.

3.6.3.2 Adjust the Descriptor (DescriptorParamName, DescriptorParamValue)

The currently implemented descriptor uses randomized ferns to classify the extracted points, i.e., to build characteristic descriptions of the location and the local gray value neighborhood for the interest points.

The descriptor can be adjusted with the parameters DescriptorParamName and DescriptorParamValue. The first one is used to specify the names of the generic parameters that have to be adjusted and the second one is used to specify the corresponding values.

The parameters can be divided into parameters that control the size of the descriptor and thus allow to control the detection robustness, speed, and memory consumption, and in parameters that control the simulation, especially the spatial range in which the model views are trained. The **size of the descriptor** is controlled by the following parameters:

- 'depth' specifies the depth of the classification fern. Interest points can better be discriminated when selecting a higher depth. On the other hand, a higher depth leads to an increasing runtime.
- 'number_ferns' specifies the number of used fern structures. Using many fern structures leads to a better robustness but also to an increasing runtime.
- 'patch_size' specifies the side length of the quadratic neighborhood that is used to describe the individual interest point. Again, a too large value can disadvantageously influence the runtime.

The selection of values for the depth and the number of ferns depends on your specific requirements. If a **fast online matching** is required, few ferns with a large depth are recommendend. If a **robust matching result** is needed, many ferns are needed and a large depth may additionally increase the robustness, although it might also significantly increase the runtime of the matching. If the **memory consumption** is critical, many ferns with a small depth are recommended. For many applications, a trade-off between the different requirements will be needed.

The **simulation, i.e., the training of the model** is controlled by the following parameters:

- 'tilt' is used to switch 'on' or 'off' the projective transformations during the simulation phase, which leads either to an enhanced robustness of the model or to a speed-up of the training.
- 'min_rot' and 'max_rot' define the range for the angle of rotation around the normal vector of the model.
- 'min_scale' and 'max_scale' define the scale range of the model.

The restriction to small ranges for the angle of rotation and the scale can be used to **speed up the training** significantly. But note that during the later applied matching the model can be found only if its angle and scale is within the trained scope. Note further that the training is faster for small images. Thus, you can speed up the

training by selecting a small reference image and small template images and setting 'tilt' to 'off'. Note also that the reference image and the search image should have the same size.

An example for the restriction of the orientation and scale of the model is given in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\detect_brochure_pages.hdev that creates an uncalibrated descriptor model for the matching of different brochure pages. There, the orientation is restricted to an angle range of +/- 90° ('default' is +/- 180°) and the scale range is changed to a scaling factor between 0.2 and 1.1 ('default' is between 0.5 and 1.4).

```
create_uncalib_descriptor_model (ImageReduced, 'harris_binomial', [], \
                                 [], ['min_rot','max_rot','min_scale', \
                                'max_scale'], [-90,90,0.2,1.1], 42, \
                                 ModelID)
```

3.6.3.3 Specify the Camera Parameters and the Reference Pose (CamParam, ReferencePose)

For a calibrated matching, additionally the camera parameters (CamParam) and a reference pose (ReferencePose) have to be specified. We recommend to obtain both using a camera calibration as described in the Solution Guide III-C, section 3.2 on page 59. Other approaches for determining the pose of the object plane comprise a manual measurement of the extents of the model, which is rather intricate and often inaccurate, stereo vision (see Solution Guide III-C, chapter 5 on page 115), or 3D laser triangulation, e.g., using sheet of light (see Solution Guide III-C, chapter 6 on page 145).

3.6.3.4 Inspect and Modify the Descriptor Model

If you want to visually inspect an already created model, you can get the coordinates of the interest points that are contained in it using get_descriptor_model_points with Set set to 'model'.

```
get_descriptor_model_points (ModelID, 'model', 'all', Row_D, Col_D)
```

To inspect the current parameter values of a model, you query them with get_descriptor_model_params. This may be necessary if during the creation of the model an automatic parameter selection was used or if the model was created within another program, saved to file with write_descriptor_model, and read from this file in the current program with read_descriptor_model. Additionally, you can query the coordinates of the origin of the model using get_descriptor_model_origin.

After the creation of the model and before you search the object in a search image, you can further modify the model. In particular, you can apply set_descriptor_model_origin to change its origin. But note that this is not recommended because the accuracy of the matching result may decrease, which is shown in more detail for shape-based matching in section 3.2.4.8 on page 68. For the case that you nevertheless need to modify the point of reference and you want to apply a calibrated matching, we refer to the corresponding description for perspective deformable matching in section 3.5.3.7 on page 104. There, the relation between the reference pose, the model pose, and an offset that is manually applied to the point of reference is introduced.

Note that **after the matching**, you can use get_descriptor_model_points also to query the interest points of a specific match. Then, Set must be set to 'search' instead of 'model'. Here, the interest points of the first found model instance (index 0) are queried.

```
get_descriptor_model_points (ModelIDs[Index2], 'search', 0, Row, Col)
```

Additionally, after the matching, get_descriptor_model_results can be used to query selected numerical results that were accumulated during the search like the scores for the correspondences between the individual search points and model points (ResultNames set to 'point_classification').

3.6.4 Optimize the Search Process

To locate the same interest points that are stored and described in the model in unknown images of the same or a similar object, the following operators are applied:

- `find_calib_descriptor_model` searches for the best matches of a calibrated descriptor model. It returns the 3D pose of the object and the score describing the quality of the match.
- `find_uncalib_descriptor_model` searches for the best matches of an uncalibrated descriptor model. It returns a 2D projective transformation matrix (homography) and the score describing the quality of the match.

Except for the camera parameters (`CamParam`), the uncalibrated and the calibrated case need the same parameters to be adjusted. Note that the camera parameters, assuming the same setup for the creation of the model and the search, remain the same that were already specified (see section 3.6.3 on page 110). If a different camera is used for the search, we recommend to apply a new camera calibration as described in the Solution Guide III-C, section 3.2 on page 59. In the following, we show how to

- restrict the search space to a region of interest (section 3.6.4.1),
- adjust the detector for the search, which is recommended only in very rare cases (section 3.6.4.2),
- adjust the descriptor for the search (section 3.6.4.3),
- specify the similarity of the object by adjusting the parameter `MinScore` (section 3.6.4.4),
- search for multiple instances of the object by adjusting the parameter `NumMatches` (section 3.6.4.5), and
- select a score type by adjusting the parameter `ScoreType` (section 3.6.4.6).

3.6.4.1 Restrict the Search to a Region of Interest

The obvious way to restrict the search space and thus speed up the matching is to apply the operator `find_uncalib_descriptor_model` or `find_calib_descriptor_model` not to the whole image but only to an ROI. The corresponding procedure is explained in more detail for shape-based matching in section 3.2.4.1 on page 63. For descriptor-based matching you simply have to replace `find_shape_model` by `find_uncalib_descriptor_model` or `find_calib_descriptor_model`, respectively.

3.6.4.2 Adjust the Detector for the Search (`DetectorParamName`, `DetectorParamValue`)

The parameters that control the detection, i.e., the extraction of the interest points from the image, are adjusted via `DetectorParamName` and `DetectorParamValue`. They correspond to the detector parameters that were already specified during the creation of the model (see section 3.6.3 on page 110). In most cases, they should not be changed for the search. That is, you simply pass an empty tuple ([]) to the parameters.

In rare cases, especially when there are significant illumination changes between the reference image and the search image, you may change the parameter values. For example, if a search image is extremely dark and `'lepetit'` was selected as detector, you can set `'min_score'` to a smaller value.

Generally, to test if it is necessary to change any of the parameter values, you can apply the point operator that corresponds to the detector not only to the reference image, which was proposed for the creation of the model (see also section 3.6.3.1 on page 111), but also to the search image. Again, about 50 to 450 uniformly distributed points should be extracted to get a good matching result. If the point operator needs different parameters for the reference and the search image, it might be necessary to change the values of the corresponding detector parameters accordingly.

3.6.4.3 Adjust the Descriptor for the Search (`DescriptorParamName`, `DescriptorParamValue`)

The parameters that control the determination of the correspondences between the interest points of the search image and those of the model are adjusted via `DescriptorParamName` and `DescriptorParamValue`. Two generic parameters can be set:

- The parameter 'min_score_descr' can be set to a value larger than 0.0 (but preferably below 0.1) to increase the minimal classification score that determines if the individual points are considered as potential matches. Thus, the number of points for further calculations is reduced, so that the speed of the matching can be enhanced. But note that this speed-up is obtained at the cost of a reduced robustness, especially if the number of extracted points is low.
- The parameter 'guided_matching' can be switched off by setting it from 'on' to 'off'. If the guided matching is switched on, which is the default, the robustness of the estimation of the model's location is enhanced. In particular, points are extracted from the search image and classified by the descriptor. The points that were accepted by the classification are used to calculate an initial projective 2D (uncalibrated case) or homogeneous 3D (calibrated case) transformation matrix. This is used then to project all model points into the search image. If a projected model point is near to one of the originally extracted points, i.e., independently from its classification, this point is used for the final calculation of the homography that is returned by the matching as 2D projective transformation matrix or 3D pose, respectively. As typically without the classification more points can be used for the calculations, the obtained homography is more robust. On the other hand, in some cases the runtime of the matching may increase up to 10%. Thus, if robustness is less important than speed, 'guided_matching' can be switched off.

The HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\detect_brochure_pages.hdev is an example for setting 'min_score_descr' to a higher value (0.003) to speed up the matching. As the number of points in the images is large enough, the matching is still sufficiently robust.

```
find_uncalib_descriptor_model (ImageGray, ModelIDs[Index2], \
                               'threshold', 600, \
                               ['min_score_descr', \
                              'guided_matching'], [0.003,'on'], \
                               0.25, 1, 'num_points', HomMat2D, \
                               Score)
```

3.6.4.4 Specify the Similarity of the Object (MinScore)

The parameter MinScore specifies the minimum score a potential match must have to be returned as match. The score is a value for the quality of a match, i.e., for the correspondence, or "similarity", between the model and the search image. Note that for the descriptor-based matching, different types of score are available for the output parameter Score (see section 3.6.4.6). But for the input parameter MinScore, always the score of type 'inlier_ratio' is used. It calculates the ratio of the number of point correspondences to the number of model points. In most cases, MinScore should be set to a value of at least 0.1. To speed up the search, it should be chosen as large as possible, but of course still as small as necessary for the success of the search, as, e.g., a value of 1.0 is rather unlikely to be reached by a matching.

3.6.4.5 Search for Multiple Instances of the Object (NumMatches)

All you have to do to search for more than one instance of the object is to set the parameter NumMatches to the maximum number of instances you want to find. If you select the value 0, all matches are returned.

Locating more than one instance, the operators find_uncalib_descriptor_model and find_calib_descriptor_model return the results for the individual model instances concatenated in the output tuples. That is, the Score, which is typically a single value for a single model instance (see section 3.6.4.6 for exceptions), is now returned as a tuple for which the number of elements corresponds to the number of found model instances. The number of elements for the projective transformation matrix (HomMat2D) or the 3D pose (Pose), which are tuples already for a single model instance, is the number of elements of the single instance multiplied by the number of found instances. How to access the values for a specific match is shown in section 3.6.5.1. Note that a search for multiple objects is only slightly slower than a search for a single object.

3.6.4.6 Select a Suitable Score Type (ScoreType)

The parameter ScoreType is used to select the type of score that will be returned in the parameter Score. Available types are 'num_points' and 'inlier_ratio':

- For 'num_points', the number of point correspondences per instance is returned. As any four correspondences define a mathematically correct homography between two images, this number should be at least 10 to assume a reliable matching result.
- For 'inlier_ratio' the ratio of the number of point correspondences to the number of model points is returned. Although this parameter may have a value between 0.0 and 1.0, a ratio of 1.0 is rather unlikely to be reached by a matching. Yet, objects having an inlier ratio of less than 0.1 should be disregarded.

Typically, one of both score types is selected, but it is also possible to pass both types in a tuple. Then, the result for a single found model instance is returned in a tuple as well.

3.6.5 Use the Specific Results of Descriptor-Based Matching

The descriptor-based matching returns for the uncalibrated case a 2D projective transformation matrix (HomMat2D), for the calibrated case a 3D pose (Pose), and for both cases a score (Score) that evaluates the quality of the returned object location. The 2D projective transformation matrix and the 3D pose can be used, e.g., to transform a structure of the reference image into the search image as described in section 2.4.5 on page 42 for the 2D projective transformation matrix and in section 2.4.6 on page 45 for the 3D pose. The score can be interpreted as described in section 3.6.4.6 on page 114. Similar to the shape-based matching, descriptor-based matching can return multiple instances of a model. How to use these multiple instances is shown in section 3.6.5.1.

3.6.5.1 Deal with Multiple Matches

If multiple instances of the object are searched and found, the results are concatenated in tuples. In particular, Score, which is typically a single value for a single model instance (see section 3.6.4.6 on page 114 for exceptions) is now a tuple for which the number of elements corresponds to the number of found model instances. The access of the value for a specific instance is similar to the proceeding described in section 3.2.4.6 on page 66 for shape-based matching.

The parameters HomMat2D or Pose are already returned as tuples for a single model instance. For multiple instances, the values are simply concatenated. To access them for a specific match, you must know the number of elements returned for a single model instance, which is nine for a single projective transformation matrix and seven for a single 3D pose. Then, e.g., in the tuple returned for the 3D poses, the first seven values belong to the first instance, the next seven values belong to the second instance, etc. An example for the access of the values for each instance is described in more detail for calibrated perspective deformable matching in section 3.5.4.5 on page 106.

Index

2D affine transformation, 29
2D projective transformation, 31
2D rigid transformation from points and angle, 30
2D transformation, 29
3D transformation, 32

access results of multiple matching model instances (descriptor-based), 115
access results of multiple matching models (local deformable), 98
access results of multiple matching models (perspective deformable), 107
align image using shape-based matching, 38
align regions of interest using shape-based matching, 34
allow orientation range for matching (correlation-based), 49
allow orientation range for matching (local deformable), 94
allow orientation range for matching (perspective deformable), 103
allow orientation range for matching (shape-based), 58

component-based matching
  first example, 74
  overview, 74
correlation-based matching (NCC)
  first example, 47
  overview, 47
create (train) matching model (component-based), 78
create (train) matching model (correlation-based), 49
create (train) matching model (descriptor-based), 110
create (train) matching model (local deformable), 93
create (train) matching model (perspective deformable), 101
create (train) matching model (shape-based), 55
create matching model from DXF file, 23
create matching model from synthetic template, 20
create matching model from XLD contours, 22
create template, 17

descriptor-based matching
  first example, 108
  overview, 107
determine training parameters for matching (correlation-based), 49
determine training parameters for matching (local deformable), 93
determine training parameters for matching (perspective deformable), 101
determine training parameters for matching (shape-based), 55
display results of matching, 33

find matching model (component-based), 85
find matching model (correlation-based), 50
find matching model (descriptor-based), 112
find matching model (local deformable), 95
find matching model (perspective deformable), 104
find matching model (shape-based), 61
find matching model with perspective distortion, 74
find model for matching, 17
find multiple matching models (correlation-based), 51
find multiple matching models (shape-based), 66
find multiple model instances (component-based), 86
find multiple model instances (correlation-based), 51
find multiple model instances (descriptor-based), 114
find multiple model instances (local deformable), 96
find multiple model instances (perspective deformable), 106
find multiple model instances (shape-based), 66

get found matching model components, 74, 88
get initial components of matching model, 76
get matching model contours (local deformable), 95
get matching model origin (correlation-based), 50
get matching model origin (descriptor-based), 112
get matching model origin (perspective deformable), 104
get matching model origin (shape-based), 61
get matching model parameters (component-based), 85
get matching model parameters (descriptor-based), 112
get matching model parameters (local deformable), 95
get matching model parameters (perspective deformable), 104
get multiple model results of matching (shape-based), 72
get multiple results of matching (shape-based), 72

inspect matching model (component-based), 85
inspect matching model (correlation-based), 50
inspect matching model (descriptor-based), 112
inspect matching model (local deformable), 95

inspect matching model (perspective deformable), 104
inspect matching model (shape-based), 61

local deformable matching
  first example, 90
  overview, 89

matching approaches, 47
matching model contours (perspective deformable), 104
matching model contours (shape-based), 61
matching model parameters (correlation-based), 50
matching model parameters (shape-based), 61
matching model points (descriptor-based), 112
model creation (training), 17

perspective deformable matching, 99

re-use matching model (component-based), 84
re-use matching model (correlation-based), 50
re-use matching model (descriptor-based), 112
re-use matching model (local deformable), 95
re-use matching model (perspective deformable), 104
re-use matching model (shape-based), 61
re-use model, 24
read matching model (component-based), 84
read matching model (correlation-based), 50
read matching model (descriptor-based), 112
read matching model (local deformable), 95
read matching model (perspective deformable), 104
read matching model (shape-based), 61
rectify image for matching (shape-based) to adapt to new camera orientation, 74
reference point for matching, 18
region of interest from matching model, 19
region of interest from matching model (shape-based), 54
restrict orientation range for matching (component-based), 86
restrict orientation range for matching (correlation-based), 50
restrict orientation range for matching (local deformable), 96
restrict orientation range for matching (perspective deformable), 106
restrict orientation range for matching (shape-based), 64
results of matching, 28
root component (component-based), 86
rotate 2D homogeneous matrix, 30

scale 2D homogeneous matrix, 30
score, 46
select score type for matching (descriptor-based), 114
select suitable matching approach, 8
set camera parameters for matching (descriptor-based), 112
set camera parameters for matching (perspective deformable), 104
set clutter parameters for matching (shape-based), 65
set generic parameters (local deformable), 94
set generic parameters (perspective deformable), 103
set generic parameters for speedup (local deformable), 96
set generic parameters for speedup (perspective deformable), 107
set matching model metric (correlation-based), 49
set matching model metric (local deformable), 94
set matching model metric (perspective deformable), 103
set matching model metric (shape-based), 60
set matching model origin (correlation-based), 50
set matching model origin (descriptor-based), 112
set matching model origin (local deformable), 95
set matching model origin (shape-based), 61
set matching model parameters (correlation-based), 52
set matching model parameters (shape-based), 69
set number of pyramid levels for matching (correlation-based), 51
set number of pyramid levels for matching (local deformable), 96
set number of pyramid levels for matching (perspective deformable), 106
set number of pyramid levels for matching (shape-based), 69
set threshold to extract matching model (local deformable), 94
set threshold to extract matching model (perspective deformable), 102
set threshold to extract matching model (shape-based), 57
set training model origin (perspective deformable), 104
shape-based matching
  first example, 52
  overview, 52
specify accuracy for matching (correlation-based), 51
specify accuracy for matching (shape-based), 68
specify iconic objects for matching (local deformable), 96
specify object similarity for matching (correlation-based), 50
specify object similarity for matching (descriptor-based), 114
specify object visibility for matching, 64
specify object visibility for matching (component-based), 86
specify object visibility for matching (local deformable), 96
specify object visibility for matching (perspective deformable), 106
speed up matching, 25
speed up matching (local deformable), 96
speed up matching (perspective deformable), 106

speed up matching (shape-based), 65, 70
speed up matching with subsampling, 25
speed up matching with subsampling (correlation-based), 49
speed up matching with subsampling (local deformable), 94
speed up matching with subsampling (perspective deformable), 103
speed up matching with subsampling (shape-based), 58

timeout for matching (correlation-based), 52
timeout for matching (shape-based), 69
translate 2D homogeneous matrix, 30

use 2D pose (position, orientation) result of matching, 32
use 3D pose result of template matching, 45
use deformed contours (local deformable), 98
use homography results of matching, 42
use rectified image (local deformable), 97
use region of interest, 25
use region of interest for matching (component-based), 85
use region of interest for matching (correlation-based), 48, 50
use region of interest for matching (descriptor-based), 110, 113
use region of interest for matching (local deformable), 92, 95
use region of interest for matching (perspective deformable), 101, 105
use region of interest for matching (shape-based), 63
use results of matching (component-based), 88
use results of matching (descriptor-based), 115
use results of matching (local deformable), 97
use results of matching (perspective deformable), 107
use results of matching (shape-based), 71
use scale result of matching, 41
use synthetic model for matching, 20
use vector field (local deformable), 97

write matching model (component-based), 84
write matching model (correlation-based), 50
write matching model (descriptor-based), 112
write matching model (local deformable), 95
write matching model (perspective deformable), 104
write matching model (shape-based), 61