HALCON
a product of MVTec

Solution Guide I
Basics

HALCON 23.11 *Progress*

MVTec Software GmbH

Building Vision for Business

How to use HALCON's machine vision methods, Version 23.11.0.0

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

Contents

Chapter 1

Guide to HALCON Methods

This manual introduces you to important machine vision methods. To guide you from your specific application to the sections of the documentation to read, this section lists common application areas and the methods used for them.

Generally, a lot of applications use the following methods:

- Image Acquisition on page 21 for accessing images via an image acquisition device or via file.
- Visualization on page 223 for the visualization of, e.g., artificially created images or results of an image processing task.
- Region of interest on page 25 for reducing the search space for a following image processing task.
- Morphology (Reference Manual, chapter "Morphology"), e.g., for the elimination of small gaps or protrusions from regions or from structures in gray value images.

Other methods are more specific and thus are suited for specific application areas. Additionally, some application areas are part of another application area. To make the relations more obvious, for the following application areas the corresponding methods and related application areas are listed:

- Color Inspection (page 14)
- Completeness Check (page 14)
- Identification (page 14)
- Measuring and Comparison 2D (page 14)
- Measuring and Comparison 3D (page 15)
- Object Recognition 2D (page 15)
- Object Recognition 3D (page 15)
- Position Recognition 2D (page 16)
- Position Recognition 3D (page 16)
- Print Inspection (page 16)
- Quality Inspection (page 17)
- Robot Vision (page 17)
- Security System (page 17)
- Surface Inspection (page 17)
- Texture Inspection (page 18)

To speed up some applications, compute devices can be used. When and how to use compute devices is explained in the chapter Compute Devices on page 233.

1.1 Color Inspection

For color inspection, see the descriptions for Color Processing on page 131.

1.2 Completeness Check

Completeness checks can be realized by different means. Common approaches are:

- Object and position recognition 2D/3D (see page 15 ff), which is suitable, e.g., when inspecting objects on an assembly line.
- Variation Model on page 111, which compares images containing similar objects and returns the difference between them considering a certain tolerance at the object's border.

1.3 Identification

Dependent on the symbols or objects you have to identify, the following methods are suitable:

- Identify symbols or characters
  - Bar Code on page 159
  - Data Code on page 175
  - OCR on page 183
- Identify general objects
  - Object and position recognition 2D/3D (see page 15 ff)

1.4 Measuring and Comparison 2D

For measuring 2D features in images, several approaches are available. In the Solution Guide III-B, section 3.1 on page 27, a graph leads you from the specific features you want to measure and the appearance of the objects in the image to the suitable measuring approach. Generally, the following approaches are common:

- Blob Analysis on page 33 for objects that consist of regions of similar gray value, color, or texture.
- Contour Processing on page 79 for objects that are represented by clear-cut edges. The contours can be obtained by different means:
  - Edge Extraction on page 55 if pixel precision is sufficient.
  - Edge and Line Extraction on page 63 if subpixel precision is needed.
- Matching on page 89 for objects that can be represented by a template. Matching comprises different approaches. For detailed information about matching see the Solution Guide II-B.
- 1D Measuring on page 45 if you want to obtain the positions, distances, or angles of edges that are measured along a line or an arc. More detailed information can be found in the Solution Guide III-A.

1.5 Measuring and Comparison 3D

For measuring in 3D, the following approaches are available:

- The approaches used for measuring and comparison 2D (see page 14) in combination with a camera calibration (see Solution Guide III-C, section 3.2 on page 61) for measuring objects that are viewed by a single camera and that lie in a single plane.
- Pose estimation (Solution Guide III-C, chapter 4 on page 91) for the estimation of the poses of 3D objects that are viewed by a single camera and for which knowledge about their 3D model (e.g., known points, known circular or rectangular shape) is available.

3D reconstruction is an important subcategory of 3D measuring and comprises the following methods:

- Stereo for measuring in images obtained by a binocular or multi-view stereo system on page 215. Further information can be found in the Solution Guide III-C, chapter 5 on page 117.
- Laser triangulation using the sheet-of-light technique (Solution Guide III-C, chapter 6 on page 147) for measuring height profiles of an object by triangulating the camera view with a projected light line.
- Depth from focus (Solution Guide III-C, chapter 7 on page 163) for getting depth information from a sequence of images of the same object but with different focus positions.
- Photometric Stereo (Reference Manual, chapter "3D Reconstruction ▷ Photometric Stereo") for getting information about an object's shape because of its shading behavior (e.g., by the operator `photometric_stereo`).

1.6 Object Recognition 2D

For finding specific objects in images, various methods are available. Common approaches comprise:

- Blob Analysis on page 33 for objects that are represented by regions of similar gray value, color, or texture.
- Contour Processing on page 79 for objects that are represented by clear-cut edges. The contours can be obtained by different means:
  - Edge Extraction on page 55 if pixel precision is sufficient.
  - Edge and Line Extraction on page 63 if subpixel precision is needed.
- Matching on page 89 for objects that can be represented by a template. Matching comprises different approaches. For detailed information about matching see the Solution Guide II-B.
- Classification on page 119 for the recognition of objects by a classification using, e.g., Gaussian mixture models, neural nets, or support vector machines. For more detailed information about classification see the Solution Guide II-D.
- Color Processing on page 131 for the recognition of objects that can be separated from the background by their color.
- Texture Analysis on page 143 for the recognition of objects that can be separated from the background by their specific texture.
- Movement detection (see section 1.13 on page 17) for the recognition of moving objects.

1.7 Object Recognition 3D

For the recognition of 3D objects that are described by a 3D Computer Aided Design (CAD) model, see the descriptions for 3D Matching on page 101. For the recognition of planar objects that can be oriented arbitrarily in the 3D space, see the descriptions for perspective, deformable matching and descriptor-based matching in the chapter about Matching on page 89 for the uncalibrated case and in the Solution Guide III-C, chapter 4 on page 91 for the calibrated case.

1.8 Position Recognition 2D

In parts, the approaches for measuring 2D features are suitable to get the position of objects. In the Solution Guide III-B, section 3.1 on page 27, a graph leads you from several specific features, amongst others the object position, and the appearance of the objects in the image to the suitable approach. For position recognition, in particular the following approaches are common:

- Blob Analysis on page 33 for objects that consist of regions of similar gray value, color, or texture.
- Contour Processing on page 79 for objects that are represented by clear-cut edges. The contours can be obtained by different means:
  - Edge Extraction on page 55 if pixel precision is sufficient.
  - Edge and Line Extraction on page 63 if subpixel precision is needed.
- Matching on page 89 for objects that can be represented by a template. Matching comprises different approaches. For detailed information about matching see the Solution Guide II-B.
- 1D Measuring on page 45 if you want to obtain the positions of edges that are measured along a line or an arc. More detailed information can be found in the Solution Guide III-A.

1.9 Position Recognition 3D

For the position recognition of 3D objects, the following approaches are available:

- The approaches used for measuring and comparison 2D (see page 14) in combination with a camera calibration (see Solution Guide III-C, section 3.2 on page 61) for measuring objects that are viewed by a single camera, and which lie in a single plane.
- Pose estimation (Solution Guide III-C, chapter 4 on page 91) for the estimation of the poses of 3D objects that are viewed by a single camera and for which knowledge about their 3D model (e.g., known points, known circular or rectangular shape) is available.
- Stereo for measuring positions in images obtained by a binocular stereo system on page 215 or a multi-view stereo system. Further information can be found in the Solution Guide III-C, chapter 5 on page 117.
- 3D matching on page 101 for objects that are searched for based on a 3D Computer Aided Design (CAD) model.

1.10 Print Inspection

For print inspection, the suitable method depends on the type of print you need to inspect:

- Optical character verification (Reference Manual, chapter "Inspection ▷ OCV") for the verification of characters.
- Data Code on page 175 for inspecting the quality of a printed 2D data code symbol. Further information can be found in the Solution Guide II-C, section 6 on page 45.
- Variation Model on page 111, which compares images containing similar objects and returns the difference between them considering a certain tolerance at the object's border.
- Component-based matching (part of Matching on page 89), if the print is built by several components which may vary in their relation to each other (orientation and distance) within a specified tolerance.

1.11 Quality Inspection

How to inspect the quality of an object depends on the features describing the quality. The following application areas and the methods used by them can be applied for quality inspection:

- Surface inspection (see page 17) if, e.g., scratches in an object's surface have to be detected.
- Completeness check (see page 14) if you need to check an object for missing parts.
- Measuring and comparison 2D/3D (see page 14 ff) if an object has to fulfill certain requirements related to its area, position, orientation, dimension, or number of parts.

Additionally, Classification on page 119 can be used to check the color or texture of objects. For more detailed information about classification see the Solution Guide II-D.

1.12 Robot Vision

For robot vision, you can combine the approaches for object and position recognition 2D/3D (see page 15 ff) with the hand-eye calibration described in the Solution Guide III-C, chapter 8 on page 175.

1.13 Security System

For a sequence analysis or movement detection you can use, e.g., one of the following approaches:

- Blob Analysis on page 33, e.g., by using the operator `dyn_threshold` to obtain the difference between two images and thus detect moving objects. The approach is fast, but detects objects only as long as they are moving.
- Background estimator (Reference Manual, chapter "Tools ▷ Background Estimator") for the recognition of moving objects even if they stop temporarily. It adapts to global changes concerning, e.g., illumination.
- Optical flow (Reference Manual, chapter "Filters ▷ Optical Flow") for the recognition of moving objects even if they stop temporarily. Because of complex calculations, it is slower than the other approaches, but additionally returns the velocity for each object.
- Scene flow (Reference Manual, chapter "Filters ▷ Scene Flow") for the recognition of moving objects in 3D even if they stop temporarily. In contrast to the optical flow, which returns the information about moving objects only in 2D, the scene flow calculates the position and motion in 3D. Because of complex calculations, it is even slower than the optical flow, but additionally returns 3D information.
- Kalman filter (Reference Manual, chapter "Tools ▷ Kalman Filter") can be applied after the recognition of moving objects to predict the future positions of objects.

For the recognition of, e.g., irises or faces, Classification on page 119 may be suitable. For more detailed information about classification see the Solution Guide II-D.

Examples solving different tasks relevant for security systems can be found via the Open Example dialog inside HDevelop for the category `Industry/Surveillance and Security`.

1.14 Surface Inspection

For surface inspection, several approaches are available:

- Via comparison with a reference image

  - Variation Model on page 111, which compares images containing similar objects and returns the difference between them. This is suited especially if you need to detect irregularities that are placed inside the object area, whereas small irregularities at the object's border can be tolerated.

- Via comparison with a reference pattern or color
  - Texture Analysis on page 143
  - Color Processing on page 131
  - Classification on page 119 (for more detailed information about classification see the Solution Guide II-D)

- Via defect description for uniformly structured surfaces
  - Blob Analysis on page 33 for objects that consist of regions of similar gray value, color, or texture.
  - Contour Processing on page 79 for objects that are represented by clear-cut edges. The contours can be obtained by an Edge Extraction on page 55 if pixel precision is sufficient, or by an Edge and Line Extraction on page 63 if subpixel precision is needed.

If a single image is not suited to cover the object to inspect, several approaches exist to combine images after their acquisition:

- Calibrated mosaicking (Solution Guide III-C, chapter 9 on page 191) for a high-precision mosaicking of a discrete number of overlapping images obtained by two ore more cameras.
- Uncalibrated mosaicking (Solution Guide III-C, chapter 10 on page 205) for a less precise mosaicking of a discrete number of overlapping images obtained as an image sequence.
- Combination of lines obtained by a line scan camera to so-called pages (Solution Guide II-A, section 6.6 on page 39) for inspecting continuous material on an assembly line.

1.15 Texture Inspection

Common approaches for texture inspection are:

- Fast Fourier Transformation (see section 15.5 on page 152)
- Classification on page 119 (for more detailed information about classification see the Solution Guide II-D)
- Texture Analysis on page 143

1.16 Text Processing

This chapter describes how HALCON processes text. Although HALCON is mainly an image processing library, there is also some text processing in HALCON as the following examples demonstrate:

- Passing string parameters such as file names or data code contents to and from HALCON operators via programming language interfaces.
- Files that are opened, read or written with HALCON operators.
- Strings that are transmitted via sockets.
- Reading characters with the help of OCR classifiers, which make use of user defined class names.
- String processing with the help of HALCON's tuple operators, e.g., by means of regular expressions.

1.16.1 String Encoding

All proprietary HALCON files, i.e. files whose format is controlled by HALCON, encode strings in UTF-8 in order to allow exchanging these files between different countries, locales, and operating systems. This affects mainly HALCON tuples as well as OCR and OCV classifiers, training data, and sample identification models, which all contain user defined class or character names. This is also true when the data is serialized. Note, that the encoding affects only strings that contain special characters, i.e. characters that are not plain ASCII.

For backwards compatibility, the operator `set_system` offers two new options. They can be used to control the encoding when old and new HALCON versions have to be used in parallel.

Chapter 2

Image Acquisition

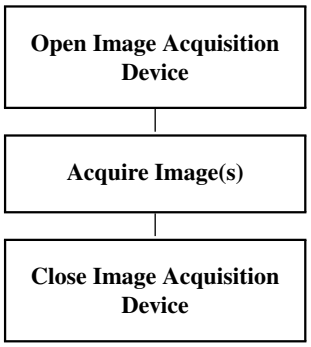
Obviously, the acquisition of images is a task that must be solved in all machine vision applications. Unfortunately, this task mainly consists of interacting with special, non-standardized hardware in the form of the image acquisition device, e.g., a frame grabber board or an IEEE 1394 camera. To let you concentrate on the actual machine vision problem, HALCON provides you with interfaces performing this interaction for a large number of image acquisition devices (see http://www.mvtec.com/products/interfaces for the latest information).

Within your HALCON application, the task of image acquisition is thus reduced to a few lines of code, i.e., a few operator calls. What's more, this simplicity is not achieved at the cost of limiting the available functionality: Using HALCON, you can acquire images from various configurations of acquisition devices and cameras in different timing modes.

Besides acquiring images from cameras, HALCON also allows you to input images that were stored in files (supported formats: BMP, TIFF, GIF, JPEG, PNG, PNM, PCX, XWD). Of course, you can also store acquired images in files.

2.1 Basic Concept

Acquiring images with HALCON basically consists of three steps. Reading images from files is even simpler: It consists of a single call to the operator read_image.



2.1.1 Open Image Acquisition Device

If you want to acquire images from a frame grabber board or an image acquisition device like an IEEE 1394 camera, the first step is to connect to this device. HALCON relieves you of all device-specific details; all you need to do is to call the operator open_framegrabber, specifying the name of the corresponding image acquisition interface.

There is also a "virtual" image acquisition interface called File. As its name suggests, this "frame grabber" reads images from files, and also from so-called image sequence files. The latter are HALCON-specific files, typically

with the extension .seq; they contain a list of image file names, separated by new lines (you can create it easily using a text editor). If you connect to such a sequence, subsequent calls to grab_image return the images in the sequence specified in the file. Alternatively, you can also read all images from a specific directory. Then, you do not have to create a sequence file, but simply specify the directory name instead of the sequence file as value for the parameter 'CameraType'. Now, subsequent calls to grab_image return the images found in the specified image directory. Both approaches are useful if you want to test your application with a sequence of image files and later switch to a real image acquisition device.

2.1.2 Acquire Image(s)

Having connected to the device, you acquire images by simply calling grab_image.

To load an image from disk, you use read_image. Images are searched for in the current directory and in the directories specified in the environment variable HALCONIMAGES.

2.1.3 Close Image Acquisition Device

At the end of the application, you close the connection to the image acquisition device to free its resources with the operator close_framegrabber.

2.1.4 A First Example

As already remarked, acquiring images from file corresponds to a single operator call:

```
read_image (Image, 'particle')
```

The following code processes images read from an image sequence file:

```
SequenceName := 'pendulum/pendulum.seq'
open_framegrabber ('File', -1, -1, -1, -1, -1, -1, 'default', -1, 'default', \
                   -1, 'default', SequenceName, 'default', -1, -1, \
                   AcqHandle)
while (ImageNum <= MaxImageNum)
    grab_image (Image, AcqHandle)
    ... process image ...
    ImageNum := ImageNum + 1
endwhile
```

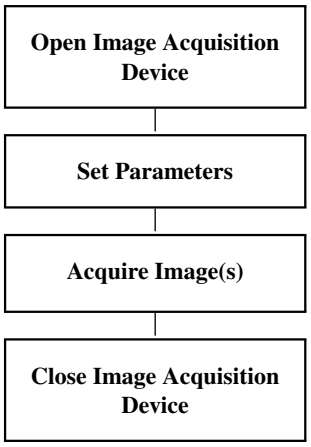
2.2 Extended Concept

In real applications, it is typically not enough to tell the camera to acquire an image; instead, it may be important that images are acquired at the correct moment or rate, and that the camera and the image acquisition interface are configured suitably. Therefore, HALCON allows to further parameterize the acquisition process. In HDevelop, an assistant is provided via the menu item Assistants > Image Acquisition that assists you when selecting your image source, adjusting the parameters, and generating suitable code.

2.2.1 Open Image Acquisition Device

When connecting to your image acquisition device with open_framegrabber, the main parameter is the name of the corresponding HALCON image acquisition interface. As a result, you obtain a so-called handle, with which you can access the device later, e.g., to acquire images with grab_image or grab_image_async.

With other parameters of open_framegrabber you can describe the configuration of image acquisition device(s) and camera(s), which is necessary when using more complex configurations, e.g., multiple cameras connected to different ports on different frame grabber boards. Further parameters allow you to specify the desired image format



(size, resolution, pixel type, color space). For most of these parameters there are default values that are used if you specify the values `'default'` (string parameters) or `-1` (numeric parameters).

With the operator `info_framegrabber` you can query information like the version number of the interface or the available boards, port numbers, and camera types.

Detailed information about the parameters of `open_framegrabber` can be found in the Solution Guide II-A (configuring the connection: section 3 on page 11; configuring the acquired image: section 4 on page 17).

2.2.2 Set Parameters

As described above, you already set parameters when connecting to the image acquisition device with `open_framegrabber`. These parameters (configuration of image_acquisition device(s) / camera(s) and image size etc.) are the so-called *general parameters*, because they are common to almost all image acquisition interfaces. However, image acquisition devices differ widely regarding the provided functionality, leading to many more *special parameters*. These parameters can be customized with the operator `set_framegrabber_param`.

With the operator `get_framegrabber_param` you can query the current values of the common and special parameters.

Detailed information about setting parameters can be found in the Solution Guide II-A in section 4 on page 17.

2.2.3 Acquire Image(s)

Actually, in a typical machine vision application you will not use the operator `grab_image` to acquire images, but `grab_image_async`. The difference between these two operators is the following: If you acquire and process images in a loop, `grab_image` always requests the acquisition of a new image and then blocks the program until the acquisition has finished. Then, the image is processed, and afterwards, the program waits for the next image. When using `grab_image_async`, in contrast, images are acquired and processed in parallel: While an image is processed, the next image is already being acquired. This, of course, leads to a significant speedup of the applications.

HALCON offers many more modes of acquiring images, e.g., triggering the acquisition by external signals or acquiring images simultaneously from multiple cameras. Detailed information about the various modes of acquiring images can be found in the Solution Guide II-A in section 5 on page 21.

2.3 Programming Examples

Example programs for all provided image acquisition interfaces can be downloaded via the “MVTec Software Manager” (SOM) or at `http://www.mvtec.com/products/interfaces`. Further examples are described in the Solution Guide II-A.

2.4 Tips & Tricks

2.4.1 Direct Access to External Images in Memory

You can also pass externally created images, i.e., the raw image matrix in the computer's memory, to HALCON using the operators gen_image1, gen_image3, gen_image1_extern or gen_image3_extern. For an example see the Solution Guide II-A, section 6.2 on page 34.

2.4.2 Unsupported Image Acquisition Devices

If you want to use an image acquisition device that is currently not supported by HALCON, i.e., for which no HALCON image acquisition interface exists, you can create your own interface. A description how to create and integrate an image acquisition interface as well as a template source code that can be used as the basis of an integration can be downloaded from MVTec's web server under http://www.mvtec.com/products/interfaces.

Chapter 3

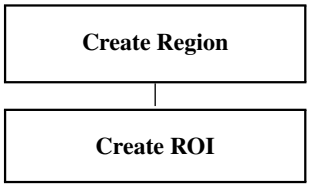
Region Of Interest

The concept of regions of interest (ROIs) is essential for machine vision in general and for HALCON in particular. The aim is to focus the processing on a specific part of the image. This approach combines region information with the image matrix: Only the image part corresponding to the region remains relevant, which reduces the number of pixels to be processed.

The advantages of using ROIs are manifold. First of all, it is a very good method to speed up a process because fewer pixels need to be processed. Furthermore, it focuses processing, e.g., a gray value feature is usually calculated only for a part of the image. Finally, ROIs are used to define templates, e.g., for matching. HALCON allows to make full use of the concept of ROIs because it enables using arbitrary shapes for the regions. This means that you are not limited to standard shapes like rectangles or polygons, but can really use *any* form - the best one to solve a given problem.

3.1 Basic Concept

Making use of ROIs is split into two simple parts: creating regions and combining them with the image.



3.1.1 Create Region

HALCON provides many ways to create regions, which can then be used as ROIs. The traditional way is to generate standard shapes like circles, ellipses, rectangles, or polygons. In addition, regions can be derived by converting them from other data types like XLD, by segmenting an image, or by user interaction.

3.1.2 Create ROI

By combining a region with an image, the region assumes the role of an ROI, i.e., it defines which part of the image must be processed. In HALCON, the ROI is also called the domain of the image. This term comes from mathematics where an image can be treated as a function that maps coordinates to gray values. An ROI reduces the domain of this function from the complete image to the relevant part. Therefore, the operator to combine regions and images is called `reduce_domain`. This simple operator fulfills the desired task in almost all applications.

3.1.3 A First Example

As an example for the basic concept, the following program shows all important steps to make use of an ROI. The image is acquired from file. Inside the image, only a circular part around the center should be processed. To achieve this, a circular region is generated with gen_circle. This region is combined with the image using reduce_domain. This has the effect that only the pixels of the ROI are processed when calling an operator. If, e.g., the operator edges_sub_pix is applied to this image, the subpixel accurate contours are extracted only inside the circle. To make this visible, some visualization operators are added to the end of the example program.

```
read_image (Image, 'mreut')
gen_circle (ROI, 256, 256, 200)
reduce_domain (Image, ROI, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'lanser2', 0.5, 20, 40)
dev_display (Image)
dev_display (ROI)
dev_display (Edges)
```

Figure 3.1: Processing the image only within the circular ROI.

3.2 Extended Concept

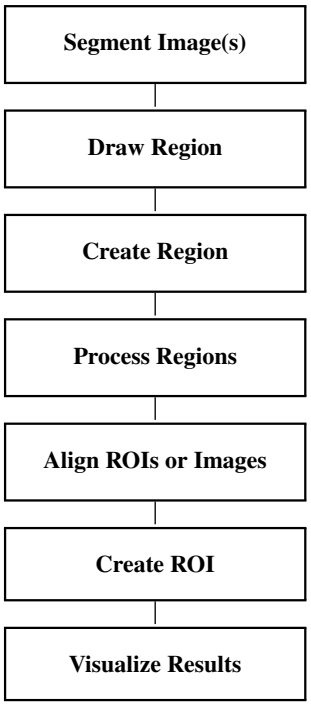
When we take a closer look at ROIs, extra steps become important if an application needs to be more flexible.

3.2.1 Segment Image(s)

Very typical for HALCON is the creation of ROIs by a segmentation step. Instead of having a predefined ROI, the parts of the image that are relevant for further processing are extracted from the image using image processing methods. This approach is possible because ROIs are nothing else but normal HALCON regions, and therefore share all their advantages like efficient processing and arbitrary shapes. The segmentation of regions used for ROIs follows the same approach as standard blob analysis. For more details, please refer to the description of this step on page 34.

3.2.2 Draw Region

The standard way to specify ROIs is to draw the shape interactively using the mouse. To make this easy, HALCON provides special operators for standard shapes and free-form shapes. All operators for this kind of action start with the prefix draw_. The drawing is performed by making use of the left mouse button (drawing, picking, and dragging) and finished by clicking the right mouse button. For each such draw-operator HALCON provides



operators to generate regions by using the returned parameters (see the description of the step Create Region on page 25). Operators for mouse interaction can be found in the reference manual in the chapter "Graphics ▷ Drawing". More information on user interaction can be also found in the chapter Visualization on page 223.

3.2.3 Create Region

The standard way is to generate regions based on the coordinates and dimensions returned by a user interaction or by coordinate values stored in a file. In this case, operators like `gen_circle`, `gen_rectangle2`, or `gen_region_polygon_filled` are used. More advanced are special shapes used to guide a preprocessing step to save execution time. Typical examples for this are grids of lines or dots or checker boards. With these shapes, the images can be covered in a systematic way and checked for specific object occurrences. If you want to segment, e.g., blobs of a given minimum size it is sufficient to use in a first step a search grid that is finer than the minimum object size to locate fragments. In a second step these fragments are dilated (`dilation_rectangle1`) and the segmentation method is called once again, now within this enlarged area. If the objects cover only a relatively small area of the image this approach can speed up the process significantly.

3.2.4 Process Regions

Sometimes the shape of a given ROI, either generated from the program or defined by the user, does not fulfill the requirements. Here, HALCON provides many operators to modify the shape to adapt it accordingly. Often used operators are, e.g., `fill_up` to fill holes inside the region, `shape_trans` to apply a general transformation like the convex hull or the smallest rectangle, or morphological operators like `erosion_circle` to make the region smaller or `closing_circle` to fill gaps. For more details, please refer to the description of this step on page 36.

3.2.5 Align ROIs or Images

Sometimes the coordinates of an ROI depend on the position of another object in the image. If the object moves, the ROI must be moved (aligned) accordingly. This is achieved by first locating the object using template matching. Based on the determined position and the orientation, the coordinates of the ROIs are then transformed.

How to perform alignment using shape-based matching is described in the Solution Guide II-B in section 2.5.3.2 on page 35.

3.2.6 Create ROI

This step combines the region and the image to make use of the region as the domain of the image. The standard method that is recommended to be used is `reduce_domain`. It has the advantage of being safe and having a simple semantics. `rectangle1_domain` is a shortcut for generating rectangular ROIs (instead of calling `gen_rectangle1` and `reduce_domain` in sequence). For advanced applications `change_domain` can be used as a slightly faster version than `reduce_domain`. This operator does not perform an intersection with the existing domain and does not check if the region is outside the image - which will cause a system crash when applying an operator to the data afterwards if the region lies partly outside the image. If the programmer ensures that the input region is well defined, this is a way to save (a little) execution time.

3.2.7 Visualize Results

Finally, you might want to display the ROIs or the reduced images. With the operator `get_domain`, the region currently used by the image can be accessed and displayed (and processed) like any other region. When displaying an image, e.g., with `disp_image`, only the defined pixels are displayed. Pixels in the graphics window outside the domain of the image will not be modified.

For detailed information see the description of this method on page 223.

3.3 Programming Examples

This section gives a brief introduction to programming ROIs in HALCON. Two examples show the principles of region generation, combining these with images, and then processing the data.

3.3.1 Processing inside a User Defined Region

Example: `%HALCONEXAMPLES%/solution_guide/basics/critical_points.hdev`

Figure 3.2 shows an image with marks that are used for a camera calibration in a 3D application. Here, we assume that the marks must be extracted in a given part of the image only.

a) b)

Figure 3.2: (a) Original image with drawn ROI; (b) reduced image with extracted points.

To achieve this, the user draws a region of interest with the mouse. The corresponding operator is `draw_region`. It has the window handle returned by `dev_open_window` as input and returns a region when the right mouse button is pressed. The operator `reduce_domain` combines this region with the image.

```
draw_region (Region, WindowHandle)
reduce_domain (Image, Region, ImageReduced)
```

When calling the point extraction operator `critical_points_sub_pix` on this reduced image, only points inside the ROI are found. The final part of the program shows how to display these points overlaid on the image.

```
critical_points_sub_pix (ImageReduced, 'facet', 1.5, 8, RowMin, ColumnMin, \
                         RowMax, ColumnMax, RowSaddle, ColSaddle)
dev_clear_window ()
dev_display (ImageReduced)
dev_set_color ('yellow')
for i := 0 to |RowSaddle| - 1 by 1
    gen_cross_contour_xld (Cross, RowSaddle[i], ColSaddle[i], 25, 0.785398)
    dev_display (Cross)
endfor
```

3.3.2 Interactive Partial Filtering of an Image

Example: %HALCONEXAMPLES%/solution_guide/basics/median_interactive.hdev

The task is to filter an image with a median filter only at the points where the user clicks with the mouse into the image, i.e., in the graphics window displaying the image.

Figure 3.3: Partially filtered image.

To do this, a loop is used inside which the mouse position is continuously requested with get_mposition. Because this operator throws an exception if the mouse is outside the graphics window the call is protected with dev_set_check.

```
Button := 0
while (Button != 4)
    Row := -1
    Column := -1
    dev_set_check ('~give_error')
    get_mposition (WindowHandle, Row, Column, Button)
    dev_set_check ('give_error')
```

If the mouse is over the window, a circular region is displayed, which shows where the filter would be applied.

```
    if (Row >= 0 and Column >= 0)
        gen_circle (Circle, Row, Column, 20)
        boundary (Circle, RegionBorder, 'inner')
        dev_display (RegionBorder)
```

If the left mouse button is pressed, median_image must be applied in the local neighborhood of the current mouse position. This is done by generating a circle with gen_circle and then calling reduce_domain.

```
        if (Button == 1)
            reduce_domain (Image, Circle, ImageReduced)
```

Now, the filter is called with this reduced image and the result is painted back into the input image for possible repetitive filtering. The loop will be terminated when the right mouse button is clicked.

```
            median_image (ImageReduced, ImageMedian, 'circle', 5, \
                          'mirrored')
            overpaint_gray (Image, ImageMedian)
        endif
```

3.3.3 Inspecting the Contours of a Tool

Example: %HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/circles.hdev

The task of this example is to inspect the contours of the tool depicted in figure 3.4.

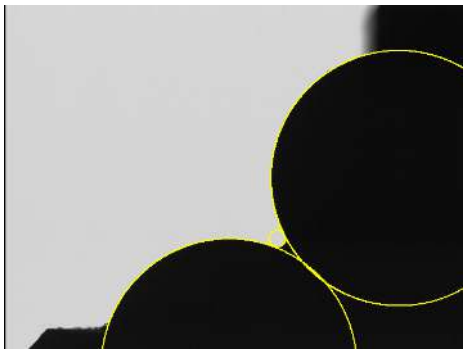


Figure 3.4: Fitting circles to the contours of the tool.

Because the subpixel-precise contour extraction is time-consuming, in a first step an ROI is created via a standard blob analysis (see Blob Analysis on page 33): With a threshold operator the object to be measured is extracted. This region is converted to its boundary, omitting the pixels at the image border.

```
fast_threshold (Image, Region, 0, 120, 7)
boundary (Region, RegionBorder, 'inner')
clip_region_rel (RegionBorder, RegionClipped, 5, 5, 5, 5)
```

The result is a small region close to the edge of the object. The boundary of the region, i.e., the edge, is dilated to serve as the ROI for the edge extraction. Now, the subpixel-precise edge extractor is called and the contour is segmented into straight lines and circular arcs.

```
dilation_circle (RegionClipped, RegionDilation, 2.5)
reduce_domain (Image, RegionDilation, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'canny', 2, 20, 60)
segment_contours_xld (Edges, ContoursSplit, 'lines_circles', 5, 4, 3)
```

For the segments that represent arcs, the corresponding circle parameters are determined. For inspection purposes, circles with the same parameters are generated and overlaid on the image (see also Contour Processing on page 79).

```
    get_contour_global_attrib_xld (ObjectSelected, 'cont_approx', Attrib)
    if (Attrib > 0)
        fit_circle_contour_xld (ObjectSelected, 'ahuber', -1, 2, 0, 3, 2, \
                                Row, Column, Radius, StartPhi, EndPhi, \
                                PointOrder)
        gen_circle_contour_xld (ContCircle, Row, Column, Radius, 0, \
                                rad(360), 'positive', 1.0)
        dev_display (ContCircle)
    endif
```

3.4 Relation to Other Methods

One class of operators does not follow the standard rules for ROI handling: the operators of the measure tool (see the description of this method on page 45). Here, the ROI is defined during the creation of a tool (gen_measure_arc and gen_measure_rectangle2) by specifying the coordinates as numeric values. The domain defined for the image will be ignored in this case.

3.5 Tips & Tricks

3.5.1 Reuse ROI

If an ROI is used multiple times it is useful to save the region to file and load it at the beginning of the application. Storing to file is done using write_region, loading with read_region.

3.5.2 Effect of ROI Shape on Speed Up

ROIs are a perfect way to save execution time: The smaller the ROI, the faster the application. This can be used as a general rule. If we consider this in more detail, we also need to think about the shape of ROIs. Because ROIs are based on the HALCON regions they use runlength encoding. This type of encoding is perfect if the runs are long. Therefore, a horizontal line can be both stored and processed more efficiently than a vertical line. This holds as well for the processing time of ROIs. Obviously this type of overhead is very small and can only be of importance with very fast operators like threshold.

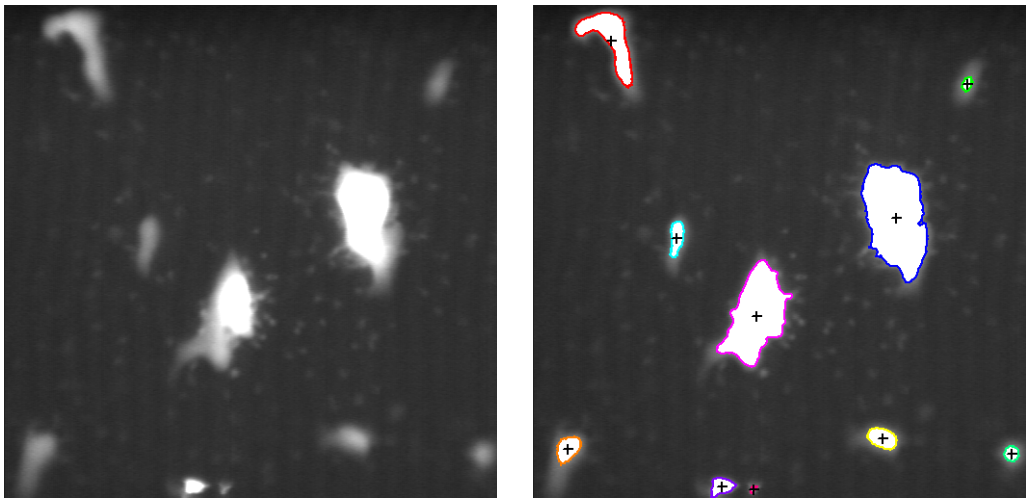
3.5.3 Binary Images

In some applications, it might be necessary to use ROIs that are available as binary images. To convert these to HALCON regions, you must use gen_image1 to convert them into a HALCON image, followed by threshold to generate the region. The conversion back can easily be achieved using region_to_bin followed by get_image_pointer1. It is also possible to import binary image files using read_region.

Chapter 4

Blob Analysis

The idea of blob analysis is quite easy: In an image the pixels of the relevant objects (also called foreground) can be identified by their gray value. For example, figure 4.1 shows tissue particles in a liquid. These particles are bright and the liquid (background) is dark. By selecting bright pixels (thresholding) the particles can be detected easily. In many applications this simple condition of dark and bright pixels no longer holds, but the same results can be achieved with extra pre-processing or alternative methods for pixel selection / grouping.



a) b)

Figure 4.1: Basic idea of blob analysis: (a) original image, (b) extracted blobs with calculated center points.

The advantage of blob analysis is the extreme flexibility that comes from the huge number of operators that HALCON offers in this context. Furthermore, these methods typically have a very high performance. Methods known from blob analysis can also be combined with many other vision tasks, e.g., as a pre-processing step for a flexible generation of regions of interest.

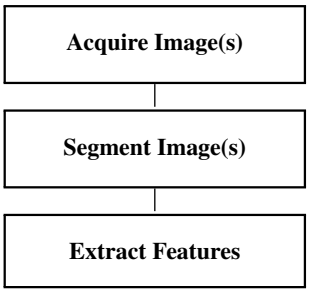
4.1 Basic Concept

Blob analysis mainly consists of three parts:

4.1.1 Acquire Image(s)

First, an image is acquired.

For detailed information see the description of this method on page 21.



4.1.2 Segment Image(s)

Having acquired the image, the task is to select the foreground pixels. This is also called segmentation. The result of this process typically is referred to as blobs (binary large objects). In HALCON, the data type is called a region.

4.1.3 Extract Features

In the final step, features like the area (i.e., the number of pixels), the center of gravity, or the orientation are calculated.

4.1.4 A First Example

An example for this basic concept is the following program, which belongs to the example explained above. Here, the image is acquired from file. All pixels that are brighter than 120 are selected using `threshold`. Then, an extra step is introduced which is not so obvious: The operator `connection` separates the set of all bright pixels into so called connected components. The effect of this step is that we now have multiple regions instead of the single region that is returned by `threshold`. The last step of this program is the calculation of some features. Here, the operator `area_center` determines the size (number of pixels) and the center of gravity. Please note that `area_center` returns multiple values for all three feature parameters (one value for each connected component).

```
read_image (Image, 'particle')
threshold (Image, BrightPixels, 120, 255)
connection (BrightPixels, Particles)
area_center (Particles, Area, Row, Column)
```

4.2 Extended Concept

In many cases the segmentation of blobs will be more advanced than in the above example. Reasons for this are, e.g., clutter or inhomogeneous illumination. Furthermore, postprocessing like transforming the features to real world units or visualization of results are often required.

4.2.1 Use Region of Interest

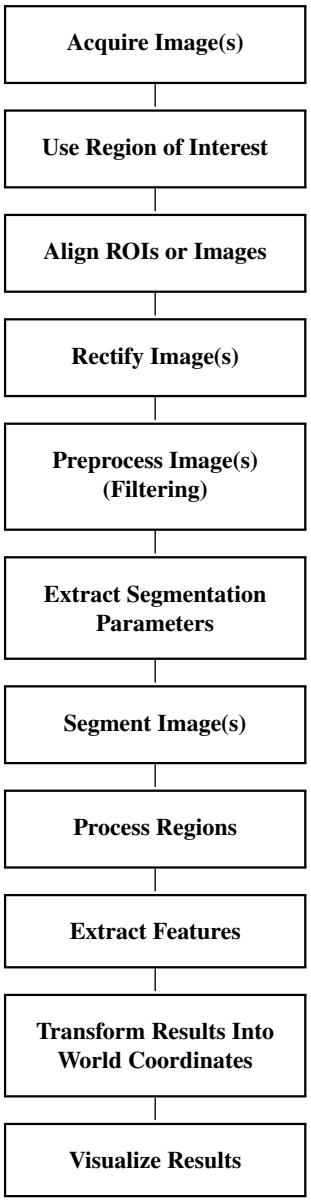
Blob analysis can be sped up by using a region of interest. The more the region in which the blobs are searched can be restricted, the faster and more robust the search will be.

For detailed information see the description of this method on page 25.

4.2.2 Align ROIs or Images

In some applications, the regions of interest must be aligned relative to another object. Alternatively, the image itself can be aligned, e.g., by rotating or cropping it.

How to perform alignment using shape-based matching is described in the Solution Guide II-B in section 2.5.3.2 on page 35.



4.2.3 Rectify Image(s)

Similarly to alignment, it may be necessary to rectify the image, e.g., to remove lens distortions or to transform the image into a reference point of view.

Detailed information about rectifying images can be found in the Solution Guide III-C in section 3.4 on page 80.

4.2.4 Preprocess Image(s) (Filtering)

The next important part is the pre-processing of the image. Here, operators like `mean_image` or `gauss_filter` can be used to eliminate noise. A fast but less perfect alternative to `gauss_filter` is `binomial_filter`. The operator `median_image` is useful for suppressing small spots or thin lines. The operator `anisotropic_diffusion` is useful for edge-preserving smoothing, and finally `fill_interlace` is used to eliminate defects caused by interlaced cameras.

4.2.5 Extract Segmentation Parameters

Instead of using fixed threshold values, they can be extracted dynamically for each image. One example for this is a gray value histogram that has multiple peaks, one for each object class. Here, you can use the operators

gray_histo_abs and histo_to_thresh.

As an advanced alternative, you can use the operator intensity in combination with a reference image that contains only background: During setup, you determine the mean gray value of a background region. During the inspection, you again determine this mean gray value. If it has changed, you adapt the threshold accordingly.

4.2.6 Segment Image(s)

For the segmentation various methods can be used. The most simple method is threshold, where one or more gray value ranges that belong to the foreground objects are specified. Another very common method is dyn_threshold. Here, a second image is passed as a reference. With this approach, a local threshold instead of a global threshold is used. These local threshold values are stored in the reference image. The reference image can either be static by taking a picture of the empty background or can be determined dynamically with smoothing filters like mean_image.

4.2.7 Process Regions

Once blob regions are segmented, it is often necessary to modify them, e.g., by suppressing small areas, regions of a given orientation, or regions that are close to other regions. In this context, the morphological operators opening_circle and opening_rectangle1 are often used to suppress noise and closing_circle and closing_rectangle1 to fill gaps.

Blobs with a specific feature can be selected with select_shape, select_shape_std, and select_shape_proto.

4.2.8 Extract Features

To finalize the image processing, features of the blobs are extracted. The type of features needed depends on the application. A full list can be found in the Reference Manual in the chapters "Regions ▷ Features" and "Image ▷ Features".

4.2.9 Transform Results Into World Coordinates

Features like the area or the center of gravity often must be converted to world coordinates. This can be achieved with the HALCON camera calibration.

How to transform results into world coordinates is described in detail in the Solution Guide III-C in section 3.3 on page 76.

4.2.10 Visualize Results

Finally, you might want to display the images, the blob (regions), and the features.

For detailed information see the description of this method on page 223.

4.3 Programming Examples

This section gives a brief introduction to using HALCON for blob analysis.

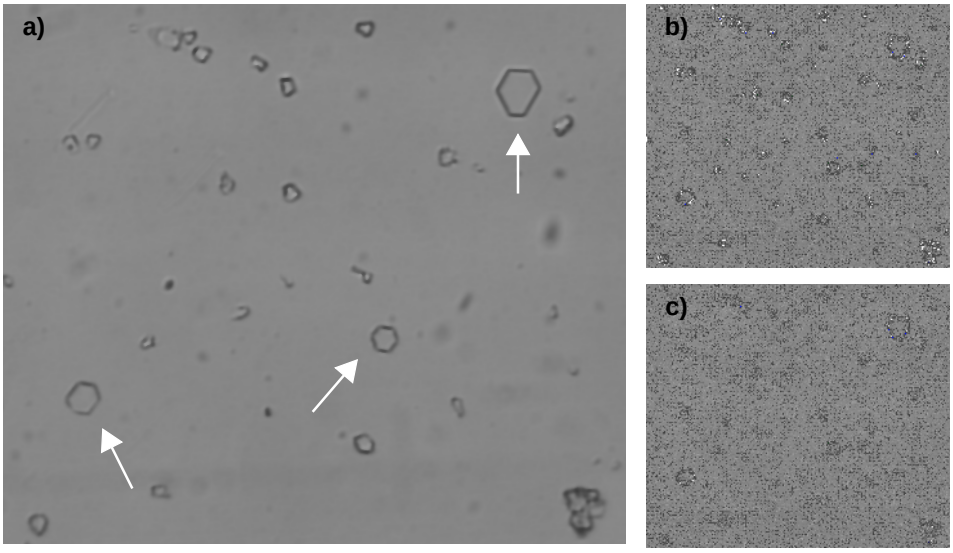


Figure 4.2: Extracting hexagonal crystals: (a) original image with arrows marking the crystals to be extracted, (b) result of the initial segmentation, (c) finally selected blobs.

4.3.1 Crystals

Example: `%HALCONEXAMPLES%/solution_guide/basics/crystal.hdev`

Figure 4.2a shows an image taken in the upper atmosphere with collected samples of crystals. The task is to analyze the objects to determine the frequency of specific shapes. One of the important objects are the hexagonally shaped crystals.

First, the image is read from file with `read_image`. The segmentation of objects is performed with a local threshold because of the relatively low contrast of the crystals combined with a non-homogeneous background. The background is determined with the average filter `mean_image`. The filter mask size is selected such that it has about three times the width of the dark areas. `dyn_threshold` now compares the smoothed with the original gray values, selecting those pixels that are darker by a contrast of 8 gray values. `connection` separates the objects into connected components. Figure 4.2b shows the result of this initial segmentation.

```
read_image (Image, 'crystal')
mean_image (Image, ImageMean, 21, 21)
dyn_threshold (Image, ImageMean, RegionDynThresh, 8, 'dark')
connection (RegionDynThresh, ConnectedRegions)
```

In the following processing step, the task now is to select only the hexagonally shaped crystals. For this, they are first transformed into their convex hull. This is like putting a rubber band around each region. From these regions, those that are big (`select_shape`) and have a given gray value distribution (`select_gray`) are selected. The parameters for the selection are determined so that only the relevant crystals remain (see figure 4.2c).

```
shape_trans (ConnectedRegions, ConvexRegions, 'convex')
select_shape (ConvexRegions, LargeRegions, 'area', 'and', 600, 2000)
select_gray (LargeRegions, Image, Crystals, 'entropy', 'and', 1, 5.6)
```

4.3.2 Atoms

Example: `%HALCONEXAMPLES%/solution_guide/basics/atoms.hdev`

Specialized microscopes are able to determine the rough location of single atoms. This is useful, e.g., to analyze the grid change of crystals at a p-n-junction. A segmentation that works perfectly well on images like these is the watershed method. Here, each dark basin is returned as a single region. Because at the outer part of the image atoms are only partially visible, the first task is to extract only those that are not close to the image border. Finally,

the irregularity is extracted. This is done by looking for those atoms that have an abnormal (squeezed) shape (see figure 4.3).

```
gauss_filter (Image, ImageGauss, 5)
watersheds (ImageGauss, Basins, Watersheds)
select_shape (Basins, SelectedRegions1, 'column1', 'and', 2, Width - 1)
select_shape (SelectedRegions1, SelectedRegions2, 'row1', 'and', 2, \
              Height - 1)
select_shape (SelectedRegions2, SelectedRegions3, 'column2', 'and', 1, \
              Width - 3)
select_shape (SelectedRegions3, Inner, 'row2', 'and', 1, Height - 3)
select_shape (Inner, Irregular, ['moments_i1', 'moments_i1'], 'or', [0, \
              9.5e8], [1.5e8, 1e10])
```

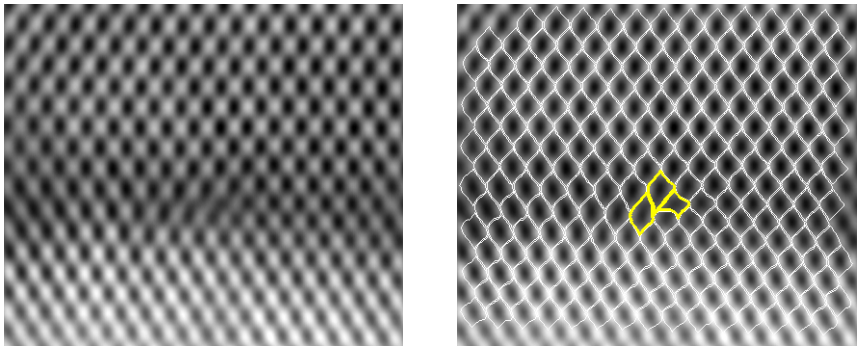
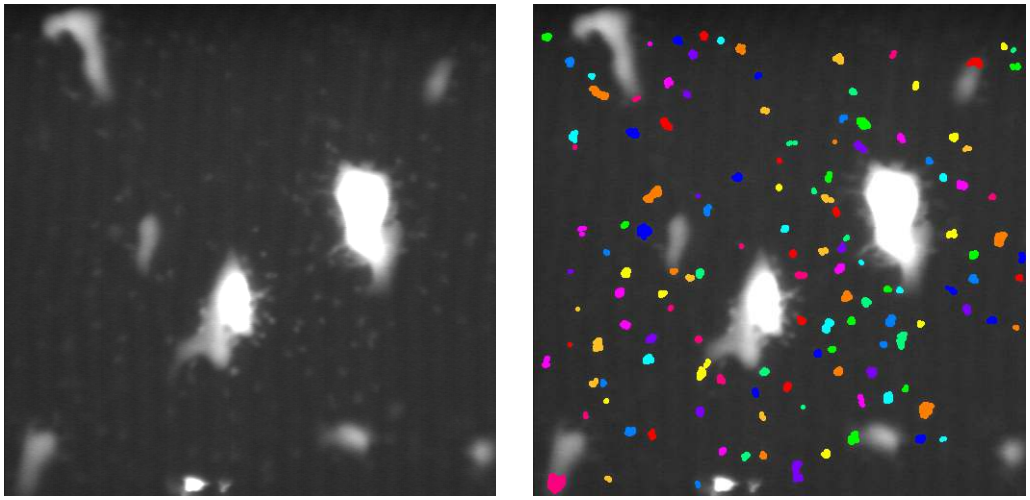


Figure 4.3: Inspecting atom structure.

4.3.3 Analyzing Particles

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/particle.hdev`

The task of this example is to analyze particles in a liquid. The main difficulty in this application is the presence of two types of objects: big bright objects and small objects with low contrast. In addition, the presence of noise complicates the segmentation.



a) **b)**

Figure 4.4: Extracting the small particles: (a) original image, (b) result.

The program segments the two classes of objects separately using two different methods: global and local thresholding. With additional post-processing, the small particles can be extracted in a robust manner.

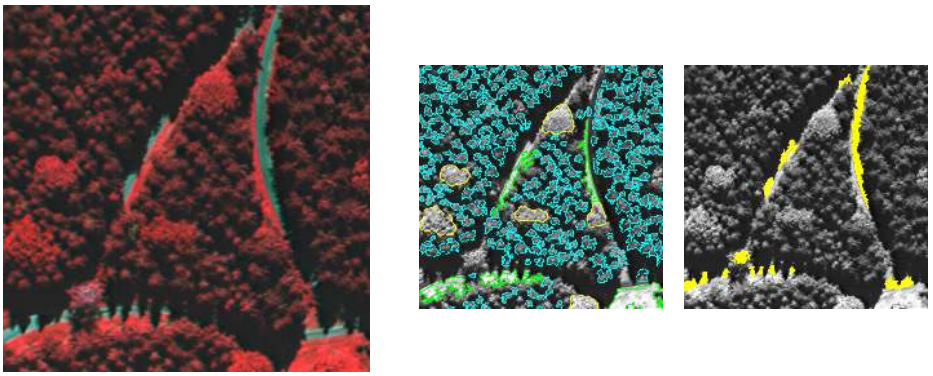
```
threshold (Image, Large, 110, 255)
dilation_circle (Large, LargeDilation, 7.5)
complement (LargeDilation, NotLarge)
reduce_domain (Image, NotLarge, ParticlesRed)

mean_image (ParticlesRed, Mean, 31, 31)
dyn_threshold (ParticlesRed, Mean, SmallRaw, 3, 'light')
opening_circle (SmallRaw, Small, 2.5)
connection (Small, SmallConnection)
```

4.3.4 Extracting Forest Features from Color Infrared Image

Example: %HALCONEXAMPLES%/hdevelop/Applications/Object-Recognition-2D/forest.hdev

The task of this example is to detect different object classes in the color infrared image depicted in figure 4.5: trees (coniferous and deciduous), meadows, and roads.



a) b) c)

Figure 4.5: (a) Original image; (b) extracted trees and meadows; (c) extracted roads.

The image data is a color infrared image, which allows to extract roads very easily because of their specific color. For that, the multi-channel image is split into its three channels and each channel is investigated individually. Further information about decomposing multi-channel images can be found in Color Processing on page 131.

```
read_image (Forest, 'forest_air1')
decompose3 (Forest, Red, Green, Blue)
threshold (Blue, BlueBright, 80, 255)
connection (BlueBright, BlueBrightConnection)
select_shape (BlueBrightConnection, Path, 'area', 'and', 100, 100000000)
```

Beech trees are segmented in the red channel based on their intensity and minimum size.

```
threshold (Red, RedBright, 120, 255)
connection (RedBright, RedBrightConnection)
select_shape (RedBrightConnection, RedBrightBig, 'area', 'and', 1500, \
              10000000)
closing_circle (RedBrightBig, RedBrightClosing, 7.5)
opening_circle (RedBrightClosing, RedBrightOpening, 9.5)
connection (RedBrightOpening, RedBrightOpeningConnection)
select_shape (RedBrightOpeningConnection, BeechBig, 'area', 'and', 1000, \
              100000000)
select_gray (BeechBig, Blue, Beech, 'mean', 'and', 0, 59)
```

Meadows have similar spectral properties, but are slightly brighter.

```
union1 (Beech, BeechUnion)
complement (BeechUnion, NotBeech)
difference (NotBeech, Path, NotBeechNotPath)
reduce_domain (Red, NotBeechNotPath, NotBeechNotPathRed)
threshold (NotBeechNotPathRed, BrightRest, 150, 255)
connection (BrightRest, BrightRestConnection)
select_shape (BrightRestConnection, Meadow, 'area', 'and', 500, 1000000)
```

The coniferous trees are extracted using the watershed approach with an additional thresholding inside the basins to get rid of the shadow areas.

```
union2 (Path, RedBrightClosing, BeechPath)
smooth_image (Red, RedGauss, 'gauss', 4.0)
invert_image (RedGauss, Invert)
watersheds (Invert, SpruceRed, Watersheds)
select_shape (SpruceRed, SpruceRedLarge, 'area', 'and', 100, 5000)
select_gray (SpruceRedLarge, Red, SpruceRedInitial, 'max', 'and', 100, 200)
gen_empty_obj (LocalThresh)
count_obj (SpruceRedInitial, NumSpruce)
for i := 1 to NumSpruce by 1
    select_obj (SpruceRedInitial, SingleSpruce, i)
    min_max_gray (SingleSpruce, Red, 50, Min, Max, Range)
    reduce_domain (Red, SingleSpruce, SingleSpruceRed)
    threshold (SingleSpruceRed, SingleSpruceBright, Min, 255)
    connection (SingleSpruceBright, SingleSpruceBrightCon)
    select_shape_std (SingleSpruceBrightCon, MaxAreaSpruce, 'max_area', 70)
    concat_obj (MaxAreaSpruce, LocalThresh, LocalThresh)
endfor
opening_circle (LocalThresh, FinalSpruce, 1.5)
```

4.3.5 Checking a Boundary for Fins

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/fin.hdev`

The task of this example is to check the outer boundary of a plastic part. In this case, some objects show fins that are not allowed for faultless pieces (see figure 4.6).

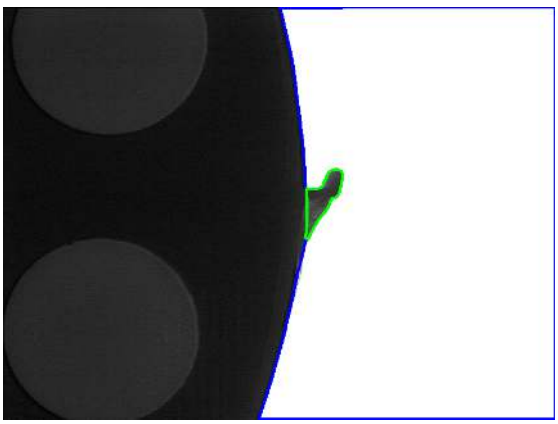


Figure 4.6: Boundary with extracted fin.

The program first extracts the background region (in which the fin appears as an indentation).

```
    binary_threshold (Fin, Background, 'max_separability', 'light', \
                      UsedThreshold)
```

This indentation in the background region is then closed using a morphological operator.

```
closing_circle (Background, ClosedBackground, 250)
```

Significant differences between the closed region and the original region are production errors.

```
difference (ClosedBackground, Background, RegionDifference)
opening_rectangle1 (RegionDifference, FinRegion, 5, 5)
```

4.3.6 Bonding Balls

Example: %HALCONEXAMPLES%/hdevelop/Applications/Completeness-Check/ball.hdev

The task of this example is to inspect the diameter of the ball bonds depicted in figure 4.7.

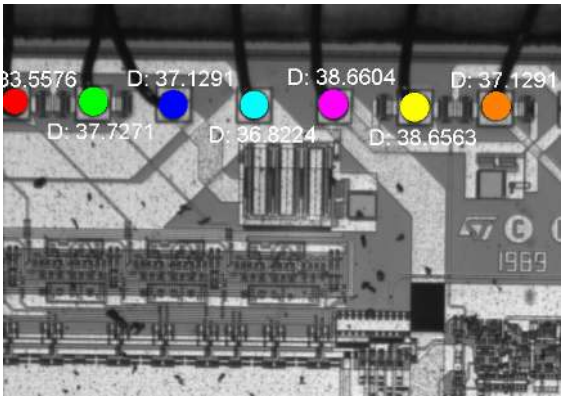


Figure 4.7: Measuring the diameter of ball bonds.

The extraction of the ball bonds is a two step approach: First, the die is located by segmenting bright areas and transforming them into their smallest surrounding rectangle.

```
threshold (Bond, Bright, 100, 255)
shape_trans (Bright, Die, 'rectangle2')
```

Now, the processing is focused to the region inside the die using `reduce_domain`. In this ROI, the program checks for dark areas that correspond to wire material.

```
reduce_domain (Bond, Die, DieGrey)
threshold (DieGrey, Wires, 0, 50)
fill_up_shape (Wires, WiresFilled, 'area', 1, 100)
```

After removing irrelevant structures and arranging the bonds in a predefined order, the desired features are extracted.

```
opening_circle (WiresFilled, Balls, 15.5)
connection (Balls, SingleBalls)
select_shape (SingleBalls, IntermediateBalls, 'circularity', 'and', 0.85, \
              1.0)
sort_region (IntermediateBalls, FinalBalls, 'first_point', 'true', 'column')
smallest_circle (FinalBalls, Row, Column, Radius)
```

4.3.7 Surface Scratches

Example: %HALCONEXAMPLES%/solution_guide/basics/surface_scratch.hdev

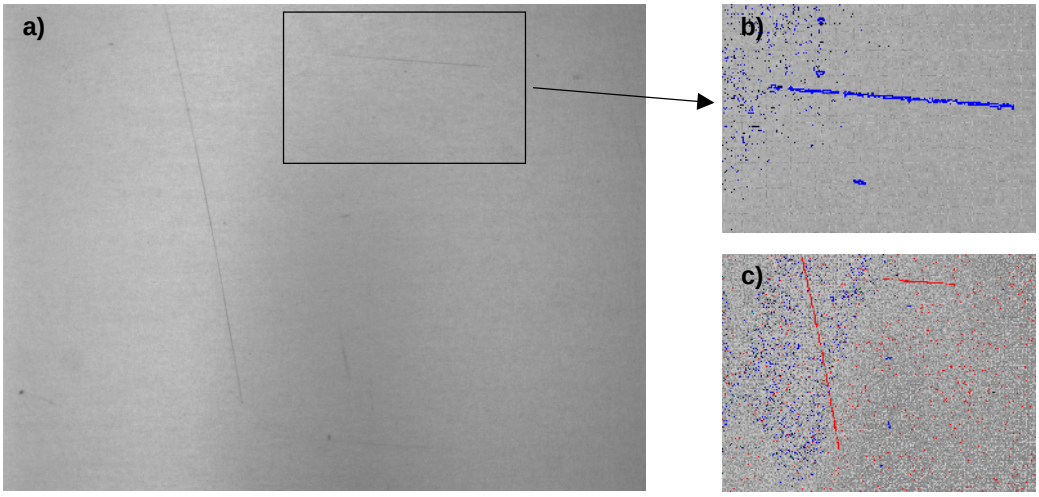


Figure 4.8: Detecting scratches on a metal surface: (a) original image, (b) extracted scratches still partly fractioned, (c) final result with merged scratches.

This example detects scratches on a metal surface (see figure 4.8).

The main difficulties for the segmentation are the inhomogenous background and the fact that the scratches are thin structures. Both problems can be solved using a local threshold, i.e., the operators `mean_image` and `dyn_threshold`. After `connection`, the small objects that are mainly noise are removed (see figure 4.8b).

```
mean_image (Image, ImageMean, 7, 7)
dyn_threshold (Image, ImageMean, DarkPixels, 5, 'dark')
connection (DarkPixels, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 10, 1000)
```

The scratches are part of the selection, but if we look closely we see that they are partially fractioned. To solve this problem we combine all fractions again into one big region. By applying `dilation_circle`, neighboring parts with a given maximum distance are now combined. To finally get the correct shape of the scratches - which are now too wide because of the dilation - `skeleton` is used to thin the shape to a width of one pixel.

```
union1 (SelectedRegions, RegionUnion)
dilation_circle (RegionUnion, RegionDilation, 3.5)
skeleton (RegionDilation, Skeleton)
connection (Skeleton, Errors)
```

The last step is to distinguish between small dots and scratches on the surface. This is achieved with `select_shape`, using the size as feature. Figure 4.8c depicts the result.

```
select_shape (Errors, Scratches, 'area', 'and', 50, 10000)
select_shape (Errors, Dots, 'area', 'and', 1, 50)
```

4.4 Relation to Other Methods

4.4.1 Methods that are Useful for Blob Analysis

Color Processing (see description on page 131)
Color processing can be considered as an advanced way of blob analysis that uses three color channels instead of one gray value channel. HALCON provides operators for color space transformation, feature extraction, and pixel classification, which can be used in combination with blob analysis.

Texture Analysis (see description on page 143)

Texture analysis is a method for finding regular or irregular structures, e.g., on the surface of an object and it is therefore useful if texture is a feature of the object to be inspected and simple blob analysis is not sufficient. For texture analysis not only the single gray values but also a larger pixel neighborhood is used. HALCON provides filters that emphasize or suppress specific textures. The result of these filters can then be segmented.

4.4.2 Methods that are Using Blob Analysis

OCR (see description on page 183)
Blob analysis is typically used as a preprocessing step for OCR to segment the characters.

4.4.3 Alternatives to Blob Analysis

Edge Extraction (Subpixel-Precise) (see description on page 63)
In blob analysis a region is described by the gray values of its pixels. As an alternative, a region could be described by the change of the gray values at their borders. This approach is called edge detection.

Classification (see description on page 119)
To select specific gray values, thresholds must be determined. In most cases, fixed values are used or the current value is determined by a feature operator. In some cases it is useful if the system determines the ranges automatically. This can be achieved by using a classifier. In addition, a classifier can also be used to automatically distinguish between good and bad objects based on the extracted features and samples for both classes.

4.5 Tips & Tricks

4.5.1 Connected Components

By default, most HALCON segmentation operators like `threshold` return one region even if you see multiple not connected areas on the screen. To transform this region into separated objects (i.e., connected components in the HALCON nomenclature) one has to call `connection`.

4.5.2 Speed Up

Many online applications require maximum speed. Because of its flexibility, HALCON offers many ways to achieve this goal. Here the most common methods are listed.

- Regions of interest are the standard method to increase the speed by processing only those areas where objects need to be inspected. This can be done using pre-defined regions but also by an online generation of the regions of interest that depend on other objects found in the image.
- If an object has a specific minimum size, the operator `fast_threshold` is a fast alternative to `threshold`. This kind of fast operator can also directly be generated by using operators like `gen_grid_region` and `reduce_domain` before calling the thresholding operator.
- By default, HALCON performs some data consistency checks. These can be switched off using `set_check`.
- By default, HALCON initializes new images. Using `set_system` with the parameter `"init_new_image"`, this behavior can be changed.

4.6 Advanced Topics

4.6.1 Line Scan Cameras

In general, line scan cameras are treated like normal area sensors. In some cases, however, not single images but an “infinite” sequence of images showing objects, e.g., on a conveyor belt, must be processed. In this case the end

of one image is the beginning of the next one. This means that objects that partially lie in both images must be combined into one object. For this purpose HALCON provides the operator `merge_regions_line_scan`. This operator is called after the segmentation of one image, and combines the current objects with those of previous images. For more information see the Solution Guide II-A.

4.6.2 High Accuracy

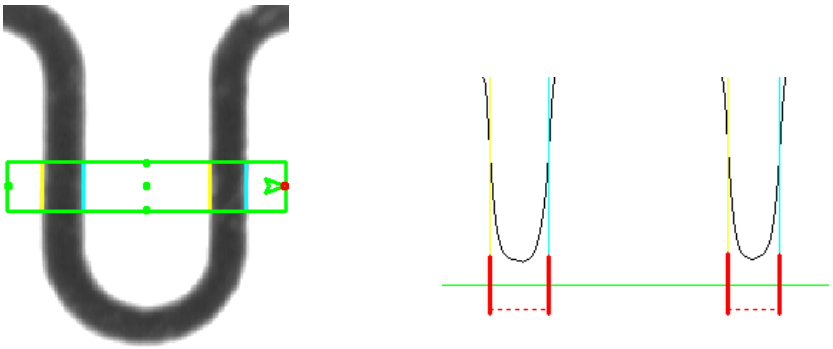
Sometimes high accuracy is required. This is difficult with blob analysis because objects are only extracted with integer pixel coordinates. Note, however, that many features that can be calculated for regions, e.g., the center of gravity, will be subpixel-precise. One way to get higher accuracy is to use a higher resolution. This has the effect that the higher number of pixels for each region results in better statistics to estimate features like the center of gravity (`area_center`). As an alternative, gray value features (like `area_center_gray`) can be used if the object fulfills specific gray value requirements. Here, the higher accuracy comes from the fact that for each pixel 255 values instead of one value (foreground or background) is used. If a very high accuracy is required, you should use the subpixel-precise edge and line extraction on page 63.

Chapter 5

1D Measuring

The idea of 1D measuring (also called 1D metrology or caliper) is very intuitive: Along a predefined region of interest, edges are located that are mainly perpendicular to the orientation of the region of interest. Here, edges are defined as transitions from dark to bright or from bright to dark.

Based on the extracted edges, you can measure the dimensions of parts. For example, you can measure the width of a part by placing a region of interest over it and locating the edges on its left and the right side. The effect of this can be seen in figure 5.1a, whereas figure 5.1b shows the corresponding gray value profile.



a) b)

Figure 5.1: (a) Measuring a fuse wire; (b) gray value profile along the region of measurement with extracted edges.

In addition to these simple rectangular regions of interest, circular arcs can be used to measure, e.g., the widths of the cogs on a cog wheel.

The advantage of the measure approach is its ease of use combined with a short execution time and a very high accuracy. With only a few operators, high-performing applications can be realized.

Alternatively, you can use HDevelop's Measure Assistant, which allows you to perform measurements with just a few mouse clicks. How to measure with this assistant is described in detail in the HDevelop User's Guide, section 7.4 on page 219.

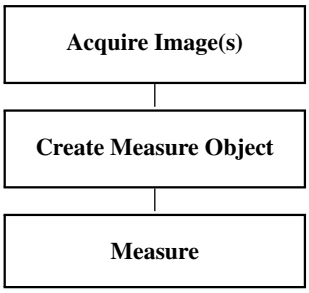
5.1 Basic Concept

Measuring consists of these main steps:

5.1.1 Acquire Image(s)

First, an image is acquired.

For detailed information see the description of this method on page 21.



5.1.2 Create Measure Object

Having acquired the image, you specify where to measure, i.e., you describe the position, orientation, etc. of the line or arc along which you want to measure. Together with some other parameters, this information is stored in the so-called measure object.

You access the measure object by using a so-called handle. Similarly to a file handle, this handle is needed when working with the tool. Each time the measure tool is executed, this handle is passed as a parameter.

In object-oriented languages like C++ it is possible to use the measure class instead of the low-level approach with handles. Here, creation and destruction are realized with the standard object-oriented methods.

5.1.3 Measure

Then, you can apply the measuring by specifying the measure object and some other vision parameters like, e.g., the minimum contrast. You can find detailed information about this step in the Solution Guide III-A in chapter 3 on page 15.

5.2 Extended Concept

In many cases, a measuring application will be more complex than described above. Reasons for this are, e.g., clutter or inhomogeneous illumination. Furthermore, post-processing like transforming the features to real-world units, or visualization of results may be required.

5.2.1 Radiometrically Calibrate Image(s)

To allow high-accuracy measurements, the camera should have a linear response function, i.e., the gray values in the images should depend linearly on the incoming energy. Since some cameras do not have a linear response function, HALCON provides the so-called radiometric calibration (gray value calibration): With the operator `radiometric_self_calibration` you can determine the inverse response function of the camera (offline) and then apply this function to the images using `lut_trans` before performing the measuring.

5.2.2 Align ROIs or Images

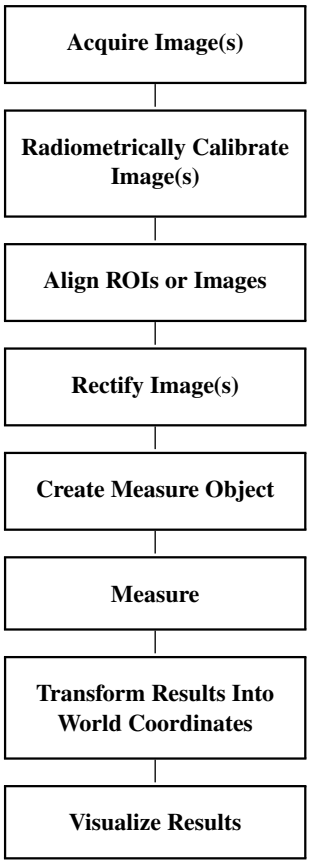
In some applications, the line or arc along which you want to measure, must be aligned relative to another object.

How to perform alignment using shape-based matching is described in the Solution Guide II-B in section 2.5.3.2 on page 35.

5.2.3 Rectify Image(s)

Similarly to alignment, it may be necessary to rectify the image, e.g., to remove lens distortion.

Detailed information about rectifying images can be found in the Solution Guide III-C in section 3.4 on page 80.



5.2.4 Create Measure Object

You can teach the measurement line or arc interactively with operators like `draw_rectangle2` or read its parameters from file (`read_string`). As an alternative, its coordinates can be generated based on the results of other vision tools like Blob Analysis (see the description of this method on page 33). In particular, the measurement line or arc may need to be aligned to a certain object as described above.

If the measurement is always performed along the same line or arc, you can create the measure object offline and then use it multiple times. However, if you want to align the measurement, the position and orientation of the line or arc will differ for each image. In this case, you must create a new measure object for each image. An exception to this rule is if only the position changes but not the orientation. Then, you can keep the measure object and adapt its position via `translate_measure`.

Please refer to the Solution Guide III-A, chapter 2 on page 9, for more information.

5.2.5 Transform Results Into World Coordinates

If you have calibrated your vision system, you can easily transform the results of measuring into world coordinates with `image_points_to_world_plane`. How to do this is described in the Solution Guide III-A in section 3.5 on page 22.

This is described in detail in the Solution Guide III-C in section 3.3 on page 76.

5.2.6 Visualize Results

The best way to visualize edge positions is to create (short) XLD line segments with operators like `gen_contour_polygon_xld`.

For detailed information see the description of this method on page 223.

5.3 Programming Examples

The following examples gives a brief introduction to using the 1D measuring tool of HALCON. The longest parts are pre- and postprocessing; the measurement itself consists only of two operator calls. Further examples are described in the Solution Guide III-A.

5.3.1 Inspecting a Fuse

Example: `%HALCONEXAMPLES%/solution_guide/basics/fuse.hdev`

Preprocessing consists of the generation of the measurement line. In the example program, this step is accomplished by assigning the measure object's parameters to variables.

```
read_image (Fuse, 'fuse')
Row := 297
Column := 545
Length1 := 80
Length2 := 10
Angle := rad(90)
gen_measure_rectangle2 (Row, Column, Angle, Length1, Length2, Width, Height, \
                        'bilinear', MeasureHandle)
```

Now the actual measurement is performed by applying the measure object to the image. The parameters are chosen such that edges around dark areas are grouped to so called pairs, returning the position of the edges together with the width and the distance of the pairs.

```
measure_pairs (Fuse, MeasureHandle, 1, 1, 'negative', 'all', RowEdgeFirst, \
               ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, \
               ColumnEdgeSecond, AmplitudeSecond, IntraDistance, \
               InterDistance)
```

The last part of the program displays the results by generating a region with the parameters of the measurement line and converting the edge positions to short XLD contours (see figure 5.2).

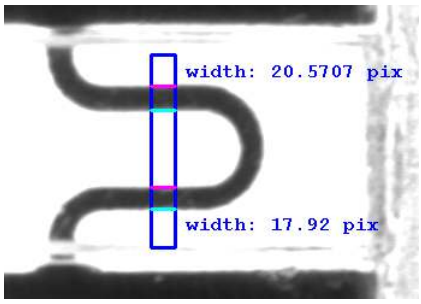


Figure 5.2: Measuring the width of the fuse wire.

```
for i := 0 to |RowEdgeFirst| - 1 by 1
    gen_contour_polygon_xld (EdgeFirst, \
                             [-sin(Angle + rad(90)) * Length2 + RowEdgeFirst[i], \
                             -sin(Angle - rad(90)) * Length2 + RowEdgeFirst[i]], \
                              [cos(Angle + rad(90)) * Length2 + ColumnEdgeFirst[i], \
                             cos(Angle - rad(90)) * Length2 + ColumnEdgeFirst[i]])
    gen_contour_polygon_xld (EdgeSecond, \
                              [-sin(Angle + rad(90)) * Length2 + RowEdgeSecond[i], \
                             -sin(Angle - rad(90)) * Length2 + RowEdgeSecond[i]], \
                              [cos(Angle + rad(90)) * Length2 + ColumnEdgeSecond[i], \
                             cos(Angle - rad(90)) * Length2 + ColumnEdgeSecond[i]])
    write_string (WindowID, 'width: ' + IntraDistance[i] + ' pix')
endfor
```

5.3.2 Inspect Cast Part

Example: %HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/measure_arc.hdev

The task of this example is to inspect the distance between elongated holes of a cast part after chamfering (see figure 5.3). Note that to achieve best accuracy it would be recommended to use backlight combined with a telecentric lens instead of the depicted setup.

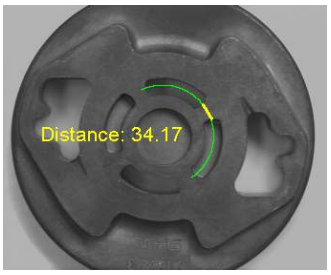


Figure 5.3: Measuring the distance between the holes.

This task can be solved easily by using the measure tool with a circular measurement ROI. The center of the ROI is placed into the center of the cast part; its radius is set to the distance of the elongated holes from the center.

```
Row := 275
Column := 335
Radius := 107
AngleStart := -rad(55)
AngleExtent := rad(170)
gen_measure_arc (Row, Column, Radius, AngleStart, AngleExtent, 10, Width, \
                 Height, 'nearest_neighbor', MeasureHandle)
```

Now, the distance between the holes can be measured with a single operator call:

```
measure_pos (Zeiss1, MeasureHandle, 1, 10, 'all', 'all', RowEdge, \
             ColumnEdge, Amplitude, Distance)
```

5.3.3 Inspecting an IC Using Fuzzy Measuring

Example: %HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/fuzzy_measure_pin.hdev

The task of this example is to inspect the lead width and the lead distance of the IC depicted in figure 5.4.

The illumination conditions in this example are quite difficult. This has the effect that four edges are visible for each lead. Fuzzy rules are used to restrict the measurement to the correct (outer) leads.

```
gen_measure_rectangle2 (Row1, Col1, Phi1, Length1, Length2, Width, Height, \
                        'nearest_neighbor', MeasureHandle1)
create_funct_1d_pairs ([0.0, 0.3], [1.0, 0.0], FuzzyAbsSizeDiffFunction)
set_fuzzy_measure_norm_pair (MeasureHandle1, 11.0, 'size_abs_diff', \
                             FuzzyAbsSizeDiffFunction)
fuzzy_measure_pairs (Image, MeasureHandle1, 1, 30, 0.5, 'positive', \
                     RowEdgeFirst1, ColumnEdgeFirst1, AmplitudeFirst1, \
                     RowEdgeSecond1, ColumnEdgeSecond1, AmplitudeSecond1, \
                     RowEdgeMiddle1, ColumnEdgeMiddle1, FuzzyScore1, \
                     IntraDistance1, InterDistance1)
```

5.3.4 Measuring Leads of a Moving IC

Example: %HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/pm_measure_board.hdev

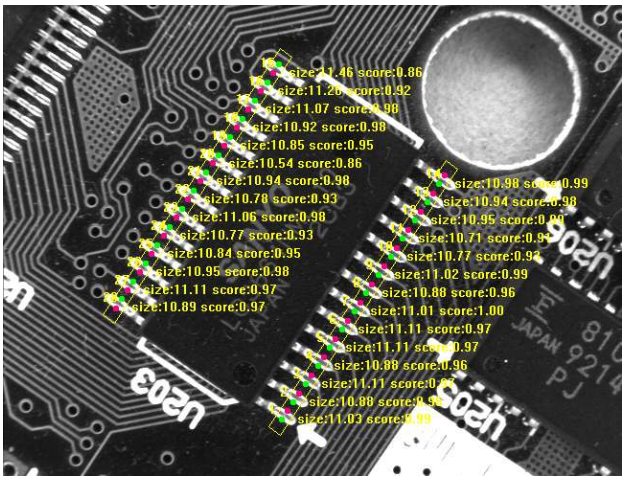
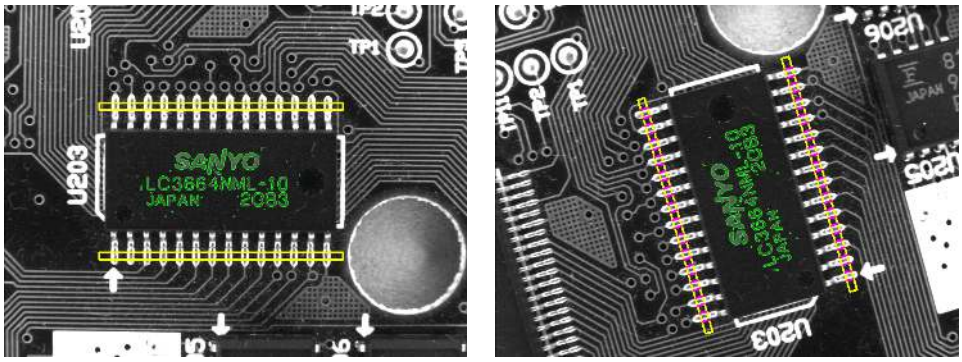


Figure 5.4: Measuring the width and distance of the leads.

The task of this example is to measure the positions of the leads of a chip (see figure 5.5). Because the chip can appear at varying positions and angles, the regions of interest used for the measurement must be aligned.



a) b)

Figure 5.5: (a) Model image with measurement ROIs; (b) measuring the leads in the aligned ROIs.

In this case, the alignment is achieved by searching for the print on the chip using shape-based matching (see Matching on page 89).

```
gen_rectangle1 (Rectangle, ModelRow1, ModelColumn1, ModelRow2, ModelColumn2)
reduce_domain (Image, Rectangle, ImageModel)
create_generic_shape_model (ModelID)
```

After the print has been found, the positions of the measurement ROIs are transformed relative to the position of the print.

```
find_generic_shape_model (SearchImage, ModelID, MatchResultID, \
                          NumMatchResult)
get_generic_shape_model_result_object (ShapeModelTrans, MatchResultID, \
                                       'all', 'contours')
get_generic_shape_model_result (MatchResultID, 'all', 'score', Score)
get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                HomMat2D)
affine_trans_pixel (HomMat2D, -RectDeltaRow, RectDeltaColumn, Rect1RowCheck, \
                    Rect1ColCheck)
affine_trans_pixel (HomMat2D, RectDeltaRow, RectDeltaColumn, Rect2RowCheck, \
                    Rect2ColCheck)
get_generic_shape_model_result (MatchResultID, 'best', 'angle', AngleCheck)
gen_rectangle2 (Rectangle1Check, Rect1RowCheck, Rect1ColCheck, AngleCheck, \
                RectLength1, RectLength2)
```

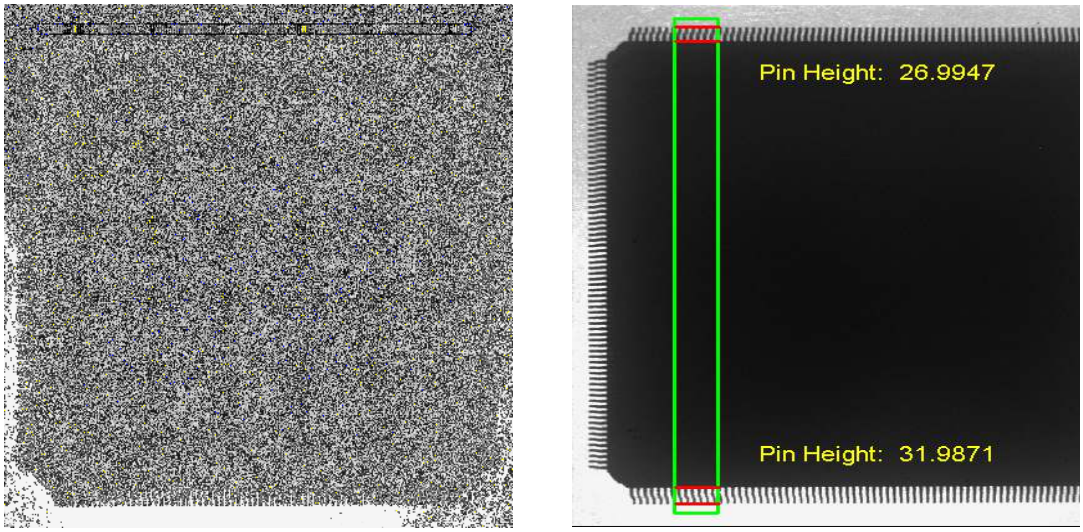
Then, the measure tools are created and the measurement is applied.

```
        gen_measure_rectangle2 (Rect1RowCheck, Rect1ColCheck, AngleCheck, \
                                RectLength1, RectLength2, Width, Height, \
                                'bilinear', MeasureHandle1)
        gen_measure_rectangle2 (Rect2RowCheck, Rect2ColCheck, AngleCheck, \
                                RectLength1, RectLength2, Width, Height, \
                                'bilinear', MeasureHandle2)
        measure_pairs (SearchImage, MeasureHandle1, 2, 40, 'positive', \
                       'all', RowEdgeFirst1, ColumnEdgeFirst1, \
                       AmplitudeFirst1, RowEdgeSecond1, ColumnEdgeSecond1, \
                       AmplitudeSecond1, IntraDistance1, InterDistance1)
        measure_pairs (SearchImage, MeasureHandle2, 2, 40, 'positive', \
                       'all', RowEdgeFirst2, ColumnEdgeFirst2, \
                       AmplitudeFirst2, RowEdgeSecond2, ColumnEdgeSecond2, \
                       AmplitudeSecond2, IntraDistance2, InterDistance2)
```

5.3.5 Inspect IC

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/measure_pin.hdev`

The task of this example is to inspect major dimensions of an IC (see figure 5.6).



a) b)

Figure 5.6: Measuring the dimensions of leads: (a) width of the leads and distance between them; (b) length of the leads.

In the first step the extent of each lead and the distance between the leads is measured. For this, a rectangle that contains the leads is defined (see figure 5.6a), which is used to generate the measure object. This is used to extract pairs of straight edges that lie perpendicular to the major axis of the rectangle.

```
gen_measure_rectangle2 (Row, Column, Phi, Length1, Length2, Width, Height, \
                        'nearest_neighbor', MeasureHandle)
measure_pairs (Image, MeasureHandle, 1.5, 30, 'negative', 'all', \
               RowEdgeFirst, ColumnEdgeFirst, AmplitudeFirst, \
               RowEdgeSecond, ColumnEdgeSecond, AmplitudeSecond, PinWidth, \
               PinDistance)
```

From the extracted pairs of straight edges, the number of leads, their average width, and the average distance between them is derived.

```
numPins := |PinWidth|
avgPinWidth := sum(PinWidth) / |PinWidth|
avgPinDistance := sum(PinDistance) / |PinDistance|
```

The second part shows the power of the measure tool: The length of the leads is determined. This is possible although each lead has a width of only a few pixels. For this, a new measure object is generated based on a rectangle that contains the leads on two opposite sides of the IC (see figure 5.6b). The distance between the first and the second found edge is the length of the upper leads, and the distance between the third and the fourth edge is the length of the lower leads.

```
gen_measure_rectangle2 (Row, Column, Phi, Length1, Length2, Width, Height, \
                        'nearest_neighbor', MeasureHandle)
measure_pos (Image, MeasureHandle, 1.5, 30, 'all', 'all', RowEdge, \
             ColumnEdge, Amplitude, Distance)
```

5.4 Relation to Other Methods

5.4.1 Alternatives to 1D Measuring

Edge Extraction (Subpixel-Precise) (see description on page 63)

A very flexible way to measure parameters of edges is to extract the edge contour with `edges_sub_pix`. The advantage of this approach is that it can handle free-form shapes. Furthermore, it allows to determine attributes like the edge direction for each edge point.

5.5 Tips & Tricks

5.5.1 Suppress Clutter or Noise

In many applications there is clutter or noise that must be suppressed. The measure operators offer multiple approaches to achieve this. The best one is to increase the threshold for the edge extraction to eliminate faint edges. In addition, the value for the smoothing parameter can be increased to smooth irrelevant edges away.

When grouping edges to pairs, noise edges can lead to an incorrect grouping if they are in the vicinity of the “real” edge and have the same polarity. In such a case you can suppress the noise edges by selecting only the strongest edges of a sequence of consecutive rising and falling edges.

5.5.2 Reuse Measure Object

Because the creation of a measure object needs some time, we recommend to reuse them if possible. If no alignment is needed, the measure object can, for example, be created offline and reused for each image. If the alignment involves only a translation, `translate_measure` can be used to correct the position.

5.5.3 Use an Absolute Gray Value Threshold

As an alternative to edge extraction, the measurements can be performed based on an absolute gray value threshold by using the operator `measure_thresh`. Here, all positions where the gray value crosses the given threshold are selected.

5.6 Advanced Topics

5.6.1 Fuzzy Measuring

In case there are extra edges that do not belong to the measurement, HALCON offers an extended version of measuring: fuzzy measuring. This tool allows to define so-called fuzzy rules, which describe the features of good edges. Possible features are, e.g., the position, the distance, the gray values, or the amplitude of edges. These functions are created with `create_funct_1d_pairs` and passed to the tool with `set_fuzzy_measure`. Based on these rules, the tool will select the most appropriate edges.

The advantage of this approach is the flexibility to deal with extra edges even if a very low minimum threshold or smoothing is used. An example for this approach is the example program `fuzzy_measure_pin.hdev` on page 49.

Please refer to the Solution Guide III-A, chapter 4 on page 27, for more information.

5.6.2 Evaluation of Gray Values

To have full control over the evaluation of the gray values along the measurement line or arc, you can use `measure_projection`. The operator returns the projected gray values as an array of numbers, which can then be further processed with HALCON operators for tuple or function processing (see the chapters "Tuple" and "Tools ▷ Function" in the Reference Manual). Please refer to the Solution Guide III-A, section 3.4 on page 19, for more information.

Chapter 6

Edge Extraction (Pixel-Precise)

The traditional way of finding edges, i.e., dark / light transitions in an image, is to apply an edge filter. These filters have the effect to find pixels at the border between light and dark areas. In mathematical terms this means that these filters determine the image gradient. This image gradient is typically returned as the edge amplitude and/or the edge direction. By selecting all pixels with a high edge amplitude, contours between areas can be extracted.

Another approach to find edges is to use a deep learning model trained to find edges. This method offers the advantage that its results can be further improved by retraining the model. Such a retraining makes it even possible to find specific edges on a custom task.

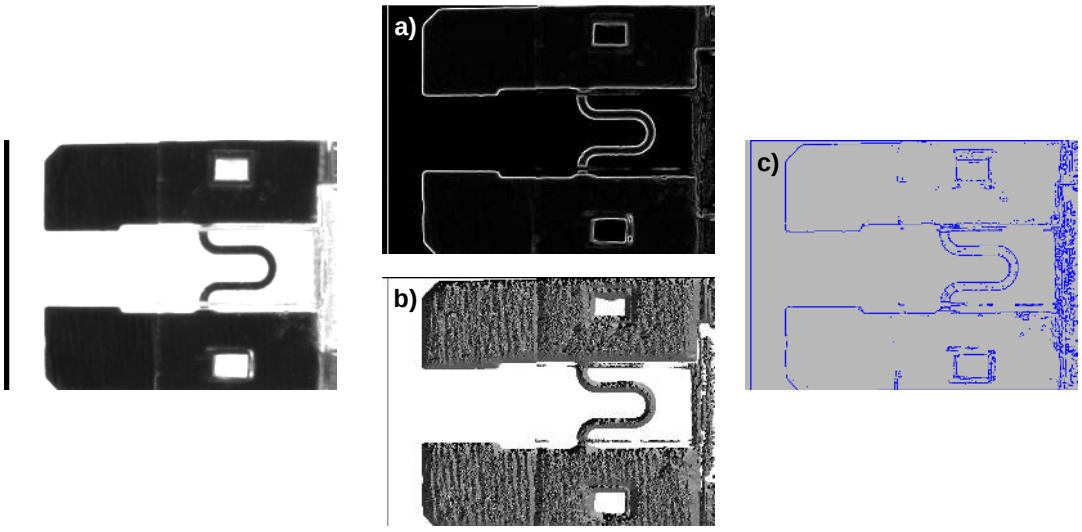


Figure 6.1: Result of applying an edge filter: (a) amplitude, (b) direction, (c) extracted edges.

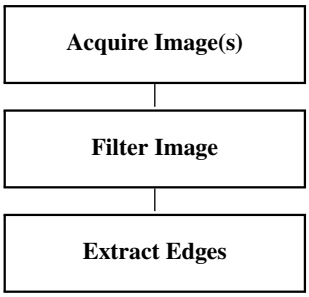
Please note that HALCON also provides operators for subpixel-precise edge and line extraction (see the description of this method on page 63) and for successive post-processing and feature extraction.

6.1 Edge Extraction Using Edge Filters

HALCON offers standard edge filters like the Sobel, Roberts, Robinson, or Frei filters. Besides these, post-processing operators like hysteresis thresholding or non-maximum suppression are provided. In addition, filters that determine the edge amplitude and edge direction accurately are provided. This enables you to apply the filters in a flexible manner.

6.1.1 Basic Concept

Using edge filters typically consists of three basic steps:



6.1.1.1 Acquire Image(s)

First, an image is acquired.

For detailed information see the description of this method on page 21.

6.1.1.2 Filter Image

On the input image, an edge filter is applied. This operation results in one or two images. The basic result is the edge amplitude, which is typically stored as a byte image, with the gray value of each pixel representing the local edge amplitude. Optionally, the direction of the edges is returned. These values are stored in a so-called direction image, with the values 0...179 representing the angle in degrees divided by two.

6.1.1.3 Extract Edges

The result of applying the edge filter is an image containing the edge amplitudes. From this image, the edges are extracted by selecting the pixels with a given minimum edge amplitude using a threshold operator. The resulting edges are typically broader than one pixel and therefore have to be thinned. For this step, various methods are available.

6.1.2 A First Example

The following program shows an example for the basic concept of edge filters. As an edge filter, `sobel_amp` is applied with the mode `'thin_sum_abs'` to get thin edges together with a 3x3 filter mask. Then, the operator `threshold` is used to extract all pixels with an edge amplitude higher than 20. The resulting region contains some areas where the edge is wider than one pixel. Therefore, the operator `skeleton` is applied to thin all edges completely. The result is depicted in figure 6.1c on page 55.

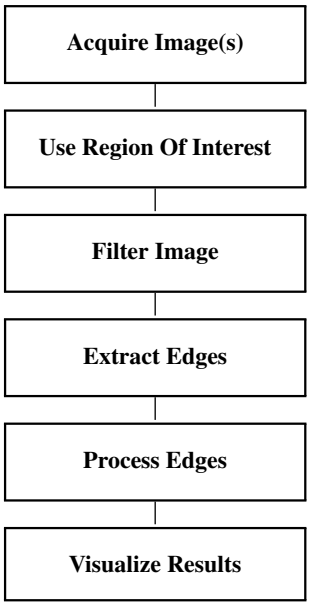
```
read_image (Image, 'fuse')
sobel_amp (Image, EdgeAmplitude, 'thin_sum_abs', 3)
threshold (EdgeAmplitude, Region, 20, 255)
skeleton (Region, Skeleton)
```

6.1.3 Extended Concept

6.1.3.1 Use Region Of Interest

Edge extraction can be sped up by using a region of interest. The more the region in which edge filtering is performed can be restricted, the faster and more robust the extraction will be.

For detailed information see the description of this method on page 25.



6.1.3.2 Filter Image

HALCON offers a wide range of edge filters. One of the most popular filters is the Sobel filter. This is the best of the old-fashioned filters. It combines speed with a reasonable quality. The corresponding operators are called `sobel_amp` and `sobel_dir`.

In contrast, `edges_image` provides the state of the art of edge filters. This operator is actually more than just a filter. It includes a thinning of the edges using a non-maximum suppression and a hysteresis threshold for the selection of significant edge points. It also returns the edge direction and the edge amplitude very accurately, which is not the case with the Sobel filter. This operator is recommended if higher quality is more important than a longer execution time. If the images are not noisy or blurred, you can even combine accuracy and speed by using the mode `'sobel_fast'` inside `edges_image`. The corresponding operator to find edges in multi-channel images, e.g., a color image, is `edges_color`.

6.1.3.3 Extract Edges

The easiest way to extract the edges from the edge amplitude image is to apply `threshold` to select pixels with a high edge amplitude. The result of this step is a region that contains all edge points. With `skeleton`, these edges can be thinned to a width of one pixel. As an advanced version for `threshold`, `hysteresis_threshold` can be used to eliminate insignificant edges. A further advanced option is to call the operator `nonmax_suppression_dir` before `skeleton`, which in difficult cases may result in more accurate edges. Note that in order to use this operator you must have computed the edge direction image.

In contrast, the advanced filter `edges_image` already includes the non-maximum suppression and the hysteresis threshold. Therefore, in this case a simple `threshold` suffices to extract edges that are one pixel wide.

If only the edge points as a region are needed, the operator `inspect_shape_model` can be used. Here, all steps including edge filtering, non-maximum suppression, and hysteresis thresholding are performed in one step with high efficiency.

6.1.3.4 Process Edges

If you want to extract the coordinates of edge segments, `split_skeleton_lines` is the right choice. This operator must be called for each connected component (result of `connection`) and returns all the control points of the line segments. As an alternative, a Hough transform can be used to obtain the line segments. Here, the operators `hough_lines_dir` and `hough_lines` are available. You can also convert the edge region into XLD contours by using, e.g., the operator `gen_contours_skeleton_xld`. The advantage of this approach is the extended set of operators offered for XLD contour processing on page 79, e.g., for contour segmentation, feature extraction, or approximation.

You can extract the regions enclosed by the edges easily using background_seg. If regions merge because of gaps in the edges, the operators close_edges or close_edges_length can be used in advance to close the gaps before regions are extracted. As an alternative, morphological operators like opening_circle can be applied to the output regions of background_seg. In general, all operators described for the method Process Regions on page 36 can be applied here as well.

6.1.3.5 Visualize Results

Finally, you might want to display the images, the edges (regions), and the line segments.

For detailed information see the description of this method on page 223.

6.1.4 Programming Examples

This section gives a brief introduction to using HALCON for edge filtering and edge extraction.

6.1.4.1 Aerial Image Interpretation

Example: %HALCONEXAMPLES%/solution_guide/basics/edge_segments.hdev

Figure 6.2 shows an image taken from an airplane. The task is to extract the edges of roads and buildings as a basis for the image interpretation.

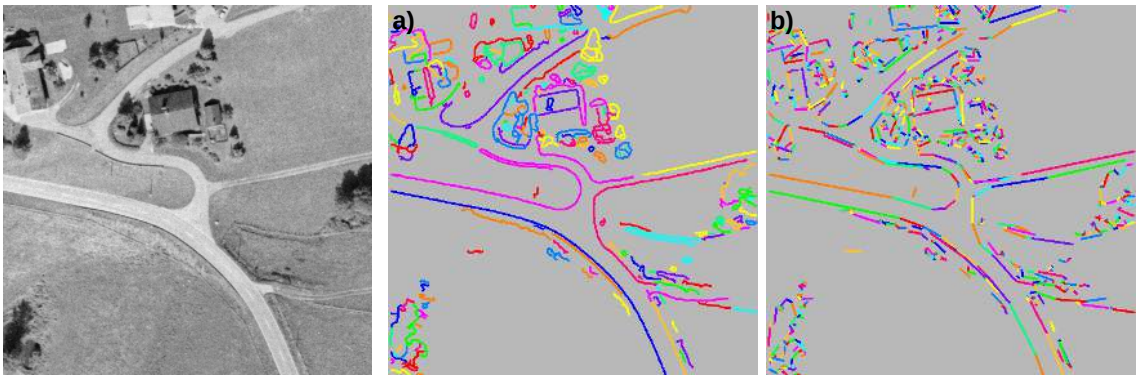


Figure 6.2: (a) Extracting edges and (b) approximating them by segments.

The extraction of edges is very simple and reliable when using the operator edges_image. This operator returns both the edge amplitude and the edge direction. Here, the parameters are selected such that a non-maximum suppression (parameter value 'nms') and a hysteresis threshold (threshold values 20 and 40) are performed. The non-maximum suppression has the effect that only pixels in the center of the edge are returned, together with the corresponding values for the amplitude and the direction. All other pixels are set to zero. Therefore, a threshold with the minimum amplitude of 1 is sufficient here. As a preparation for the next step, the edge contour regions are split up into their connected components.

```
read_image (Image, 'mreut')
edges_image (Image, ImaAmp, ImaDir, 'lanser2', 0.5, 'nms', 20, 40)
threshold (ImaAmp, Region, 1, 255)
connection (Region, ConnectedRegions)
```

The rest of the example program converts the region data into numeric values. To be more precise: the edges are approximated by individual line segments. This is performed by calling split_skeleton_lines for each connected component. The result of this call are four tuples that contain the start and the end coordinates of the line segments. For display purposes, each of these line segments is converted into an XLD contour.

```
count_obj (ConnectedRegions, Number)
gen_empty_obj (XLDContours)
for i := 1 to Number by 1
    select_obj (ConnectedRegions, SingleEdgeObject, i)
    split_skeleton_lines (SingleEdgeObject, 2, BeginRow, BeginCol, EndRow, \
                          EndCol)
    for k := 0 to |BeginRow| - 1 by 1
        gen_contour_polygon_xld (Contour, [BeginRow[k],EndRow[k]], \
                                 [BeginCol[k],EndCol[k]])
        concat_obj (XLDContours, Contour, XLDContours)
    endfor
endfor
dev_display (XLDContours)
```

6.1.4.2 Segmenting a Color Image

Example: %HALCONEXAMPLES%/hdevelop/Filters/Edges/edges_color.hdev

The task of this example is to segment the color image depicted in figure 6.3.

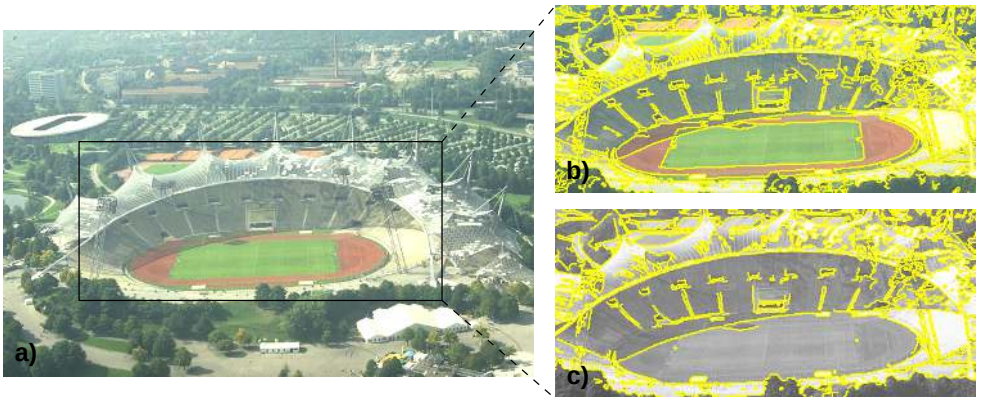


Figure 6.3: (a) Original image; (b) extracted color edges, overlaid on the color image; (c) extracted gray value edges, overlaid on the gray value image.

The example demonstrates the possibilities of a multi-channel edge filter. First, the gray value image is derived from the color information to show that some object borders can no longer be seen. For example, the (green) soccer field cannot be distinguished from the surrounding (red) track.

```
read_image (Image, 'olympic_stadium')
rgb1_to_gray (Image, GrayImage)
```

The color edge filter is applied and the edge amplitude is displayed. If you compare this to the filter result of the gray image the difference can be easily seen.

```
edges_color (Image, ImaAmp, ImaDir, 'canny', 1, 'none', -1, -1)
edges_image (GrayImage, ImaAmpGray, ImaDirGray, 'canny', 1, 'none', -1, -1)
```

Finally, the edge segments are extracted for both the color and the gray image and overlaid on the original image.

```
edges_color (Image, ImaAmpHyst, ImaDirHyst, 'canny', 1, 'nms', 20, 40)
threshold (ImaAmpHyst, RegionColor, 1, 255)
skeleton (RegionColor, EdgesColor)
dev_display (Image)
dev_display (EdgesColor)
stop ()
edges_image (GrayImage, ImaAmpGrayHyst, ImaDirGrayHyst, 'canny', 1, 'nms', \
             20, 40)
threshold (ImaAmpGrayHyst, RegionGray, 1, 255)
skeleton (RegionGray, EdgesGray)
dev_display (GrayImage)
dev_display (EdgesGray)
```

6.1.5 Relation to Other Methods

6.1.5.1 Alternatives to Edge Extraction Using Edge Filters

Blob Analysis (see description on page 33)
As an alternative to edge extraction, blob analysis can be used. This approach provides many methods from simple thresholding to region growing and watershed methods.

6.1.6 Tips & Tricks

6.1.6.1 Speed Up

Many online applications require maximum speed. Because of its flexibility, HALCON offers many ways to achieve this goal. Here the most common ones are listed.

- Regions of interest are the standard way to reduce the processing to only those areas where objects must be inspected. This can be achieved using pre-defined regions but also by an online generation of the regions of interest that depends on other objects found in the image.
- If high speed is important, the operators `sobel_amp` and `inspect_shape_model` are the preferred choice.
- By default, HALCON initializes new images. Using `set_system('init_new_image', 'false')`, this behavior can be changed to save execution time.

6.2 Deep-Learning-Based Edge Extraction

Deep-learning-based edge extraction is a special use case of deep-learning-based semantic segmentation. Thus, the model is a segmentation model designed and trained to extract edges. It comes with the advantages and drawbacks of this approach. This said, the model allows wide variations in the application images without need for setting changes. Further it can be retrained on custom applications just as any other segmentation model, shown in Specific Edges (see section 6.2.2.2). As for every deep learning approach, the accuracy of the extracted edges cannot be guaranteed. However, commonly, the results are satisfying for applications where a high precision is not required.

6.2.1 Concept

Deep-learning-based edge extraction is used the same way as other deep learning semantic segmentation models. Thus, the requirements and workflow are as described in "Deep Learning ▷ Semantic Segmentation".

HALCON delivers a specific model designed and pretrained to extract edges. Such an application is shown in Example: All Edges (see section 6.2.2.1).
But it is not guaranteed the model handles all kinds of specific problems as, e.g., low contrast and high noise images, since the model is pretrained to generalize well. In such cases the model performance can be (often significantly) improved by a retraining on application-specific images. An example is illustrated in Example: Specific Edges (see section 6.2.2.2).

6.2.2 Programming Examples

This section shows examples using deep-learning-based edge extraction in HALCON.

6.2.2.1 Example: All Edges

Example: `segment_edges_deep_learning.hdev`
in `%HALCONEXAMPLES%/hdevelop/Deep-Learning/Segmentation/`

This example demonstrates the deep-learning-based edge extractor on low contrast and high noise images. The task is to find the edges of the crosses. The result is depicted in figure 6.4.

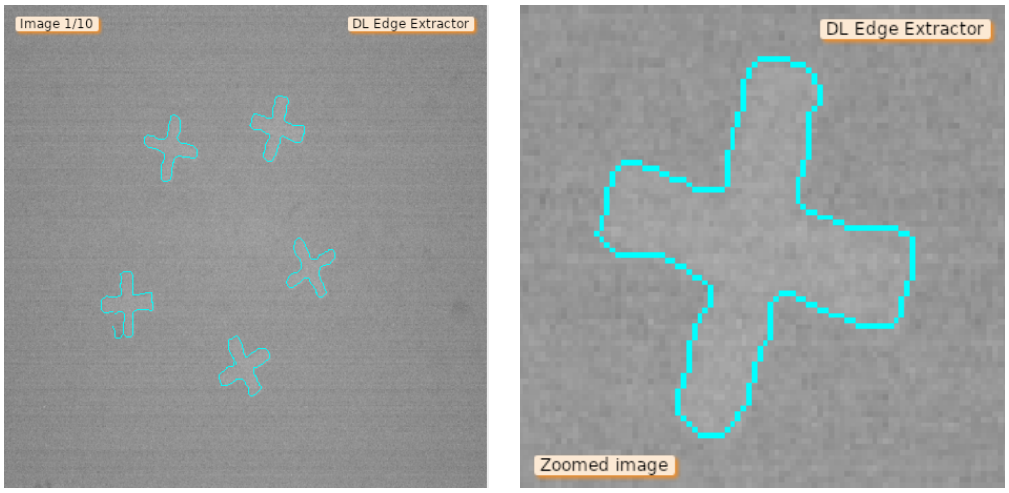


Figure 6.4: Left: The image with the inferred edges (cyan).
Right: A zoomed part to make the cross and its edges better visible.

The edge extraction is done using a segmentation model `pretrained_dl_edge_extractor.hdl`. For this task, we look for all edges and not specific ones. So it is generally not necessary to retrain the model and the task corresponds to the part `Inference on new images` in the deep learning workflow of "Deep Learning ▷ Semantic Segmentation".

Like every deep-learning-based segmentation model, every pixel of the input image gets a class assigned: edge or background. Additionally, every assignment is done with a certain confidence. So taking all pixels assigned as edge with a certain confidence results in edge regions. From this region the center can be extracted, e.g., using `skeleton`. This center marks the edges visible in figure 6.4.

6.2.2.2 Example: Specific Edges

Example: `segment_edges_deep_learning_with_retraining.hdev`
in `%HALCONEXAMPLES%/hdevelop/Deep-Learning/Segmentation/`

This example demonstrates the deep-learning-based edge extractor on fabric images. The task is to find a specific edge in the pattern. The result is depicted in figure 6.5.

The edge extraction is done using a segmentation model `pretrained_dl_edge_extractor.hdl`. We are not interested in any edge but only in specific ones. So we have to teach the network how to find the specific edges while ignoring unwanted ones. This is done by retraining the model. The workflow is described in the part `Training of the model` of "Deep Learning ▷ Semantic Segmentation". Note, the images have been labeled and preprocessed before.

The model returns the results in the same way as in segment_edges_deep_learning (see section 6.2.2.1), so the postprocessing is done in a similar way.

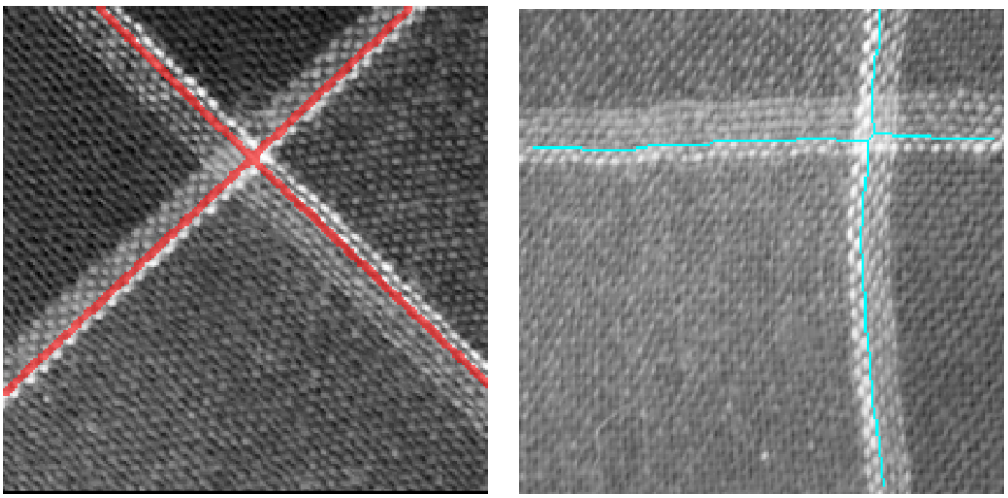


Figure 6.5: Left: The labeled image with the marked edge to find (red).Right: The inferred edge (cyan).

Chapter 7

Edge Extraction (Subpixel-Precise)

HALCON provides different ways to extract edges. There are the pixel-precise approaches of applying an edge filter or a deep-learning-based semantic segmentation model (see Edge Extraction (Pixel-Precise) on page 55). In addition, there are one-step operators that return subpixel-precise XLD contours. Besides this, not only edges but also lines can be extracted. This approach can also be applied to color images.

The advantage of this approach is its ease of use, because only a single operator call is needed. Furthermore, the accuracy and stability of the found contours is extremely high. Finally, HALCON offers a wide set of operators for the post-processing of the extracted contours, which includes, e.g., contour segmentation and fitting of circles, ellipses, and lines.

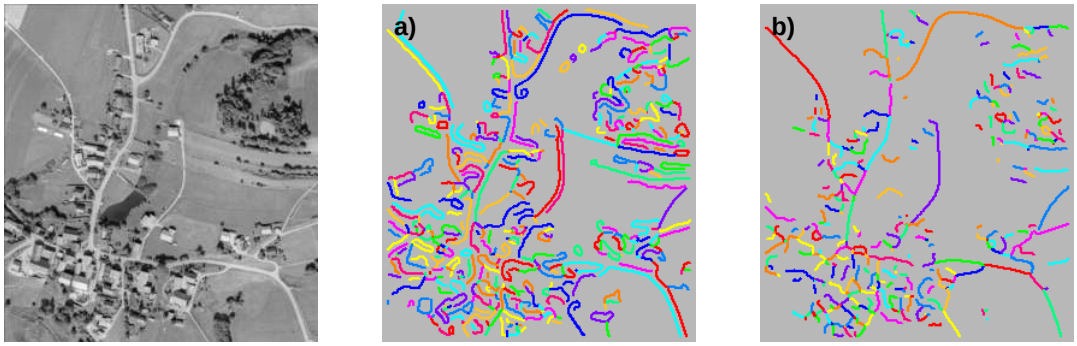


Figure 7.1: Result of contour extraction: (a) edge contours, (b) line contours.

This chapter covers only the extraction of contours. For information about processing them see the method Contour Processing on page 79.

7.1 Basic Concept

Extracting contours can easily be performed in a single step. Normally, no other operation is required.

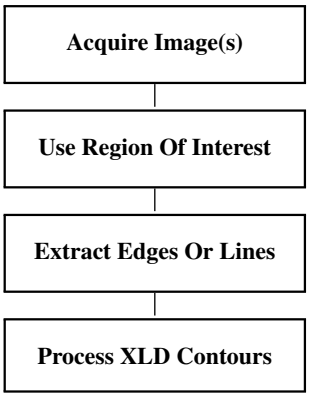
7.1.1 Acquire Image(s)

First, an image is acquired as input for the process.

For detailed information see the description of this method on page 21.

7.1.2 Extract Edges Or Lines

HALCON offers various operators for the subpixel-accurate extraction of contours. The standard operator is based on the first derivative. It takes the image as input and returns the XLD contours. When using the second derivatives,



first a Laplace operator must be executed before the contours along the zero crossings can be extracted. Besides the gray-value-based methods, HALCON provides the latest technology for the extraction of color edges.

Besides the extraction of edges, HALCON provides operators for the extraction of lines. In other systems lines are also called ridges. In contrast to edges, a line consists of two gray value transitions. Thus, a line can be considered as two parallel edges.

7.1.3 A First Example

The following program explains the basic concept of edge extraction. The only operator needed to extract edge contours is `edges_sub_pix`. It has the image as input and returns the XLD contours. Here, the filter `'lanser2'` is selected with a medium-sized smoothing mask. The low value for the parameter `Low` ensures that contours are tracked even along low-contrast parts. To show that the result consists of multiple contours, the 12-color mode for visualization is selected. The result is depicted in figure 7.1b on page 63.

```
read_image (Image, 'mreut4_3')
edges_sub_pix (Image, Edges, 'lanser2', 0.5, 8, 50)
dev_set_colored (12)
dev_clear_window ()
dev_display (Edges)
```

7.2 Extended Concept

In addition to the extraction, optional steps can be performed.

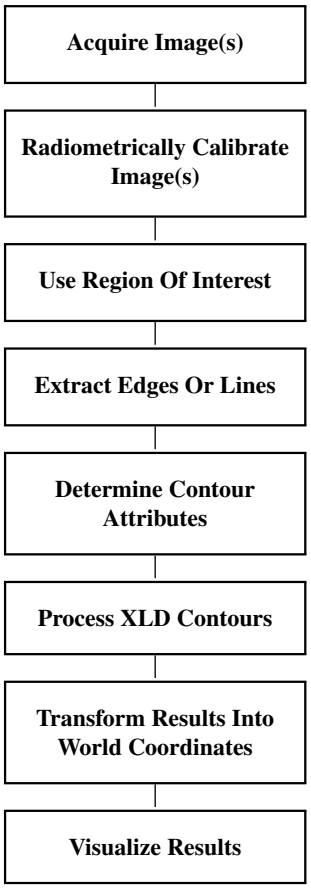
7.2.1 Radiometrically Calibrate Image(s)

To extract edges or lines with high accuracy, the camera should have a linear response function, i.e., the gray values in the images should depend linearly on the incoming energy. Since some cameras do not have a linear response function, HALCON provides the so-called radiometric calibration (gray value calibration): With the operator `radiometric_self_calibration` you can determine the inverse response function of the camera (offline) and then apply this function to the images using `lut_trans` before performing the edge and line extraction.

7.2.2 Use Region Of Interest

Edge extraction can be sped up by using a region of interest. The more the region in which edges or lines are extracted can be restricted, the faster and more robust the extraction will be.

For detailed information see the description of this method on page 25.



7.2.3 Extract Edges Or Lines

The most often used operator for edge contour extraction is edges_sub_pix. You can select various filter methods by specifying the corresponding name with the parameter Filter. For standard applications, common values are, e.g., 'canny' (based on a Gaussian convolution) or 'lanser2'. The advantage of 'lanser2' is the recursive implementation which has no increase in execution time when using a large smoothing. As a fast version the parameter value 'sobel_fast' can be used, which is recommended as long as the image is not noisy or blurred.

The operator zero_crossing_sub_pix can be used in combination with a filter like derivate_gauss with parameter value 'laplace'. The Laplace operator is mainly applied in the medical area.

To extract edges in multi-channel images, e.g., in a color image, HALCON provides the operator edges_color_sub_pix . Similar to edges_sub_pix, the parameter value 'sobel_fast' is recommended for a fast edge extraction as long as the image is not noisy or blurred.

The most commonly used operator for line extraction is lines_gauss. Compared to lines_facet it is more robust and provides more flexibility. The width of lines that should be extracted is specified by the parameter Sigma: The wider the line, the larger the value must be chosen. For very wide lines we recommend to zoom down the image (zoom_image_factor) in order to reduce the overall execution time.

Like for edges, HALCON provides line extraction also for multi-channel images. The corresponding operator is lines_color.

7.2.4 Determine Contour Attributes

The edge and line extraction operators not only provide the XLD contours but also so-called attributes. Attributes are numerical values; they are associated either with each control point of the contour (called contour attribute) or with each contour as a whole (global contour attribute). The operators get_contour_attrib_xld and get_contour_global_attrib_xld enable you to access these values by specifying the attribute name.

The attribute values are returned as tuples of numbers. Typical attributes for edges are, e.g., the edge amplitude and direction. For lines a typical attribute is the line width. The available attributes can be queried for a given contour with `query_contour_attribs_xld` and `query_contour_global_attribs_xld`.

7.2.5 Process XLD Contours

Typically, the task is not finished by just extracting the contours and accessing the attributes. HALCON provides further processing like contour segmentation, feature extraction, or approximation.

For detailed information see the description of this method on page 79.

7.2.6 Transform Results Into World Coordinates

In many applications the coordinates of contours should be transformed into another coordinate system, e.g., into 3D world coordinates. After you have calibrated your vision system, you can easily perform the transformation with the operator `contour_to_world_plane_xld`. With this approach you can also eliminate lens distortions and perspective distortions.

This is described in detail in the Solution Guide III-C in section 3.3 on page 76.

7.2.7 Visualize Results

Finally, you might want to display the images and the contours.

For detailed information see the description of this method on page 223.

7.3 Programming Examples

This section gives a brief introduction to using HALCON for edge extraction.

7.3.1 Measuring the Diameter of Drilled Holes

Example: `%HALCONEXAMPLES%/solution_guide/basics/rim_simple.hdev`

Figure 7.2 shows an image of a car rim. The task is to measure the diameters of the drilled holes.

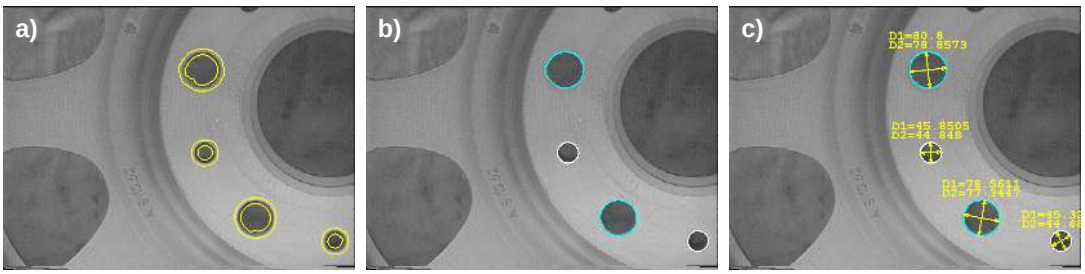


Figure 7.2: (a) automatically determined ROIs; (b) extracted edges; (c) computed ellipses and diameters.

First, a segmentation step is performed to roughly find the borders of the holes. The actual edge extraction is then performed only in these regions of interest (ROIs). This has two advantages: First, there are many edges in the image that are of no interest for the measurement. By restricting the processing to ROIs you can easily select the relevant objects. Secondly, the contour extraction is time-consuming. Thus, a reduced domain is an efficient way to speed-up the process.

Locating the holes is quite easy: First, all dark pixels are selected. After selecting all those connected components that are circular and have a certain size, only the holes remain. Finally, the regions of interest are obtained by accessing the borders of the holes and dilating them. The resulting ROIs are depicted in figure 7.2a.

```
threshold (Image, Dark, 0, 128)
connection (Dark, DarkRegions)
select_shape (DarkRegions, Circles, ['circularity', 'area'], 'and', [0.85, \
              50], [1.0, 99999])
boundary (Circles, RegionBorder, 'inner')
dilation_circle (RegionBorder, RegionDilation, 6.5)
union1 (RegionDilation, ROIEdges)
```

Calling reduce_domain changes the domain of the image to the prepared region of interest. Now, the edge extractor can be applied (see figure 7.2b).

```
reduce_domain (Image, ROIEdges, ImageROI)
edges_sub_pix (ImageROI, Edges, 'lanser2', 0.3, 10, 30)
```

The extracted contours are further processed to determine their diameter: With fit_ellipse_contour_xld, ellipses are fitted to the contours. In other words, those ellipses are determined that fit the extracted contours as closely as possible. The operator returns the parameters of the ellipses. With the operator gen_ellipse_contour_xld, the corresponding ellipses are created and displayed (compare figure 7.2b and figure 7.2c). Another option is to use the operator gen_circle_contour_xld.

```
fit_ellipse_contour_xld (Edges, 'ftukey', -1, 2, 0, 200, 3, 2, Row, Column, \
                         Phi, Ra, Rb, StartPhi, EndPhi, PointOrder)
NumHoles := |Ra|
gen_ellipse_contour_xld (ContEllipse, Row, Column, Phi, Ra, Rb, \
                         gen_tuple_const(NumHoles,0), \
                         gen_tuple_const(NumHoles,rad(360)), \
                         gen_tuple_const(NumHoles,'positive'), 1)
```

The diameters can easily be computed from the ellipse parameters and then be displayed in the image using write_string (see figure 7.2c).

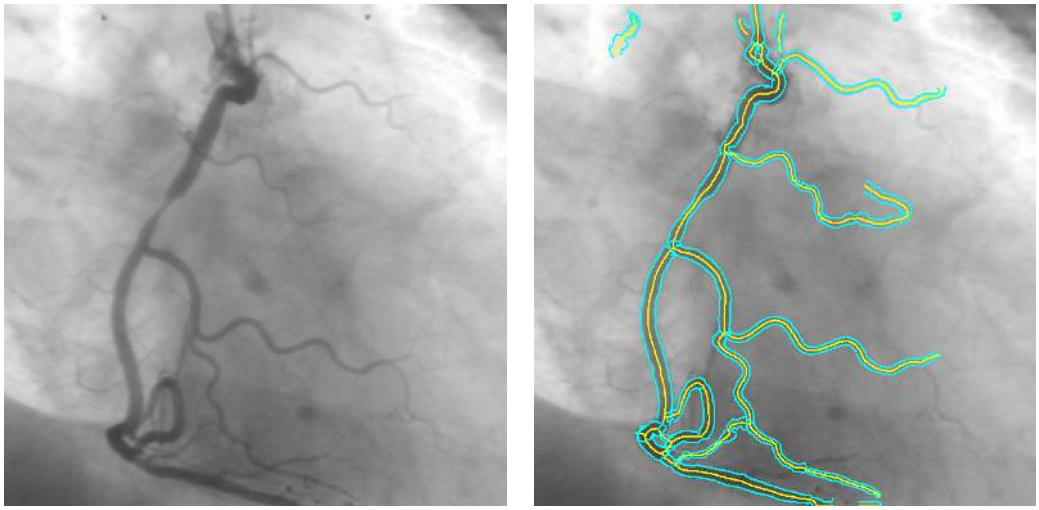
```
for i := 0 to NumHoles - 1 by 1
    write_string (WindowID, 'D1=' + 2 * Ra[i])
    write_string (WindowID, 'D2=' + 2 * Rb[i])
endfor
```

7.3.2 Angiography

Example: %HALCONEXAMPLES%/hdevelop/Filters/Lines/lines_gauss.hdev

The task of this example is to extract the blood vessels in the X-ray image of the heart depicted in figure 7.3. The vessels are emphasized by using a contrast medium. For the diagnosis it is important to extract the width of the vessels to determine locally narrowed parts (stenoses).

The vessels are extracted using lines_gauss. The result of this operator are the centers of the vessels in the form of XLD contours. Besides this, attributes are associated with the contour points, one of which is the local line width. This width is requested and displayed as contours.



a) b)

Figure 7.3: (a) X-ray image of the heart; (b) extracted blood vessels.

```
MaxLineWidth := 8
Contrast := 12
calculate_lines_gauss_parameters (MaxLineWidth, [Contrast,0], Sigma, Low, \
                                  High)
lines_gauss (Angio, Lines, Sigma, Low, High, 'dark', 'true', 'parabolic', \
             'true')
count_obj (Lines, Number)
for I := 1 to Number by 1
    select_obj (Lines, Line, I)
    get_contour_xld (Line, Row, Col)
    get_contour_attrib_xld (Line, 'angle', Angle)
    get_contour_attrib_xld (Line, 'width_left', WidthL)
    get_contour_attrib_xld (Line, 'width_right', WidthR)
    RowR := Row + cos(Angle) * WidthR * sqrt(0.75)
    ColR := Col + sin(Angle) * WidthR * sqrt(0.75)
    RowL := Row - cos(Angle) * WidthL * sqrt(0.75)
    ColL := Col - sin(Angle) * WidthL * sqrt(0.75)
    disp_polygon (WindowID, RowL, ColL)
    disp_polygon (WindowID, RowR, ColR)
endfor
```

7.4 Relation to Other Methods

7.4.1 Alternatives to Edge Extraction (Subpixel-Precise)

Subpixel Thresholding

Besides the subpixel-accurate edge and line extractors, HALCON provides a subpixel-accurate threshold operator called `threshold_sub_pix`. If the illumination conditions are stable, this can be a fast alternative.

Subpixel Point Extraction

In addition to the contour-based subpixel-accurate data, HALCON offers subpixel-accurate point operators for various applications. In the reference manual, these operators can be found in the chapter "Filters ▷ Points".

Chapter 8

Deflectometry

The aim of deflectometry is to detect defects, like scratches, bumps, and dents on specular or partially specular surfaces. For this, a known structured light pattern is generated and reflected by the surface to be inspected. In case of a non-planar surface the reflected pattern will be distorted like shown in figure 8.1.

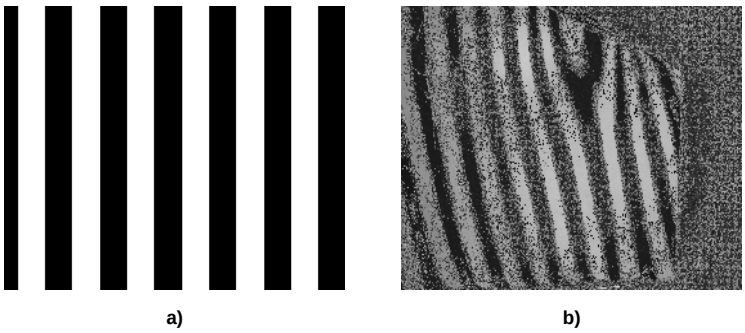


Figure 8.1: Pattern images: (a) known pattern, (b) reflected pattern.

Some example deflectometry setups to acquire images of reflected patterns are illustrated in figure 8.2. Structured light patterns are projected on the surface to be analyzed by a monitor (figure 8.2 a and b) or a projector and a diffusor plate (figure 8.2 c). Note, in parts not specific to a pattern source both, monitor and projector with diffusor plate, are meant even if only one of them is mentioned. The specular or partially specular surface reflects the pattern, which is then captured by a camera.
In case of using a projector as a pattern source, a diffuse emitting surface is required for generating a flat grid. It is not suitable to light with the projector directly onto the surface as the projector has a divergent beam. Compared to monitors, projectors will potentially offer a higher brightness, which is a key advantage for the inspection of partially specular surfaces. Furthermore, depending on the technology they rely on, some projectors provide a high frame rate, thus enabling the implementation of short inspection cycles.

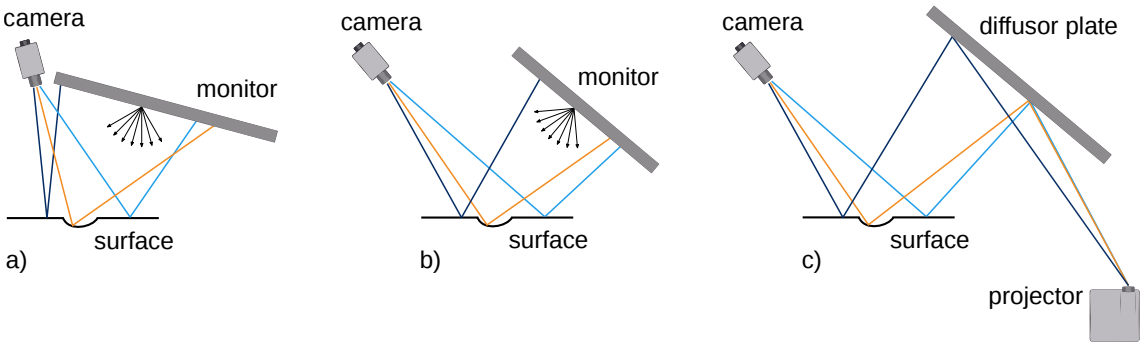
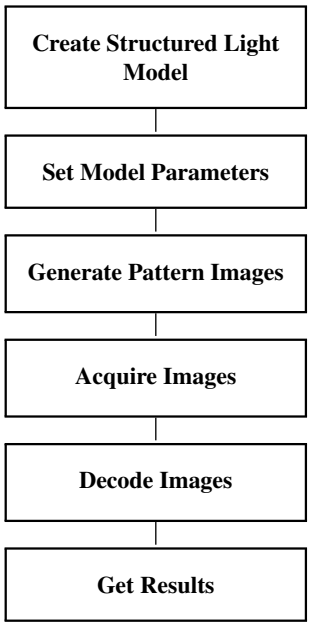


Figure 8.2: Sketch of different deflectometric setups.

Surface defects like dents deflect the light in such a way that monitor pixels which are far from each other appear closer to each other in the reflected pattern and thus, in the captured camera image. Each camera pixel 'sees' a specific monitor pixel. The purpose of deflectometry is to detect which camera pixel 'sees' which monitor pixel and to analyze these correspondences to find defects.

8.1 Basic Concept

The detection of defects with deflectometry consists basically of the following parts:



8.1.1 Create Structured Light Model

Firstly, you have to create the structured light model by the operator `create_structured_light_model`. The model stores information that is successively added during the following steps.

8.1.2 Set Model Parameters

Next, the operator `set_structured_light_model_param` is used to set the parameters of the structured light model. Note that these basic settings influence the results of the subsequent operators.

8.1.3 Generate Pattern Images

The pattern images are generated with `gen_structured_light_pattern`. There are four types of pattern images: Normalization images, Gray code images, phase shift images, and single stripe images. Information on the different types of pattern images can be found in the HALCON Operator Reference.

8.1.4 Acquire Images

The reflected images are acquired for each pattern image that is displayed on the monitor and reflected by the object toward the camera. For detailed information on image acquisition see the description of this method on page 21. Recommendations on the image acquisition are listed in section 8.3.1.

8.1.5 Decode Images

Next, the acquired images need to be decoded with decode_structured_light_pattern to calculate which monitor pixel corresponds to which camera pixel. The resulting correspondence images are stored in the structured light model.

8.1.6 Get Results

Using the operator get_structured_light_object you can get a defect image in which high gray values usually indicate the presence of defects. Those defects can then be segmented, e.g., by thresholding using threshold. We recommend to test some samples of the measuring object with defects and at least some without defects to find a good threshold value. Note that the computation of the defect image depends on the model parameter 'derivative_sigma', which controls the internal smoothing. This parameter must be set before calling get_structured_light_object with 'defect_image'.

8.2 Programming Examples

This section gives a brief introduction to using HALCON for deflectometry.

8.2.1 Inspecting a Tap Collar

Example:

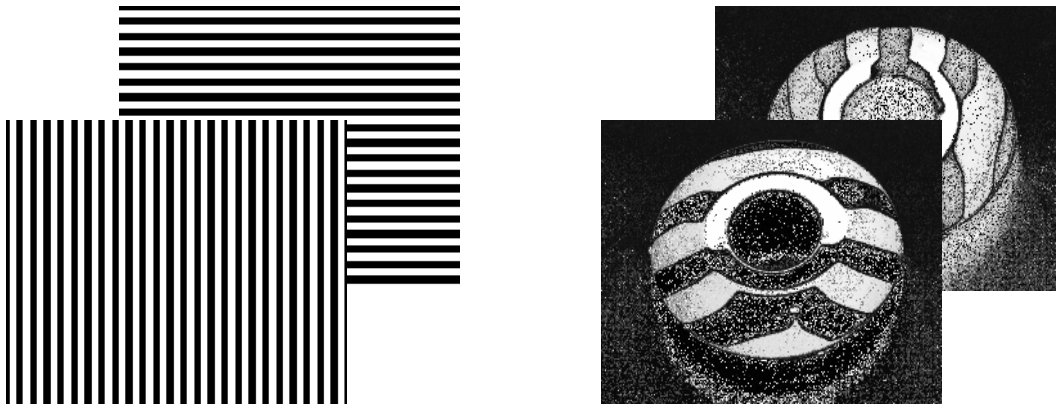
%HALCONEXAMPLES%/hdevelop/Inspection/Structured-Light/structured_light_tap_collar.hdev

The task of this example is to inspect the surface of a specular tap collar using deflectometry. For this, a structured light model of type 'deflectometry' is created (create_structured_light_model) and the parameters necessary for the generation of the pattern images are defined. For example, the width 'PatternWidth' and height 'PatternHeight' of the pattern images should be equivalent to the width or height of the monitor used to display the pattern images. Furthermore, the pattern type as well as the pattern orientation need to be defined in advance. The stripe width of the projected pattern is set with 'min_stripe_width'. A smaller stripe width usually leads to higher accuracy, but it should be chosen such that the smallest stripes are visible with distinct edges.

```
create_structured_light_model ('deflectometry', StructuredLightModel)
PatternWidth := 1600
PatternHeight := 1200
PatternType := 'gray_code_and_phase_shift'
PatternOrientation := 'both'
MinStripeWidth := 8
Normalization := 'global'
set_structured_light_model_param (StructuredLightModel, ['pattern_width', \
                                  'pattern_height', 'pattern_type', \
                                  'pattern_orientation', \
                                  'min_stripe_width', 'normalization'], \
                                  [PatternWidth,PatternHeight,PatternType, \
                                 PatternOrientation,MinStripeWidth, \
                                 Normalization])
```

Next, the pattern images are generated with gen_structured_light_pattern. Afterwards, the pattern images are displayed one by one on the monitor. For each pattern image a camera image of the corresponding reflected pattern is acquired with grab_image (see figure 8.3).

```
gen_structured_light_pattern (PatternImages, StructuredLightModel)
DirName := 'structured_light/tap_collar/'
open_framegrabber ('File', -1, -1, -1, -1, -1, -1, 'default', -1, 'default', \
                   -1, 'default', DirName, 'default', -1, -1, AcqHandle)
count_obj (PatternImages, NumberOfPatternImages)
gen_empty_obj (CameraImages)
for Index := 1 to NumberOfPatternImages by 1
    select_obj (PatternImages, DisplayImage, Index)
    dev_display (DisplayImage)
    grab_image (CameraImage, AcqHandle)
    concat_obj (CameraImages, CameraImage, CameraImages)
endfor
close_framegrabber (AcqHandle)
```



a) b)

Figure 8.3: Gray code patterns: (a) generated Gray code patterns, (b) reflected Gray code patterns. In this example the horizontal stripes appear vertical in the camera image, because the camera was rotated by 90 degrees prior to the image acquisition. The same applies for the vertical stripes.

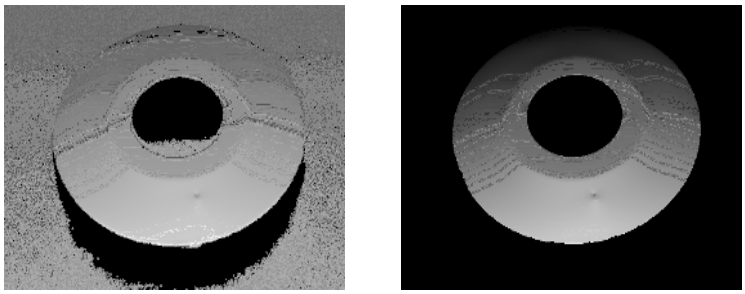
The next subtask is to decode the camera images. For that, the parameter `'min_gray_difference'` is set to the expected gray value difference between white and dark stripes in the camera images. This parameter is necessary to segment the object to be inspected, i.e., the image part in which the reflection of the pattern is visible. Depending on the value of `'min_gray_difference'`, the reflective object is more or less distinguishable from the background (see figure 8.4). We recommend to use sample images to determine the appropriate value for the minimal gray value difference.

The camera images can now be decoded using the operator `decode_structured_light_pattern`. The resulting correspondence images are extracted with `get_structured_light_object`.

```
MinGrayDifference := 130
set_structured_light_model_param (StructuredLightModel, \
                                  'min_gray_difference', MinGrayDifference)
decode_structured_light_pattern (CameraImages, StructuredLightModel)
get_structured_light_object (CorrespondenceImages, StructuredLightModel, \
                             'correspondence_image')
```

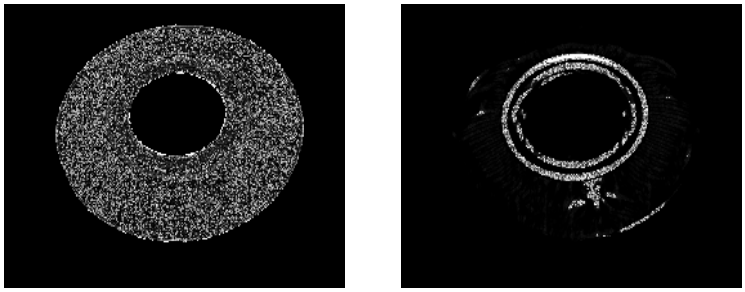
Finally, you can get a defect image with different values of `'derivative_sigma'`. The parameter `'derivative_sigma'` controls the smoothing of the defect image and thus makes different kinds of defects more or less visible like shown in figure 8.5. Note that the defect image is of type `'real'` and may have a very large gray value range. For a visual inspection it may be helpful to scale the image using, e.g., `scale_image_range`.

```
Sigma := 1
set_structured_light_model_param (StructuredLightModel, 'derivative_sigma', \
                                  Sigma)
get_structured_light_object (DefectImage1, StructuredLightModel, \
                             'defect_image')
scale_image_range (DefectImage1, ScaledDefectImage1, 0, 4)
```



a) b)

Figure 8.4: Correspondence images: a) `min_gray_difference` set to 15, b) `min_gray_difference` set to 130.



a) b)

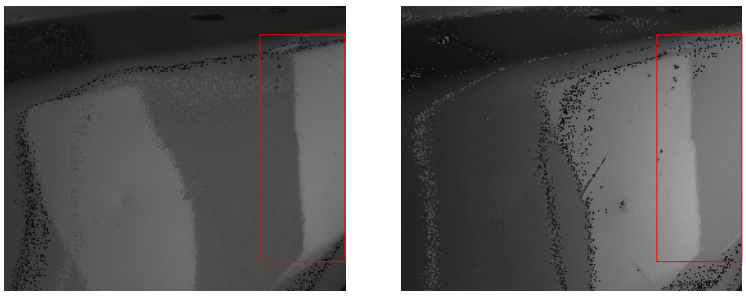
Figure 8.5: Defect images: a) `'derivative_sigma'` set to 1, b) `'derivative_sigma'` set to 5.

8.2.2 Inspecting a Partially Specular Surface

Example:

`%HALCONEXAMPLES%/hdevelop/Inspection/Structured-Light/structured_light_partically_specular.hdev`

If a partially specular surface is inspected, the diffusive part of the reflection may lead to phenomenon shown in figure 8.6.



a) b)

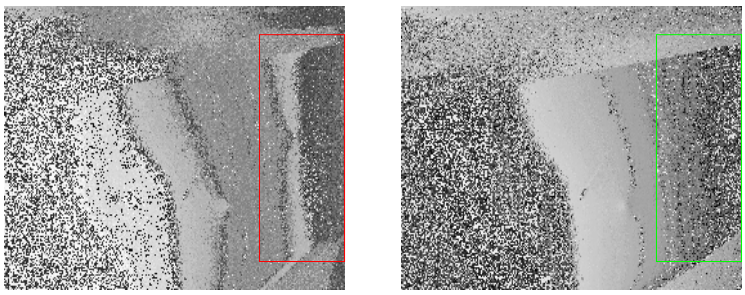
Figure 8.6: Images of a partially specular surface: a) bright stripe at the right edge, b) dark stripe at the right edge.

The stripe on the right is a bright stripe in image (a) and a dark stripe in image (b). But some gray values of the dark stripe are actually larger than the gray values of the bright stripe, though it does not seem so for the human observer. The algorithm will hence determine that these pixels are in a bright stripe in the second image (b) and in a dark stripe in the first image (a), which leads to a wrong decoding. To avoid a wrong decoding, i.e. to optimize the robustness in terms of varying surface reflectance, the `'pattern_type'` needs to be set to `'single_stripe'`:

```
create_structured_light_model ('deflectometry', StructuredLightModelSS)
PatternType := 'single_stripe'
SingleStripeWidth := 64
set_structured_light_model_param (StructuredLightModelSS, ['pattern_width', \
                                  'pattern_height', 'min_stripe_width', \
                                  'min_gray_difference', 'pattern_type', \
                                  'single_stripe_width', 'persistence'], \
                                  [PatternWidth,PatternHeight, \
                                 MinStripeWidth,MinGrayDifference, \
                                 PatternType,SingleStripeWidth, \
                                 Persistence])
```

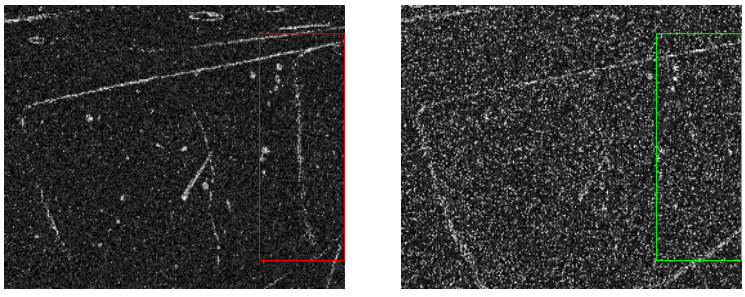
The subsequent steps (image acquisition, decoding of camera images, etc.) are analogous to the procedure for non-specular surfaces which is why those steps are not explained here. Please refer to section 8.2.1 for more information. Instead, the following example images demonstrate the differences of using the pattern type `'single_stripe'` compared to the pattern type `'gray_code_and_phase_shift'` in case of partially specular surfaces.

As can be seen in figure 8.7 a, the camera image is wrongly decoded if `pattern_type` is set to `'gray_code_and_phase_shift'`. Consequently, it is not possible to detect defects within the wrongly decoded region (figure 8.8 a). If, instead, the `pattern_type` is set to `'single_stripe'`, the camera image is decoded correctly (figure 8.7 b) and the defects are clearly visible (figure 8.8 b).



a) b)

Figure 8.7: Correspondence images: a) `pattern_type` set to `'gray_code_and_phase_shift'`, b) `pattern_type` set to `'single_stripe'`.



a) b)

Figure 8.8: Defect images: a) `pattern_type` set to `'gray_code_and_phase_shift'`, b) `pattern_type` set to `'single_stripe'`.

8.3 Tips & Tricks

8.3.1 Set Up the Measurement

If you want to achieve accurate results, please follow the recommendations given in this section:

- Arrange measuring object, camera, and pattern source such that the camera captures the reflection of the screen that is visible in the whole surface part under inspection.
  The angles of the camera and the monitor/diffusor plate should have similar absolute values. More precisely, the angles mentioned are the angle between the camera viewing direction and the normal of the flat surface (α) or the angle between the normal of the monitor/diffusor plate and the normal of the flat surface (β). See the illustration in figure 8.9. The sensitivity of the surface inspection can be improved if the mentioned angles are increased. Nevertheless, this is a theoretical consideration and from a practical point of view a modified setup may require a fairly larger monitor/diffusor plate.

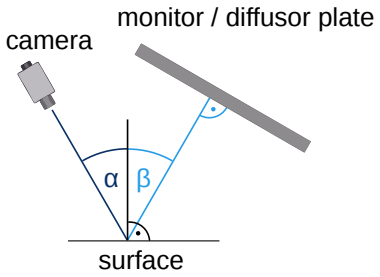


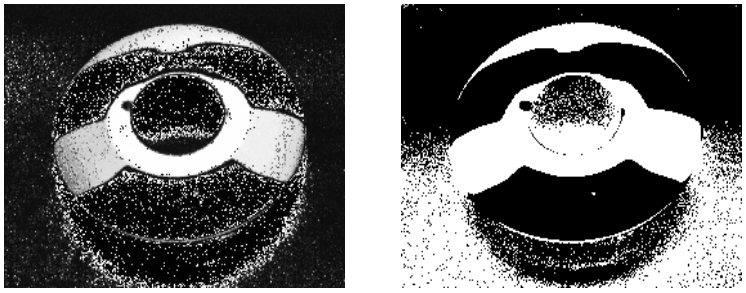
Figure 8.9: Sketch of the angles of the camera and the monitor/diffusor plate.

- The resolution of the projector should at least be the same as the resolution of the camera
- The needed area size of the projected pattern (monitor/diffusor plate) depends on the surface area and surface curvature to be inspected. The stronger the curvature the larger the projector needs to be.
- The gray value range of the captured camera image should be exploited as fully as possible. I.e., in the ideal case the gray values of the camera image would be near 0 if the screen is black while the gray values would be near 255 (in case of byte camera images) if the screen is white.
- For some objects (e.g., white glossy plastic) it is difficult to get a good contrast between the stripes and the background. In this case it helps to increase the gain value of the camera. Furthermore, when only a binary pattern (e.g., gray code) is used, one can consider setting the camera gamma value to a value larger 1 in order to increase the contrast.
- If you are using a color camera, choose the channel with the best contrast.
- The gray values of the camera image should depend nearly linearly on the gray values of the screen. This can be achieved with a radiometric calibration. See e.g., the HDevelop example program `%HALCONEXAMPLES%/hdevelop/Calibration/Self-Calibration/radiometric_self_calibration.hdev`.
- There should be no overexposed regions if a completely white image is displayed on the screen (at least in the image part where the measuring object is located).
- The camera should be focused on the object, but also the stripe patterns should be in focus. This means that the depth of field of the camera should be high enough. In order to optimize the available depth of field, consider positioning the screen closer to the object surface.
- As a rule of thumb concerning the minimal size of the defects in the camera image: As a minimum we recommend one size being in the order of magnitude of 10 pixels.
- The stripe width of the projected pattern (`'min_stripe_width'`) should be chosen such that the smallest stripes are visible with distinct edges in the camera image.
- The ambient light should be low, especially if objects are partially specular.
- The background behind the measuring object should be non-specular.
- The monitor or projector should be synchronized with the camera (see section 8.3.3.1).

8.3.2 Check the Decoding Results

Decoding of camera images relies upon being able to know whether a pixel is in a region where a light stripe is reflected or where a dark stripe is reflected. You can check whether the algorithm recognizes bright and dark regions correctly by inspecting the 'binarized image' obtained from `get_structured_light_object`. Note that for this the 'persistence' mode has to be enabled with `set_structured_light_model_param` before decoding.

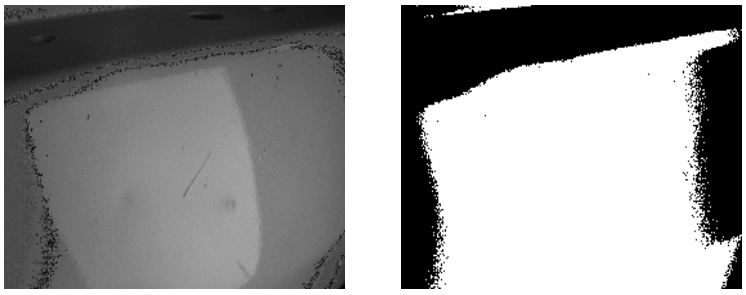
An exemplary camera image (a) compared to its binarized image (b) is shown in figure 8.10. The white or black regions in the binarized image indicate that the camera pixel observed reflections of white or black regions of the projected pattern. The binarized image below shows white and dark regions that have been recognized correctly.



a) b)

Figure 8.10: Images of a tap collar: a) camera image, b) binarized image.

For comparison, figure 8.11 shows a partially specular surface and its corresponding binarized image. Several pixels have been identified wrongly, i.e. some actually dark pixels have been identified as bright pixels. Therefore dark regions of the camera image (a) are bright regions in the binarized image (b) which is an indication for partially specular surfaces. To optimize the decoding results, we recommend the pattern type `'single_stripe'` in case of partially specular surfaces.



a) b)

Figure 8.11: Images of a partially specular surface: a) camera image, b) binarized image.

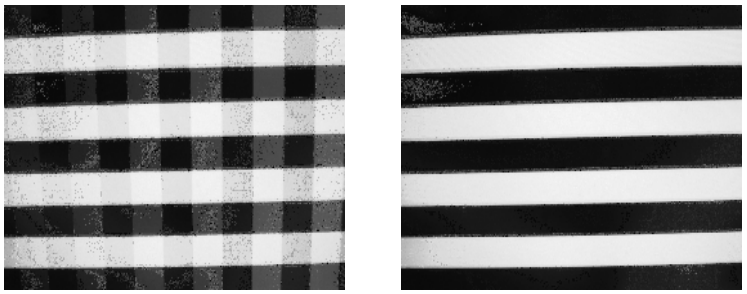
To help you deciding whether the bright or the dark regions were correctly encoded, you can use the procedure `structured_light_inspect_segmentation`. This procedure helps to validate the decoded bright or dark areas in the Gray code image in comparison to the bright or dark areas in the camera image. The procedure can thus be used to find a suitable value for the parameter `'min_gray_difference'`. Please open the HDevelop program `%HALCONEXAMPLES%/hdevelop/Inspection/Structured-Light/structured_light_partically_specular.hdev` for an example how to use this procedure.

8.3.3 Synchronize the Camera with the Pattern Source

The synchronization of the camera with the source generating the light pattern depends strongly on the source. The two main cases are explained in the following.

8.3.3.1 Synchronize the Camera with the Monitor

Each pattern image needs to be already fully displayed on the monitor when the exposure time for the corresponding camera image begins. Otherwise, (parts of) the previous pattern image may still be visible when a new camera image is acquired. An example is shown in figure 8.12.



a) b)

Figure 8.12: Gray code patterns: a) previously displayed stripes still visible, b) only the currently displayed pattern is visible.

A simple way to prevent overlapping patterns is to include a short waiting period after `disp_image` before `grab_image` is called. The correct waiting period depends on the used setup and needs to be found by trial and error.

For convenience, HALCON provides the procedure `structured_light_camera_screen_sync` that helps finding a suitable waiting period. To use this procedure, the camera should be positioned such that it sees a large part of the monitor where the pattern images are to be displayed. Alternatively, the camera can observe the reflection of the monitor via a planar mirror. The procedure alternately displays images with vertical and horizontal stripes and acquires the corresponding camera images. The waiting period between the call to `disp_image` and the call to `grab_image` can be set by the user. In the camera images there should be no visible overlap of consecutive pattern images as in figure 8.12 (a).

Furthermore, you should check that each camera image shows the correct pattern. In a badly configured setup, a camera image might be acquired while the previous pattern is still fully displayed. In that case, the n-th image would show the (n-1)-th pattern image. If any of the these problems occur, a longer waiting period should be tested. Otherwise, you can try whether a shorter waiting period is still sufficient.

8.3.3.2 Synchronize the Camera with the Projector

Projectors typically have a digital output for providing hardware triggers for cameras. As soon as an image is displayed the projector emits a digital signal which can be used for synchronize acquiring the image by the camera.

8.3.4 Speed Up the Acquisition Process

For a more time-efficient acquisition process, the exposure of each camera image should be triggered the moment the corresponding pattern image is fully visible on the screen. We summarize two possible hardware-based solutions:

Using a monitor: Position two photosensors on a small area of the monitor that is not reflected by the inspected object towards the camera. The sensors should detect whenever the pixel values at their position switch from black to white and trigger the camera at each detection.

Adapt the pattern images that are displayed on the monitor such that they alternate between white and black pixel values in the area where the sensors are positioned. Whenever the area of the first sensor is white, the area of the second sensor should be black and vice versa. This can be done using the operators `paint_region` or `overpaint_region`.

This way, whenever a new pattern image is displayed on the monitor, one of the sensors sends a trigger signal to the camera and the camera image is acquired at the right moment. As a monitor-refresh usually occurs from the top to the bottom of the monitor, it is recommended to position the sensors at the bottom of the display area. This should ensure that the sensor is not triggered before the pattern image is fully displayed.

Using a Projector: Industrial projectors have the advantage that the image projection can be realized faster compared to structured light projection by standard TFT monitors. One reason for that is the long fade-out characteristic of monitors. Another reason is that projectors typically have a digital output for providing hardware triggers for cameras.

Chapter 9

Contour Processing

One of HALCON's powerful tool sets are the subpixel-accurate contours. Contours belong to the data type XLD. These contours are typically the result of some kind of image processing and represent, e.g., the borders of objects. Figure 9.1a shows such edges overlaid on the original image; Figure 9.1b zooms into the rectangular area marked in Figure 9.1a and furthermore highlights the so-called control points of the contours with crosses. Here, you can clearly see the highly accurate positioning of the control points.

HALCON provides operators to perform advanced types of measurements with these contours. For example, the contours can be segmented into lines and circular or elliptic arcs (see Figure 9.1c). The parameters of these segments, e.g., their angle, center, or radius, can then be determined and used, e.g., in the context of a measuring task.

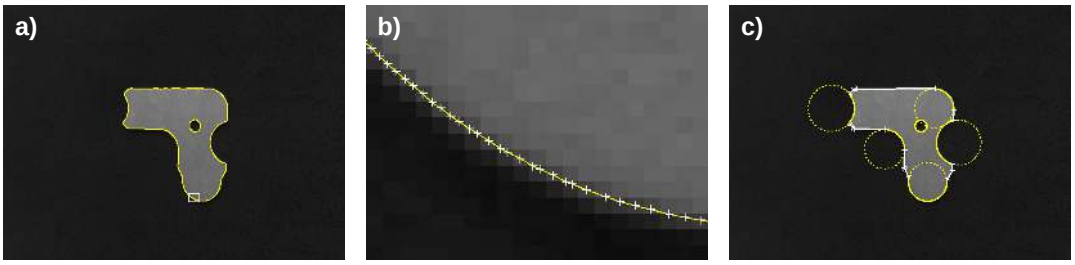


Figure 9.1: XLD contours: (a) edge contours, (b) zoom into rectangular area, (c) segmented lines and elliptic arcs.

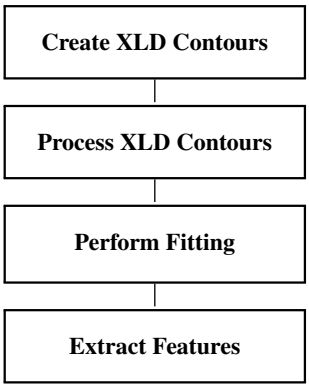
The advantage of contour processing is twofold: First, its high accuracy enables reliable measurements. Secondly, the extensive and flexible set of operators provided for this data type enables you to solve problems that cannot be solved with classical methods like 1D measuring. More detailed information about contour processing can be found in the Solution Guide III-B.

9.1 Basic Concept

The processing of contours consists of multiple steps that can be combined in a flexible way.

9.1.1 Create XLD Contours

The most common way to create XLD contours is to apply one of the subpixel-accurate extraction operators described for the method Extract Edges Or Lines on page 65. As an alternative, an edge filter with some post-processing can be used. The resulting regions are then converted to XLD contours. Please note that this approach is only pixel-accurate. For more information about this approach see the method Edge Extraction Using Edge Filters on page 55.



9.1.2 Process XLD Contours

Typically, only certain contours of an object are used for an inspection task. One possibility to restrict the extraction of contours to the desired ones is to use a well-fitting region of interest as, e.g., depicted in figure 9.2a: The rectangular ROI just covers the upper part of the blades. When applying an edge extractor, exactly one contour on each side of the objects is found.

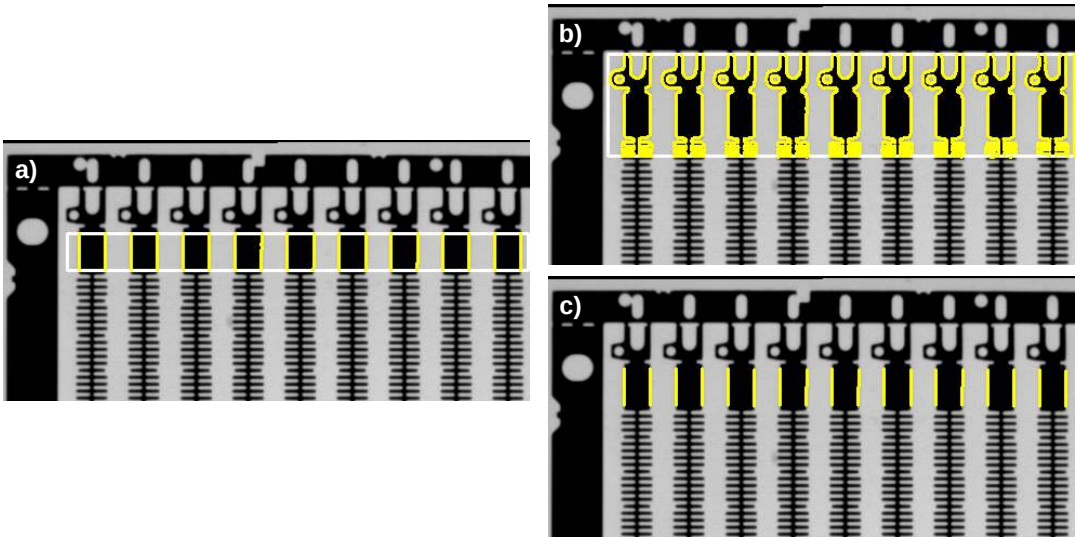


Figure 9.2: Selecting the desired contours: (a) exactly fitting ROI, (b) too many contours because of too large ROI, (c) result of post-processing the contours from (b).

In many cases, however, not only the desired contours are extracted. An example is depicted in figure 9.2b, where the ROI was chosen too large. Thus, the contours must be processed to obtain the desired parts of the contours. In the example, the contours are segmented into parts and only parallel segments with a certain length are selected (see the result in figure 9.2c).

Another reason for processing contours occurs if the extraction returns unwanted contours caused by noise or texture or if there are gaps between contours because of a low contrast or contour intersections.

9.1.3 Perform Fitting

Having obtained contour segments that represent a line, a rectangle, or a circular or elliptic arc, you can determine the corresponding parameters, e.g., the coordinates of the end points of a line or the center and radius of a circle, by calling one of the fitting operators. Their goal is to approximate the input contour as closely as possible to a line, rectangle, or a circular or elliptic arc. Because the used minimization algorithms are very advanced and all contour points are used for the process, the parameters can be calculated very reliably.

9.1.4 Extract Features

From both raw contours and processed contour parts features can be determined. Some of these consider the contour as a linear object. Others treat a contour as the outer boundary of an object. Obviously, the center of gravity makes sense only for a closed object, whereas the curvature is a feature of a linear object.

9.1.5 A First Example

The following program is an example for the basic concept of contour processing. It shows how short segments returned by the line extractor can be grouped to longer ones.

First, an image is acquired from file using `read_image`. The task is to extract the roads, which show up as thin bright lines in the image. For this the operator `lines_gauss` is used. When we look at the result of the contour extraction in figure 9.3a, we see that a lot of unwanted small segments are extracted. They can be suppressed easily by calling `select_contours_xld` with a minimum contour length. A further problem is that some roads are split into more than one segment. They can be combined with the operator `union_collinear_contours_xld`. Looking at the result in figure 9.3b, we see that many fragments have been combined along straight road parts. In curves this method fails because the orientation of the segments differs too much.

```
read_image (Image, 'mreut4_3')
lines_gauss (Image, Lines, 1.5, 2, 8, 'light', 'true', 'bar-shaped', 'true')
select_contours_xld (Lines, LongContours, 'contour_length', 15, 1000, 0, 0)
union_collinear_contours_xld (LongContours, UnionContours, 30, 2, 9, 0.7, \
                              'attr_keep')
```

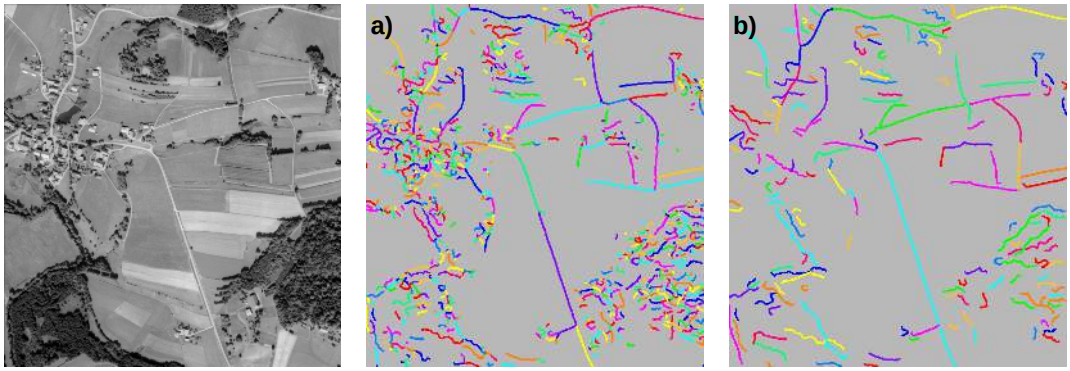


Figure 9.3: Processing XLD contours, (a) extracted contours, (b) processed contours.

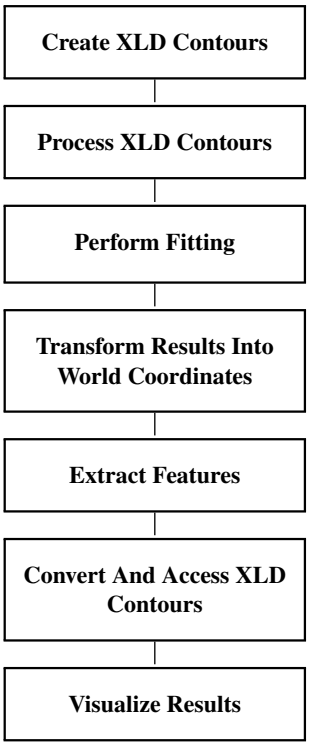
9.2 Extended Concept

In addition to the standard contour processing, HALCON provides other tools that can be added. Typical examples for these are camera calibration, geometric transformations, or type conversions. With these, the contour methods can be integrated into the overall vision task.

9.2.1 Create XLD Contours

The standard method to create contours is to call a contour extraction operator. Contour extraction for edges is performed with `edges_sub_pix`, `edges_color_sub_pix`, or `zero_crossing_sub_pix`. Lines are extracted using `lines_gauss`, `lines_facet`, or `lines_color`. For subpixel blob analysis the operator `threshold_sub_pix` can be used. These operators are described in more detail with the method Edge Extraction (Subpixel-Precise) on page 63.

If pixel-accuracy is sufficient, you can use an edge filter (like `sobel_amp` or `edges_image`) or a line filter (like `bandpass_image`) followed by thresholding and thinning. The resulting elongated regions are then converted into



XLD contours with the operator gen_contours_skeleton_xld. For more information on this approach see the method Edge Extraction Using Edge Filters on page 55.

Contours can also be synthesized from different sources, e.g., CAD data, user interaction, or measuring. Having obtained the coordinates of the control points from such a source, the operators gen_contour_polygon_xld and gen_contour_polygon_rounded_xld convert them to XLD contours. You can also draw XLD contours interactively with the operators draw_xld and draw_xld_mod.

Finally, the border of regions can be converted into XLD contours. The corresponding operator is called gen_contour_region_xld.

9.2.2 Process XLD Contours

The first method to segment contours is to call segment_contours_xld. This operator offers various modes: Splitting into line segments, linear and circular segments, or linear and elliptic segments. The individual contour segments can then be selected with select_obj and passed to one of the fitting operators described with the step Perform Fitting on page 80. Whether a contour segment represents a line, a circular, or an elliptic arc can be queried via the global contour attribute 'cont_approx' using the operator get_contour_global_attrib_xld.

If only line segments are needed, you can use the combination of gen_polygons_xld followed by split_contours_xld. The behavior is similar to using segment_contours_xld. The main difference is the possible postprocessing: When applying gen_polygons_xld, a so-called XLD polygon is generated. This is a different data type, which represents the initial step for grouping of segments to parallel lines.

An important step during contour processing is the suppression of irrelevant contours. This can be accomplished with the operator select_shape_xld, which provides almost 30 different shape features. By specifying the desired minimum and maximum value and possibly combining multiple features, contours can be selected very flexibly. As an alternative, you can use the operator select_contours_xld, which offers typical features of linear structures. Finally, the operator select_xld_point can be used in combination with mouse functions to interactively select contours.

If there are gaps within a contour, the pieces are treated as separate objects, which makes further processing and feature extraction difficult. You can merge linear segments with the operators union_collinear_contours_xld or union_straight_contours_xld. Additionally, you can also merge adjacent contours (union_adjacent_contours_xld), contours that lie on the same circle (union_cocircular_contours_xld),

or contours that are cotangential (union_cotangential_contours_xld). To handle contours with a complex shape you can first segment them into linear, circular, or elliptic segments (see above).

HALCON also provides an operator for general shape modifications: shape_trans_xld. With this operator you can, e.g., transform the contour into its surrounding circle, convex hull, or surrounding rectangle. Further, for closed contours or polygons, set theoretical operations can be applied to combine contours. For example, with intersection_closed_contours_xld you can intersect the regions that are enclosed by the closed contours, with difference_closed_contours_xld you can calculate the difference between the enclosed regions, or with union2_closed_contours_xld you can merge the enclosed regions.

9.2.3 Perform Fitting

With the operator fit_line_contour_xld you can determine the parameters of a line segment. The operator provides different optimization methods, most of which are suppressing outliers. It returns the coordinates of the start and the end point of the fitted line segment and the normal form of the line. To visualize the results, you can use the operator gen_contour_polygon_xld.

To fit a rectangle into a contour, the operator fit_rectangle2_contour_xld can be used. It provides various optimization methods as well. The returned parameters comprise mainly the center position, the extent, and the orientation of the rectangle. To generate the obtained rectangle for a visualization, you can use the operator gen_rectangle2_contour_xld.

For the fitting of circular and elliptic segments the operators fit_circle_contour_xld and fit_ellipse_contour_xld are available. They also provide various optimization methods. For a circular segment the center and the radius are returned together with the angle range of the visible part. In addition, a second radius and the orientation of the main axis are returned for elliptic segments. To visualize the results of both operators, you can use either the operator gen_ellipse_contour_xld or the operator gen_circle_contour_xld.

9.2.4 Transform Results Into World Coordinates

As a post-processing step, it may be necessary to correct the contours, e.g., to remove lens distortions, or to transform the contours into a 3D world coordinate system in order to extract dimensional features in world units. Such a transformation is based on calibrating the camera. After the calibration, you simply call the operator contour_to_world_plane_xld to transform the contours.

How to transform contours into world coordinates is described in detail in the Solution Guide III-C in section 3.3 on page 76.

9.2.5 Extract Features

HALCON offers various operators to access the feature values. Commonly used shape features are calculated by area_center_xld, compactness_xld, convexity_xld, eccentricity_xld, diameter_xld, and orientation_xld. The hulls of the contours can be determined with smallest_circle_xld or smallest_rectangle2_xld. Features based on geometric moments are calculated, e.g., by moments_xld.

9.2.6 Convert And Access XLD Contours

Finally, it might be necessary to access the raw data of the contours or to convert contours into another data type, e.g., into a region.

You can access the coordinates of the control points with the operator get_contour_xld. It returns the row and column coordinates of all control points of a contour in two tuples of floating-point values. In case of a contour array (tuple), you must loop over all the contours and select each one using select_obj.

To convert contours to regions, simply call the operator gen_region_contour_xld. The operator paint_xld paints the contour with anti-aliasing into an image.

The operators for edge and line extraction not only return the XLD contours but also so-called attributes. Attributes are numerical values; they are associated either with each control point (called contour attribute)

or with each contour as a whole (global contour attribute). The operators get_contour_attrib_xld and get_contour_global_attrib_xld enable you to access these values by specifying the attribute name. More information on this topic can be found in the description of the step Determine Contour Attributes on page 65.

9.2.7 Visualize Results

Finally, you might want to display the images and the contours.

For detailed information see the description of this method on page 223.

9.3 Programming Examples

This section gives a brief introduction to using HALCON for contour processing.

9.3.1 Measuring Lines and Arcs

Example: %HALCONEXAMPLES%/solution_guide/basics/measure_metal_part.hdev

The first example shows how to segment a contour into lines and (circular) arcs and how to determine the corresponding parameters. Figure 9.4 shows the final result of the fitted primitives overlaid on the input image.

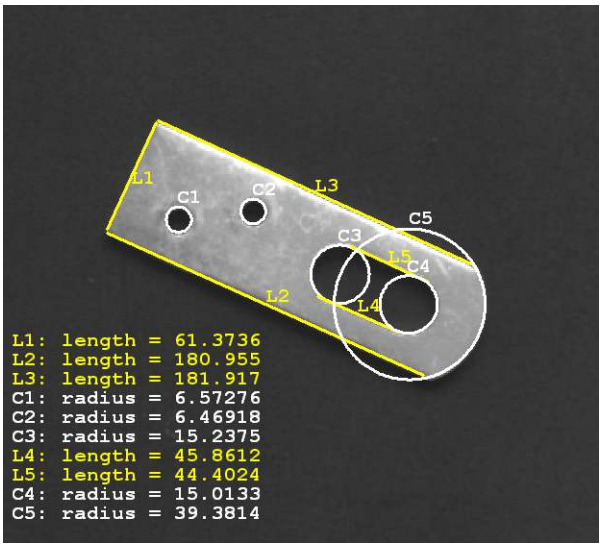


Figure 9.4: Fitted lines and circles.

As the initial step, the contours of the metal part are extracted using the operator edges_sub_pix. The resulting contours are segmented into lines and circular arcs and sorted according to the position of their upper left corner.

```
edges_sub_pix (Image, Edges, 'lanser2', 0.5, 40, 90)
segment_contours_xld (Edges, ContoursSplit, 'lines_circles', 6, 4, 4)
sort_contours_xld (ContoursSplit, SortedContours, 'upper_left', 'true', \
                   'column')
```

Then, lines and circles are fitted to the extracted segments. As already noted, the individual segments must be accessed inside a loop. For this, first their total number is determined with count_obj. Inside the loop, the individual segments are selected with the operator select_obj. Then, their type (line or circular arc) is determined by accessing a global attribute with get_contour_global_attrib_xld. Depending on the result, either a circle or a line is fitted. For display purposes, circles and lines are created using the determined parameters. Furthermore, the length of the lines is computed with the operator distance_pp.

```
count_obj (SortedContours, NumSegments)
for i := 1 to NumSegments by 1
    select_obj (SortedContours, SingleSegment, i)
    get_contour_global_attrib_xld (SingleSegment, 'cont_approx', Attrib)
    if (Attrib == 1)
        fit_circle_contour_xld (SingleSegment, 'atukey', -1, 2, 0, 5, 2, \
                                Row, Column, Radius, StartPhi, EndPhi, \
                                PointOrder)
        gen_ellipse_contour_xld (ContEllipse, Row, Column, 0, Radius, \
                                 Radius, 0, rad(360), 'positive', 1.0)
    else
        fit_line_contour_xld (SingleSegment, 'tukey', -1, 0, 5, 2, RowBegin, \
                              ColBegin, RowEnd, ColEnd, Nr, Nc, Dist)
        gen_contour_polygon_xld (Line, [RowBegin,RowEnd], [ColBegin,ColEnd])
        distance_pp (RowBegin, ColBegin, RowEnd, ColEnd, Length)
    endif
endfor
```


9.3.2 Close Gaps in a Contour

Example: %HALCONEXAMPLES%/solution_guide/basics/close_contour_gaps.hdev

The second example demonstrates how to close gaps in an object contour (see figure 9.5). The example is based on synthetic data. Instead of using a real image, a light gray square on a dark gray background is generated and a part of its boundary is blurred.

```
gen_rectangle1 (Rectangle, 30, 20, 100, 100)
region_to_bin (Rectangle, BinImage, 130, 100, 120, 130)
rectangle1_domain (BinImage, ImageReduced, 20, 48, 40, 52)
mean_image (ImageReduced, SmoothedImage, 15, 15)
paint_gray (SmoothedImage, BinImage, Image)
```

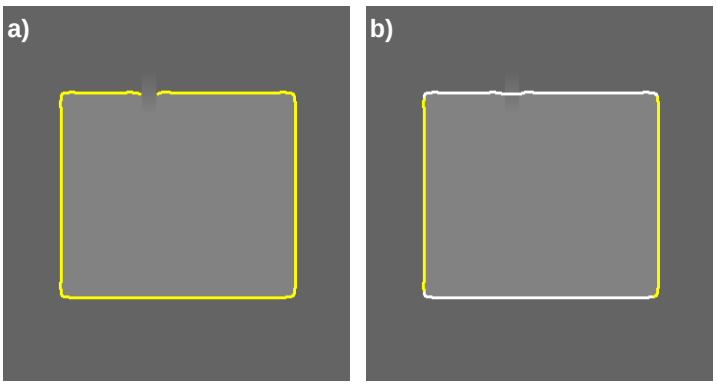


Figure 9.5: Original edges and result of grouping process.

The extraction of contours with edges_sub_pix thus results in an interrupted boundary (see figure 9.5a). Note that the edge extraction is restricted to the inner part of the image, otherwise edges would be extracted at the boundary of the image.

```
rectangle1_domain (BinImage, ImageReduced, 20, 48, 40, 52)
edges_sub_pix (ImageReduced, Edges, 'lanser2', 1.1, 22, 30)
```

A suitable operator for closing gaps in linear segments is union_collinear_contours_xld. Before we can apply this operator, some pre-processing is necessary: First, the contours are split into linear segments using segment_contours_xld. Then, regress_contours_xld is called to determine the regression parameters for each segment. These parameters are stored with each contour and could be accessed with get_regress_params_xld. Finally, union_collinear_contours_xld is called. Its result is depicted in figure 9.5b.

```
segment_contours_xld (Edges, LineSegments, 'lines', 5, 4, 2)
regress_contours_xld (LineSegments, RegressContours, 'no', 1)
union_collinear_contours_xld (RegressContours, UnionContours, 10, 1, 2, 0.1, \
                              'attr_keep')
```

9.3.3 Calculate Pointwise Distance between XLD Contours

Example: %HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/inspect_frame_width.hdev

The task of this example is to show how to inspect the camera frame of a phone housing by extracting edges and calculating the pointwise distance between them (see figure 9.6).

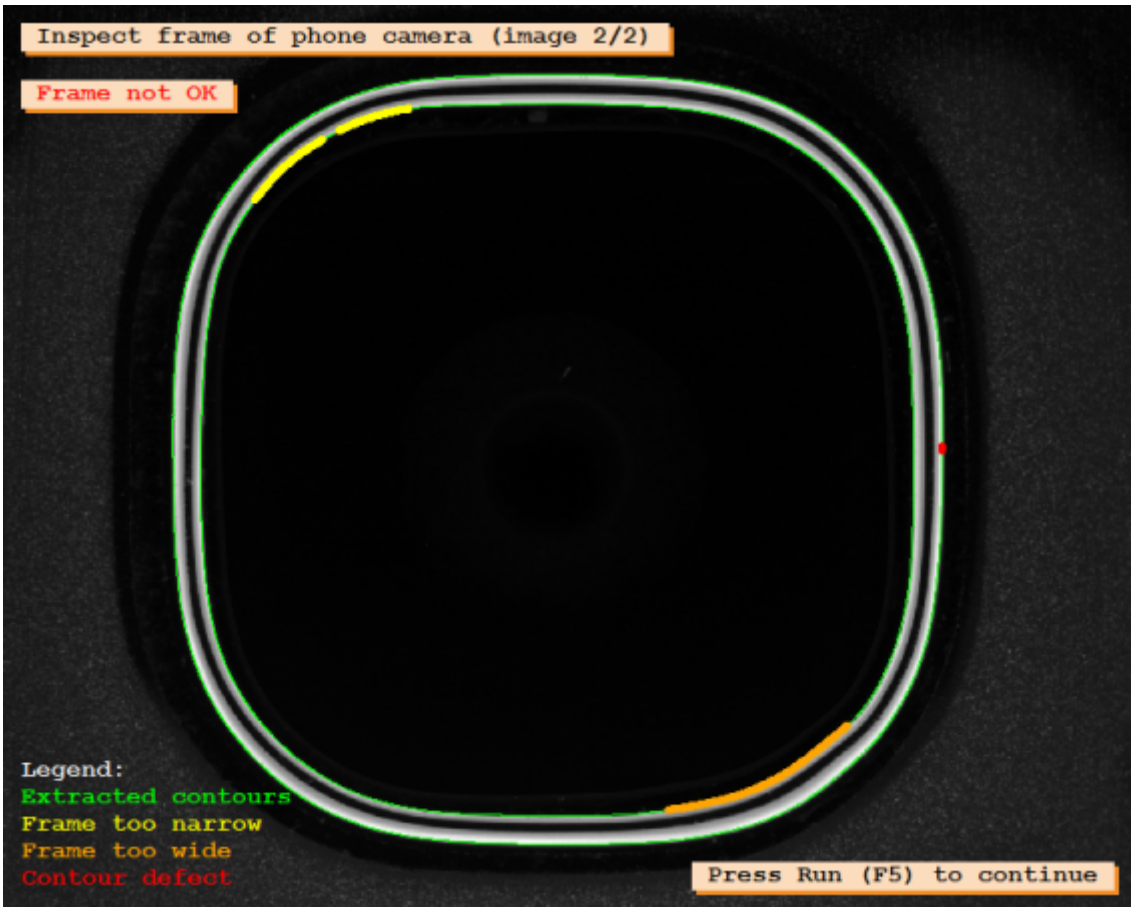


Figure 9.6: Image of the camera frame of a phone together with the extracted and classified edges.

The program starts by first extracting edges within a reduced domain of the image.

```
threshold (Image, Region, 100, 255)
dilation_rectangle1 (Region, RegionDilation, 15, 15)
reduce_domain (Image, RegionDilation, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'canny', 0.7, 10, 60)
```

The contours are unified based on the distance between their end points. From the unified contours, only sufficiently long contours are selected for further processing. These contours correspond to the inner border of the camera frame and to the outer border of the camera frame, respectively.

```
union_adjacent_contours_xld (Edges, UnionContours, 7, 7, 'attr_keep')
select_shape_xld (UnionContours, SelectedContours, 'contlength', 'and', \
                  700, 99999)
```

Then, the distance between the inner contour and the outer contour is calculated pointwise and the inner contour is segmented based on the distance to the outer contour.

```
distance_contours_xld (InnerContour, OuterContour, \
                       OuterContourWithWidth, 'point_to_segment')
* Get the contour parts that lie outside of the tolerances
segment_contour_attrib_xld (OuterContourWithWidth, \
                            OuterContourPartToNarrow, 'distance', 'or', \
                            0, MinWidth)
segment_contour_attrib_xld (OuterContourWithWidth, \
                            OuterContourPartToWide, 'distance', 'or', \
                            MaxWidth, 10000)
```

9.3.4 Extract Roads

Example: %HALCONEXAMPLES%/hdevelop/Applications/Object-Recognition-2D/roads.hdev

The task of this example is to extract the roads in the aerial image depicted in figure 9.7

Figure 9.7: (a) Original image; (b) zoomed image part; (c) extracted roads.

The programs starts by first extracting lines on a reduced scale. These lines correspond very well to roads.

```
threshold (Image, Region, 160, 255)
reduce_domain (Image, Region, ImageReduced)
MaxLineWidth := 5
Contrast := 70
calculate_lines_gauss_parameters (MaxLineWidth, Contrast, Sigma, Low, High)
lines_gauss (ImageReduced, RoadCenters, Sigma, Low, High, 'light', 'true', \
             'bar-shaped', 'true')
```

To eliminate wrong candidates, edges are extracted on a higher scale. For the road extraction it is assumed that a road consists of two parallel edges with homogeneous gray values and a line segment in between. Using some contour processing operators, this model is refined step by step.

```
edges_image (ImagePart, PartAmp, PartDir, 'mderiche2', 0.3, 'nms', 20, 40)
threshold (PartAmp, EdgeRegion, 1, 255)
clip_region (EdgeRegion, ClippedEdges, 2, 2, PartWidth - 3, PartHeight - 3)
skeleton (ClippedEdges, EdgeSkeleton)
gen_contours_skeleton_xld (EdgeSkeleton, EdgeContours, 1, 'filter')
gen_polygons_xld (EdgeContours, EdgePolygons, 'ramer', 2)
gen_parallels_xld (EdgePolygons, ParallelEdges, 10, 30, 0.15, 'true')
mod_parallels_xld (ParallelEdges, ImagePart, ModParallelEdges, \
                   ExtParallelEdges, 0.3, 160, 220, 10)
combine_roads_xld (EdgePolygons, ModParallelEdges, ExtParallelEdges, \
                   RoadCenterPolygons, RoadSides, rad(40), rad(20), 40, 40)
```

9.4 Relation to Other Methods

9.4.1 Alternatives to Contour Processing

Line Processing

A very basic alternative to contour processing are the operators for line processing. In this context, lines are treated as tuples of start and end points. The extraction can, e.g., be performed with `detect_edge_segments`. Of course, XLD polygons can also be converted into this type of lines. Operators for processing this type of lines can be found in the Reference Manual in the chapter "Tools ▷ Lines".

9.5 Advanced Topics

9.5.1 Line Scan Cameras

In general, line scan cameras are treated like normal area sensors. But in some cases, not single images but an infinite sequence of images showing objects, e.g., on a conveyor belt, have to be processed. In this case the end of one image is the beginning of the next one. This means that contours that partially lie in both images must be combined into one contour. For this purpose HALCON provides the operator `merge_cont_line_scan_xld`. This operator is called after the processing of one image and combines the current contours with those of previous images. For more information see Solution Guide II-A.

Chapter 10

2D Matching

The idea of matching is quite simple: In a training image a so-called template is presented. The system derives a model from this template. This model is then used to locate objects that look “similar” to the template in search images. Depending on the selected method, this approach is able to handle changes in illumination, clutter, varying size, position, and rotation, or even relative movement of parts of the template.

The advantage of matching is its ease of use combined with great robustness and flexibility. Matching does not require any kind of segmentation of the desired objects. By some of the matching methods, objects can be located even if they are overlapped by other objects.

HALCON offers different methods for matching. The selection depends on the image data and the task to be solved. This chapter offers an introduction to 2D matching. How to use the individual matching approaches is described in detail in the Solution Guide II-B. For the search of 3D models, see the explanations to 3D matching in Solution Guide I, chapter 11 on page 101.

The different 2D matching approaches:

- The *correlation-based matching* is based on gray values and a normalized cross correlation.
- The *shape-based matching* represents the state of the art in machine vision. Instead of using the gray values, features along contours are extracted and used both for the model generation and the matching.
- The *component-based matching* can be considered as a high-level shape-based matching: The enhancement is that an object can consist of multiple parts that can move (rotate and translate) relative to each other. A simple example of this is a pair of pliers. Logically this is considered as one object, but physically it consists of two parts. The component-based matching allows handling such a compound object in one search step.
- The *local deformable matching* is similar to the shape-based matching, but here significant deformations can be handled and returned. In particular, besides the position and score, the matching can return a rectified version of the significant part of the search image, a vector field that describes the deformations, and the deformed contours of the found model instance.
- The *perspective deformable matching* is also similar to the shape-based matching, but here also strong perspective deformations can be handled and instead of a 2D pose a 2D projective transformation matrix (homography) is returned. In addition, a calibrated version of the perspective deformable matching is available. There, instead of a 2D projective transformation matrix (homography) the 3D pose of the object is returned. Here, the focus is on the uncalibrated case. The calibrated case is described in more detail in the Solution Guide II-B in section 3.5 on page 100.
- The *descriptor-based matching* has the same intention as the perspective deformable matching, i.e., the 2D projective transformation matrix (homography) can be obtained for the uncalibrated case and the 3D pose can be obtained for the calibrated case. The main difference is that points instead of contours are used to create and find the model. Thus, it is especially suitable for highly textured objects but is not suitable for low textured objects with rounded edges. Compared to the perspective deformable matching, it is significantly faster for large search spaces but less accurate. Here, the focus is on the uncalibrated case. The calibrated case is described in more detail in the Solution Guide II-B in section 3.6 on page 108.

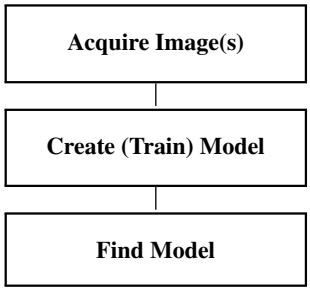
Figure 10.1 gives an overview of the different 2D matching approaches and summarizes their characteristics in order to help you to select the appropriate approach for your task.

Search for 2D models (orthogonal view)		
Correlation-based Matching	✓	Invariant to defocus, slight shape deformations, and linear illumination changes.
	✓	Works good for textured objects.
	×	Does not work with clutter, occlusion, non-linear illumination changes, scale, or multi-channel images.
Shape-based Matching	✓	Invariant to clutter, occlusion, non-linear illumination changes, scale, defocus, and slight shape deformations.
	✓	Works with multi-channel images.
	✓	Can be applied for multiple models simultaneously.
	×	Is difficult with some textures.
Component-based Matching	✓	Invariant to clutter, occlusion, non-linear illumination changes, and scale.
	✓	Works with multi-channel images.
	✓	Can be applied for multiple models simultaneously.
	×	Is difficult with some textures and does not work with defocus and shape deformations.
Local deformable Matching	✓	Invariant to clutter, occlusion, non-linear illumination changes, scale, and local deformations.
	✓	Works with multi-channel images.
Search for 2D models (orthogonal or perspective view)		
Perspective deformable Matching	✓	Invariant to clutter, occlusion, non-linear illumination changes, scale, defocus, and perspective shape deformations.
	✓	Works with multi-channel images.
	×	Is difficult with some textures.
Descriptor-based Matching	✓	Invariant to clutter, occlusion, non-linear illumination changes, scale, and perspective shape deformations.
	×	Does not work without texture, in particular distinctive points, and does not work with defocus or multi-channel images.

Figure 10.1: Overview strengths and weaknesses of the different 2D matching approaches.

10.1 Basic Concept

Matching is divided into the following parts:



10.1.1 Acquire Image(s)

Both for training and matching, first an image is acquired.

For detailed information see the description of this method on page 21.

10.1.2 Create (Train) Model

To create a matching model, first a region of interest that covers the template in the training image must be specified. Only those parts of the image that are really significant and stable should be used for training. The input for the

training operator is the reduced image together with control parameters. The handle of the model is the output of the training. The model will then be used for immediate search or stored to file.

10.1.3 Find Model

Having created (or loaded) a model, it can now be used for locating objects in the image. Each method offers specific methods to perform this task. If one or multiple objects are found, their poses (position, rotation, and scaling) or 2D projective transformation matrices (homographies) together with a score are returned. These values can already be the desired result or serve as input for the next step of the vision process, e.g., for aligning regions of interest.

10.2 Programming Examples

This section gives a brief introduction to using HALCON for template matching. This is done by showing an example for the basic concept and an example for each of the different 2D matching approaches mentioned.

10.2.1 A First Example

Example:
`%HALCONEXAMPLES%/hdevelop/Matching/Shape-Based/find_generic_shape_model_workflow.hdev`

This example shows all necessary steps mentioned in section 10.1 on page 90 from model generation to finding the object using shape-based matching.

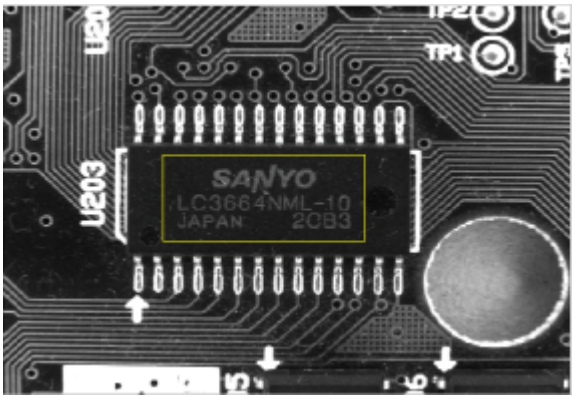


Figure 10.2: Finding the print in the image.

A training image is acquired from file and a model is created.

```
read_image (ReferenceImage, 'board/board_01')
create_generic_shape_model (ModelID)
```

A region is generated as region of interest (yellow in the image figure 10.2), covering the print in the image. After reducing the image to the extent of the region, it is used as input for the training operator `train_generic_shape_model`.

```
gen_rectangle1 (ROI, 450, 470, 580, 755)
reduce_domain (ReferenceImage, ROI, TrainingImage)
train_generic_shape_model (TrainingImage, ModelID)
```

The search parameters are set and the model is found in the images using `find_generic_shape_model`.

```
set_generic_shape_model_param (ModelID, 'num_matches', 1)
set_generic_shape_model_param (ModelID, 'min_score', 0.6)
*
for i := 1 to 9 by 1
    read_image (SearchImage, 'board/board_' + i$'02')
    find_generic_shape_model (SearchImage, ModelID, MatchResultID, \
                              NumMatchResult)
    * Retrieve iconic matching results.
    get_generic_shape_model_result_object (Objects, MatchResultID, 'all', \
                                           'contours')
    * Visualization.
    dev_display (SearchImage)
    dev_display (Objects)
    Text := 'Found instances'
    dev_disp_text (Text, 'window', 12, 12, 'black', [], [])
    wait_seconds (1)
endfor
```

10.2.2 Correlation-based Matching: Find Label in Texture

Example:
%HALCONEXAMPLES%/hdevelop/Matching/Correlation-Based/ncc_matching_workflow.hdev

This example shows how the operator `find_ncc_model` can be used to find instances without sharp edges. In the presented application, the masks are checked whether their CE mark is visible, see figure 10.3. You can also find more explanations to this example in Solution Guide II-B, section 3.1.1 on page 47 and to correlation-based matching in general in Solution Guide II-B, section 3.1 on page 47.

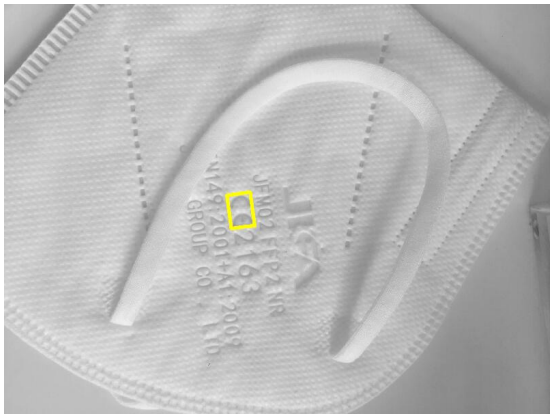


Figure 10.3: Check whether the logo (yellow) is visible in the image.

An oriented NCC model of the logo is created.

```
gen_rectangle2 (ROI, 616.5, 708.5, rad(-82.4054), 50, 35)
reduce_domain (Image, ROI, ImageReduced)
create_ncc_model (ImageReduced, 'auto', rad(0), rad(360), 'auto', \
                  'use_polarity', ModelID)
```

On new images, the model can directly be searched. The found matches are conveniently visualized using a visualization procedure.

```
for Index := 1 to NumImages by 1
    read_image (Image, 'face_masks/face_mask_' + Index$'02')
    find_ncc_model (Image, ModelID, rad(0), rad(360), 0.7, 1, 0.5, 'true', \
                    0, Row, Column, Angle, Score)
    dev_display_ncc_matching_results (ModelID, Color, Row, Column, Angle, 0)
endfor
```

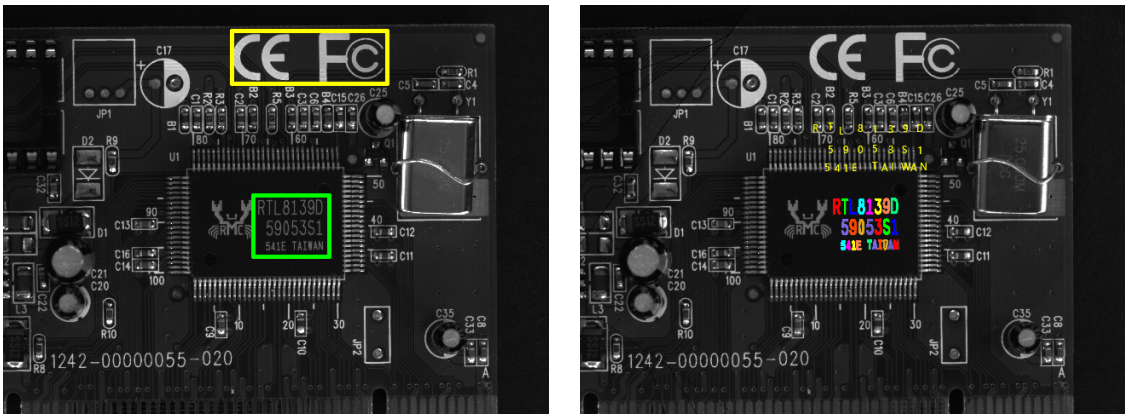


Figure 10.4: Left: The prominent logo is used to create a model (yellow) to find the text to be recognized in a specified region (green) on randomly oriented images. Right: The found text (colored) and the recognized text (yellow).

10.2.3 Shape-based Matching: Align the Image to Read Text

Example: `alignment_for_ocr_in_semiconductor.hdev` in `%HALCONEXAMPLES%/hdevelop/Matching/Shape-Based/`

The task of this example is to read text on randomly oriented images. In order to do so, a prominent logo is used as matching model, see figure 10.4 on the left. This logo is always located in a fixed relation to the text area. So the transformation of the logo is used to align the image. In the aligned image the text within the determined area is recognized, see figure 10.4 on the right.

In a first step we read the reference image and define the regions to create the model and locate the text.

```
read_image (ReferenceImage, 'board/board_01')
gen_rectangle1 (ROIModel, 60, 535, 185, 900)
gen_rectangle1 (ROIText, 445, 585, 590, 765)
reduce_domain (ReferenceImage, ROIModel, ModelImage)
```

Now we can create the matching model and train it. Additionally the text model is created, which is only needed for the additional task of text recognition.

```
create_generic_shape_model (ModelHandle)
train_generic_shape_model (ModelImage, ModelHandle)
create_text_model_reader ('auto', 'Industrial_0-9A-Z_Rej.omc', TextModel)
```

We need the reference transformation for the later alignment. We can extract it by finding the instance on the reference image. Therefore we call the operator `find_generic_shape_model` already in the offline phase.

```
set_generic_shape_model_param (ModelHandle, 'num_matches', 1)
set_generic_shape_model_param (ModelHandle, 'min_score', 0.5)
find_generic_shape_model (ReferenceImage, ModelHandle, MatchResultID, \
                          Matches)
get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                HomMat2DModel)
```

Now we created the needed models and we know the reference transformation. So we can go to the online phase. We look for a match of the logo and extract its transformation. Together with the transformation of the reference model we can align the image. In the aligned image we know the region in which we expect the text so we can extract and recognize it.

```
for i := 1 to 9 by 1
    read_image (SearchImage, 'board/board_' + i$'02')
    find_generic_shape_model (SearchImage, ModelHandle, MatchResultID, \
                              Matches)
    get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                    HomMat2DMatch)
    * Compute the transformation matrix.
    hom_mat2d_invert (HomMat2DMatch, HomMat2DMatchInvert)
    hom_mat2d_compose (HomMat2DModel, HomMat2DMatchInvert, \
                       TransformationMatrix)
    affine_trans_image (SearchImage, ImageAffineTrans, TransformationMatrix, \
                        'constant', 'false')
    get_generic_shape_model_result_object (InstanceObject, MatchResultID, \
                                           'all', 'contours')
    dev_display (InstanceObject)
    reduce_domain (ImageAffineTrans, ROIText, ImageOCR)
    find_text (ImageOCR, TextModel, TextResultID)
    get_text_object (Characters, TextResultID, 'all_lines')
    get_text_result (TextResultID, 'class', RecognizedText)
endfor
```

10.2.4 Component-based Matching: Check the State of a Dip Switch

Example:
%HALCONEXAMPLES%/hdevelop/Applications/Position-Recognition-2D/cbm_dip_switch.hdev

The task of this example is to check a dip switch, i.e., to determine the positions of the single switches relative to the casing (see figure 10.5).

a) b)

Figure 10.5: (a) Model image of the dip switch with ROIs for the components; (b) located dip switch with state of the switches.

The task is solved using component-based matching. The dip switch consists of 14 components: 12 for the switches and two for the printed text on the casing (see figure 10.5a). The training is performed using training images that show all possible positions of the switches. From this, the system learns the possible movements of the switches relative to the casing.

```
train_model_components (ModelImage, InitialComponents, TrainingImages, \
                        ModelComponents, 45, 45, 30, 0.95, -1, -1, rad(20), \
                        'speed', 'rigidity', 0.2, 0.5, ComponentTrainingID)
```

To make the recognition more robust, small tolerances are added to the trained movements.

```
modify_component_relations (ComponentTrainingID, 'all', 'all', 0, rad(4))
```

Now, the model can be created.

Figure 10.6: (left) “MVTec” logo used for the model creation; (right) deformed logo instance overlaid by the model contours.

```
create_trained_component_model (ComponentTrainingID, 0, rad(360), 10, \
                                MinScoreComp, NumLevelsComp, 'auto', \
                                'none', 'use_polarity', 'false', \
                                ComponentModelID, RootRanking)
```

When searching for the dip switch, not only the casing but each single dip together with its relative position is returned. Based on this information, the global status of the switch can easily be derived.

```
find_component_model (SearchImage, ComponentModelID, RootRanking, 0, \
                      rad(360), 0, 0, 0.5, 'stop_search', 'prune_branch', \
                      'none', MinScoreComp, 'least_squares', 0, 0.9, \
                      ModelStart, ModelEnd, Score, RowComp, ColumnComp, \
                      AngleComp, ScoreComp, ModelComp)
```

10.2.5 Local Deformable Matching: Find Deformed Logo

Example: `find_local_deformable_model.hdev` in
`%HALCONEXAMPLES%/hdevelop/Matching/Deformable`

In this example we locate differently deformed “MVTec” logos (see figure 10.6). You can also find more explanations to this example in Solution Guide II-B, section 3.4.1 on page 91 and to local deformable matching in general in Solution Guide II-B, section 3.4 on page 91.

First, the template is prepared. The colored image (figure 10.6, left) is transformed into a gray value image from which an ROI, i.e., the template image (figure 10.6, left) is derived. The template image is then used to create a model of the logo using `create_local_deformable_model`.

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

The created model is used to find instances of the logo in search images.

Figure 10.7: From left to right: template image, corresponding part of the search image, rectified image.

```
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)
```

The position of the object in the image is returned. Additionally, iconic objects can be returned, e.g., 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 10.7, right). But also the deformations can be visualized, see the example and its entry in Solution Guide II-B in section 3.4.1 on page 91.

10.2.6 Perspective Deformable Matching: Locate Road Signs

Example:
`%HALCONEXAMPLES%/hdevelop/Applications/Traffic-Monitoring/detect_road_signs.hdev`

The task of this example is to locate road signs in images as depicted in figure 10.8. Available road signs comprise the attention sign and the dead end road sign.

Figure 10.8: A dead end road sign is located via uncalibrated perspective deformable matching.

The example first reads a synthetic image that shows the model of an attention sign. The first channel of the model is accessed (`access_channel`) and zoomed (`zoom_image_factor`). Before calling `create_planar_uncalib_deformable_model` to create the perspective deformable model, it is checked if the value `'3'` for the number of levels is suitable. For this, the operator `inspect_shape_model` is applied and the result is checked visually. After the creation of the model, the model is stored in the tuple `Models`.

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

Then, the image containing a dead end road sign is read and the corresponding model is created. Here, the model is not available as artificial model but is derived from an image showing a typical street scenario. Thus, the processing of the image differs in parts from the proceeding used for the synthetic model. Among others, the domain of the image is reduced to a rectangle containing the road sign . The model obtained from the call to `create_planar_uncalib_deformable_model` is added to the tuple `Models`.

```
read_image (ImageDeadEnd, 'road_signs/dead_end_road_sign')
access_channel (ImageDeadEnd, Image, Channel[1])
gray_closing_shape (Image, ImageClosing, 5, 5, 'octagon')
zoom_image_factor (ImageClosing, ImageZoomed, 0.4, 0.4, 'weighted')
gen_rectangle1 (Rectangle1, 28, 71, 69, 97)
reduce_domain (ImageZoomed, Rectangle1, ImageReduced)
create_planar_uncalib_deformable_model (ImageReduced, 3, [], [], 0.1, \
                                        ScaleRMin[1], [], 0.05, \
                                        ScaleRMin[1], [], 0.1, 'none', \
                                        'use_polarity', 'auto', 'auto', [], \
                                        [], ModelID)
Models := [Models,ModelID]
```

To search for both models in unknown images, a rectangular ROI is created as search space. Within this ROI, the procedure `determine_area_of_interest` automatically determines more specific regions of interest (ROI) for the road signs using a blob analysis. Each model is searched in a specific channel of the image as the attention sign can better be extracted in the blue channel and the dead end road sign is better extracted in the red channel. The channels are specified at the beginning of the program inside the tuple `Channel`. The individual model is then searched in the reduced image with `find_planar_uncalib_deformable_model`.

```
gen_rectangle1 (Rectangle, 115, 0, 360, 640)
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)
    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)
```

If a model could be found, i.e., if a 2D projective transformation matrix was returned, the 2D projective transformation matrix is used to project the contour of the specific model onto the found instance of the model inside the inspected image.

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

10.2.7 Descriptor-based Matching: Locate Brochure Pages

Example:
%HALCONEXAMPLES%/hdevelop/Applications/Object-Recognition-2D/detect_brochure_pages.hdev

The task of this example is to locate different pages of a HALCON brochure as depicted in figure 10.9 using descriptor-based matching.

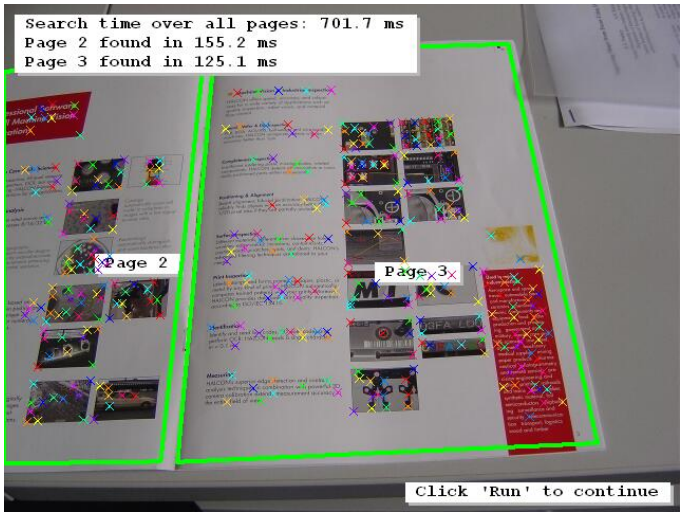


Figure 10.9: Different pages of a HALCON brochure are located via descriptor-based matching.

First, images that contain models of the different pages of the brochure are read, converted into gray value images, and their domains are reduced to a rectangular ROI. From this ROI, the operator create_uncalib_descriptor_model creates a descriptor-based model for each model image. That is, the detector, which extracts the interest points of the model, is selected (harris_binomial) and parameters for the descriptor, which builds characteristic descriptions of the gray-value neighborhood of the interest points, are set.

```
RowRoi := [10, 10,Height - 10,Height - 10]
ColRoi := [10,Width - 10,Width - 10,10]
gen_rectangle1 (Rectangle, 10, 10, Height - 10, Width - 10)
for Index := 1 to NumModels by 1
    read_image (Image, 'brochure/brochure_page_' + Index$'.2')
    rgb1_to_gray (Image, ImageGray)
    get_image_size (ImageGray, Width, Height)
    reduce_domain (ImageGray, Rectangle, ImageReduced)
    create_uncalib_descriptor_model (ImageReduced, 'harris_binomial', [], \
                                     [], ['min_rot', 'max_rot', \
                                     'min_scale', 'max_scale'], [-90, 90, \
                                     0.2, 1.1], 42, ModelID)
```

The origin of each model is moved from the center of gravity of the model to the center of the rectangle used to create the ROI. This is done with set_descriptor_model_origin so that the rectangle can be easily projected correctly onto the model when visualizing the results of the search in a later step. The adapted

model is then stored in the tuple ModelIDs. The interest points extracted for each model are queried with get_descriptor_model_points and the number of extracted points is stored in the tuple NumPoints.

```
    set_descriptor_model_origin (ModelID, -Height / 2, -Width / 2)
    ModelIDs := [ModelIDs,ModelID]
    get_descriptor_model_points (ModelID, 'model', 'all', Row_D, Col_D)
    NumPoints := [NumPoints,|Row_D|]
endfor
```

Now, the unknown images are read and transformed into gray value images. For each model, the operator find_uncalib_descriptor_model searches for instances of the model in the image.

```
for Index1 := 1 to 12 by 1
    OutputString := []
    read_image (Image, 'brochure/brochure_' + Index1$'.2')
    rgb1_to_gray (Image, ImageGray)
    for Index2 := 0 to |ModelIDs| - 1 by 1
        find_uncalib_descriptor_model (ImageGray, ModelIDs[Index2], \
                                       'threshold', 600, \
                                       ['min_score_descr', \
                                       'guided_matching'], [0.003, 'on'], \
                                       0.25, 1, 'num_points', HomMat2D, \
                                       Score)
```

If a valid 2D projective transformation matrix (homography) and a specific minimum score (that depends on the number of the extracted interest points) was obtained by the search, the points of the model are queried with get_descriptor_model_points and displayed by cross contours. Then, the rectangle specified for the creation of the model and its corner points are transformed with the 2D projective transformation matrix (projective_trans_region and projective_trans_pixel). The transformed corner points are used to calculate the angle between two neighboring edges of the perspectively projected rectangle (angle_ll). In the model, the edges are right angled. In the perspective deformed projection they should not deviate more than 20 degrees from the right angle. Thus, only if an angle between 70 and 110 degrees is obtained, the found instance of the model is accepted and the result is displayed.

```
        if ((|HomMat2D| > 0) and (Score > NumPoints[Index2] / 4))
            get_descriptor_model_points (ModelIDs[Index2], 'search', 0, Row, \
                                         Col)
            gen_cross_contour_xld (Cross, Row, Col, 6, 0.785398)
            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)
            if (Angle > 70 and Angle < 110)
                area_center (TransRegion, Area, Row, Column)
                disp_message (WindowHandle, 'Page ' + (Index2 + 1), \
                              'window', Row, Column, 'black', 'true')
            endif
        endif
    endfor
endfor
```

10.3 Relation to Other Methods

10.3.1 Methods that are Using Matching

1D Measuring (see description on page 45)
OCR (see description on page 183)

Variation Model (see description on page 111)

Bar Code (see description on page 159)

The pose or 2D projective transformation matrix (homography) returned by the matching operators can be used as the input for a so-called alignment. This means that either the position of an ROI is transformed relative to the movement of a specified object in the image, or the image itself is transformed so that the pixels are moved to the desired position.

A detailed description of alignment can be found in the Solution Guide II-B in section 2.5.3.2 on page 35. For an example of using alignment as a preprocessing for 1D Measuring on page 45 see the description of `pm_measure_board.hdev` on page 49. Further examples can be found in the description of the method Variation Model on page 111.

10.3.2 Alternatives to Matching

Blob Analysis (see description on page 33)

In some applications, the object to find can be extracted with classical segmentation methods. With the operators `area_center` and `orientation_region` you can then determine its pose and use this information, e.g., to align an ROI. With this approach, the execution time can be reduced significantly.

If the objects to be found appear only translated but not rotated or scaled, the morphological operators `erosion1` and `opening` can be used as binary matching methods. Unlike in other vision systems, in HALCON these operators are extremely fast.

Chapter 11

3D Matching

The aim of 3D matching is to find a specific 3D object in search data and determine its pose, i.e., the position and orientation of the object in space. For that, HALCON provides different approaches:

- For the *shape-based 3D matching*, a 3D shape model is generated from a 3D computer aided design (CAD) model, which must be available in one of the supported formats that are listed in the description of `read_object_model_3d` in the Reference Manual (e.g., DXF, PLY, or STL). The 3D shape model consists of 2D projections of the 3D object seen from different views. Analogously to the shape-based matching of 2D structures described in the chapter Matching on page 89, the 3D shape model is used to recognize instances of the object in an image. But here, instead of a 2D position, orientation, and scaling, the 3D pose of each instance is returned.
- For the *surface-based 3D matching*, a surface model is derived either from a CAD model that is available in one of the supported formats that are listed in the description of `read_object_model_3d` in the Reference Manual (e.g., DXF, PLY, or STL), or from a 3D object model that is available in OM3 format. The latter can be obtained from images using a 3D reconstruction approach, e.g., stereo vision on page 215. In contrast to shape-based 3D matching, the object is not searched within an image, but within a 3D scene that is available as a 3D object model. Like shape-based 3D matching, surface-based 3D matching returns the 3D pose of each instance of the object that can be located.
- The *deformable surface-based 3D matching* is similar to the above mentioned *surface-based 3D matching*, but it can also find deformed 3D objects. In addition, it allows to define some special 3D points, e.g., grasping points, for the reference object. These points are called reference points. Once a (deformed) 3D object has been found, the position of the reference points can be adapted to the deformations of the found object to, e.g., grasp the deformed object.

If you need the 3D pose of a planar object or a planar object part, we recommend to use a calibrated perspective matching approach instead of the 3D matching. Available approaches are the calibrated perspective deformable matching and the calibrated descriptor-based matching. Both are significantly faster and more convenient to use.

Note that for the shape-based 3D matching as well as for the calibrated perspective deformable matching and the calibrated descriptor-based matching a camera calibration is necessary. For the (deformable) surface-based matching, the camera calibration is needed when acquiring the 3D object model(s) from images using a 3D reconstruction approach.

The main focus of this section lies on 3D matching. For the calibrated perspective deformable matching and the calibrated descriptor-based matching we refer to the Solution Guide II-B, section 3.4 on page 91 and section 3.6 on page 108. Note that the description here provides you with an overview on the 3D matching approaches. Further details can be found in the Solution Guide III-C, section 4.2 on page 95 for shape-based 3D matching, section 4.3 on page 104 for surface-based 3D matching, and section 4.4 on page 107 for deformable surface-based 3D matching, as well as in the Reference Manual.

11.1 Basic Concept

3D matching consists of the following basic steps:

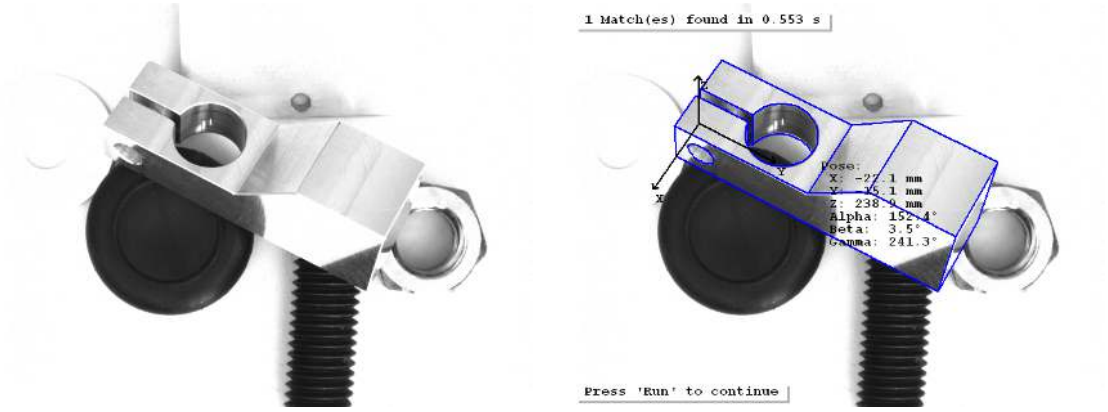
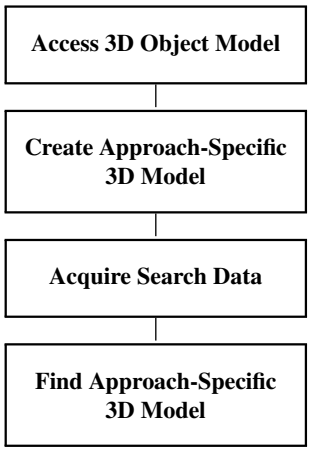


Figure 11.1: Shape-based 3D matching: (left) original image containing a clamp, (right) 3D model of the found clamp projected in the image and display of its pose.



11.1.1 Access 3D Object Model

Before creating the approach-specific 3D model, i.e., a 3D shape model or a surface model, the 3D object model describing the object of interest must be accessed.

- For *shape-based 3D matching* the 3D object model is accessed with the operator `read_object_model_3d`. Supported CAD formats are, e.g., DXF, PLY, or STL. The complete list can be found in the description of the operator in the Reference Manual. The DXF format (DXF version AC1009, AutoCad release 12) may consist of the DXF entities POLYLINE (Polyface meshes), 3DFACE, LINE, CIRCLE, ARC, ELLIPSE, SOLID, BLOCK, and INSERT.
- For *(deformable) surface-based 3D matching* the 3D object model is accessed either from file using the operator `read_object_model_3d` or by an online 3D reconstruction. If the 3D object model is accessed from a file, besides the supported CAD formats also the OM3 format can be handled, which is a HALCON-specific format that can be obtained, e.g., by an offline 3D reconstruction.

The description of `read_object_model_3d` in the Reference Manual provides additional tips on how to obtain a suitable model.

11.1.2 Create Approach-Specific 3D Model

Having access to the 3D object model, the approach-specific 3D model can be created, which prepares the 3D object model for the specific 3D matching approach.

- For *shape-based 3D matching*, the operator create_shape_model_3d creates a 3D shape model. It is generated by computing different views of the 3D object model within a user-specified pose range. The views are obtained by placing virtual cameras around the object model and projecting the 3D object model into the image plane of each camera position. The resulting 2D shape representations of all views are stored in the 3D shape model. To avoid storage and runtime problems, the specified pose range should be restricted as much as possible so that the number of 2D projections that have to be computed and stored in the 3D shape model is minimized. To create the model views correctly, the camera parameters are needed. The camera parameters can be obtained by a camera calibration. How to apply a camera calibration is described in detail in the Solution Guide III-C, section 3.2 on page 61.
- For *(deformable) surface-based matching*, the operators create_surface_model and create_deformable_surface_model, respectively, create a surface model by sampling the 3D object model with a certain distance, so that for an approximate matching, which is the first of three steps the later search consists of, only a reduced set of 3D points must be examined.

11.1.3 Acquire Search Data

If a *shape-based 3D matching* is applied, search images are acquired. For detailed information see the description of this method on page 21.

If a *(deformable) surface-based 3D matching* is applied, the search space is not an image but a 3D scene that is available as 3D object model, which can be obtained from images using a 3D reconstruction approach. An overview on the available 3D reconstruction approaches is given in the Solution Guide III-C, chapter 1 on page 9.

11.1.4 Find Approach-Specific 3D Model

With the approach-specific 3D model that was created with create_shape_model_3d, create_surface_model, or create_deformable_surface_model or that was read from file with read_shape_model_3d, read_surface_model, or read_deformable_surface_model (see Re-use Approach-Specific 3D Model on page 105), the object can be searched for in the search data, i.e., in images for *shape-based 3D matching* and in a 3D scene that is represented by a 3D object model for *(deformable) surface-based matching*.

For the search, the operator find_shape_model_3d, find_surface_model, or find_deformable_surface_model is applied. Several parameters can be set to control the search process. For detailed information, we recommend to read the descriptions of the operators in the Reference Manual. Both operators return the pose of the matched model instance and a score that describes the quality of the match.

11.1.5 A First Example

An example for this basic concept is the following program. It applies a shape-based 3D matching to locate clamps.

The DXF file containing the 3D model of the clamp shown in Figure 11.1 on page 102 is read with read_object_model_3d.

```
read_object_model_3d ('clamp_sloped', 'mm', [], [], ObjectModel3DID, \
                      DxfStatus)
```

For the creation of a 3D shape model, the camera parameters are needed. They can be determined by a camera calibration as described in the Solution Guide III-C, section 3.2 on page 61. Here, they are known and just assigned to the variable CamParam.

```
gen_cam_par_area_scan_division (0.01221, -2791, 7.3958e-6, 7.4e-6, 308.21, \
                                245.92, 640, 480, CamParam)
```

The creation of the 3D shape model is applied using create_shape_model_3d. There, besides the camera parameters the following further parameters are needed: the reference orientation and the pose range in which the object is expected, i.e., the minimum and maximum longitude and latitude, as well as the minimum and maximum

distance between the camera and the center of the object's bounding box (which correspond to the radii of spheres that are built to compute the virtual camera positions). The range for the camera roll angle is set to a full circle. Further information about these parameters can be found in the Reference Manual and in the Solution Guide III-C, section 4.2 on page 95.

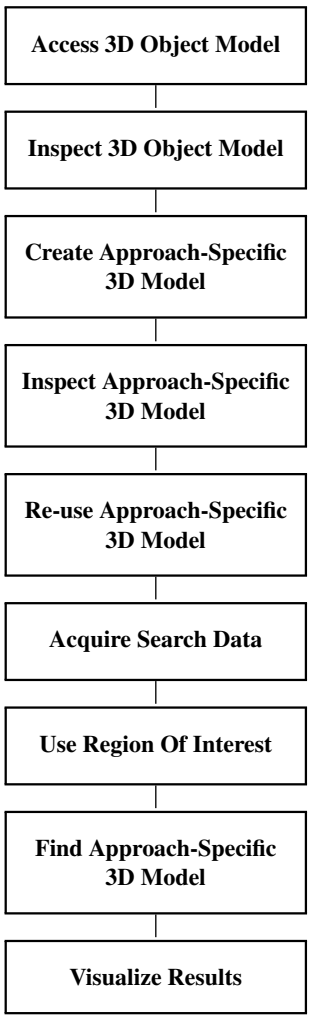
```
create_shape_model_3d (ObjectModel3DID, CamParam, RefRotX, RefRotY, \
                       RefRotZ, 'gba', LongitudeMin, LongitudeMax, \
                       LatitudeMin, LatitudeMax, 0, rad(360), DistMin, \
                       DistMax, 10, 'min_face_angle', MinFaceAngle, \
                       ShapeModel3DID)
```

The search images are acquired and the actual matching is applied in each image using the operator `find_shape_model_3d`.

```
read_image (Image, 'clamp_sloped/clamp_sloped_' + ImageNo$'02')
find_shape_model_3d (Image, ShapeModel3DID, 0.7, 0.9, 5, ['num_matches', \
                     'pose_refinement'], [2, \
                     'least_squares_very_high'], Pose, CovPose, Score)
```

11.2 Extended Concept

Often, more than the essential steps are necessary. You can, e.g., store the approach-specific model into a file and read it from another application to separate the creation of the model from the actual search process. These and further advanced steps are described in the following sections.



11.2.1 Inspect 3D Object Model

3D object models can be inspected using get_object_model_3d_params. Depending on the parameter specified in GenParamName, you can query different attributes of the 3D object model like the number of points or the parameters of the bounding box. A procedure for the visualization of 3D object models (disp_object_model_3d) can be found, e.g., with the HDevelop example program %HALCONEXAMPLES%/hdevelop/3D-Matching/Surface-Based/find_surface_model.hdev.

11.2.2 Inspect Approach-Specific 3D Model

Similar to the 3D object model, you can query also the parameters for the approach-specific 3D model:

- The 3D shape model that was created with create_shape_model_3d can be inspected using get_shape_model_3d_params. Several parameters like the reference orientation of the 3D object model, the camera parameters, or the pose range used for the creation of the 3D shape model can be queried.

  The contours of a specific view of the 3D shape model can be accessed with get_shape_model_3d_contours. The contours can be used to visualize and rate the 3D shape model and thus decide if certain parameters have to be adjusted within the search process using find_shape_model_3d or if the creation of the 3D shape model has to be repeated with adjusted parameters.

- The (deformable) surface model that was created with create_surface_model or create_deformable_surface_model, respectively, can be inspected using get_surface_model_param or get_deformable_surface_model_param, respectively. Several parameters like the center of the model or the 3D points of the sampled model versions needed for different steps of the search process can be queried.

11.2.3 Re-use Approach-Specific 3D Model

Because of the complex calculations, the creation of a 3D model can be rather time consuming. Thus, if you need a 3D model more than once, it is recommended to store the approach-specific 3D model, especially the 3D shape model that was created with create_shape_model_3d, into a file rather than to repeat the model creation. To store the model into file you use the operator write_shape_model_3d for *shape-based 3D matching*, the operator write_surface_model for *surface-based 3D matching*, and the operator write_deformable_surface_model for *deformable surface-based 3D matching*. To read the 3D shape model or (deformable) 3D surface model from file again to re-use the model in another application, you apply read_shape_model_3d, read_surface_model, or read_deformable_surface_model, respectively.

11.2.4 Use Region Of Interest

When searching in images, i.e., when using *shape-based 3D matching*, the search can be sped up using a region of interest. The more the region in which the objects are searched can be restricted, the faster and more robust the search will be.

For detailed information see the description of this method on page 25.

11.2.5 Visualize Results

A typical visualization task for *shape-based 3D matching* is to use the operator project_object_model_3d or project_shape_model_3d to project the outline of the 3D object model or the 3D shape model into the image. For both operators, an object-camera pose is needed, which can be determined, e.g., by the operator create_cam_pose_look_at_point. Another common visualization task is to display the pose that is obtained by find_shape_model_3d.

A typical visualization task for *surface-based 3D matching* is to use the operator project_object_model_3d and a known object-camera pose to project the outline of the 3D object model that is used as model into

an image. For example, if the 3D object model that is used as search data was reconstructed with a multi-view stereo setup, the model can be projected into one of the used stereo images. This type of visualization is applied, e.g., in the HDevelop example program %HALCONEXAMPLES%/hdevelop/Applications/Robot-Vision/locate_pipe_joints_stereo.hdev. Another common visualization task is to display the 3D object model that is used as model not in an image but within a 3D scene, e.g., within the 3D object model that is used as search data. A procedure that allows to interactively inspect such a 3D scene (visualize_object_model_3d) is provided with the HALCON installation as an external procedure. It is used, e.g., in the HDevelop example program %HALCONEXAMPLES%/hdevelop/3D-Matching/Surface-Based/find_surface_model.hdev.

For *deformable surface-based 3D matching* it is easy to display the 3D object model that is used as model within a 3D scene, e.g., within the 3D object model that is used as search data. The operator find_deformable_surface_model returns a result handle from which the deformed model can be retrieved with get_deformable_surface_matching_result. This is, e.g., shown in the HDevelop example program %HALCONEXAMPLES%/hdevelop/3D-Matching/Deformable-Surface-Based/find_deformable_surface_model.hdev.

For general information about visualizing results see the description of the method Visualization on page 223.

11.3 Programming Examples

This section shows how to use HALCON for 3D matching.

11.3.1 Recognize 3D Clamps and Their Poses in Images

Example: %HALCONEXAMPLES%/hdevelop/Applications/Position-Recognition-3D/3d_matching_clamps.hdev

This example shows how to apply shape-based 3D matching to find a 3D object, in particular the clamp shown in Figure 11.1 on page 102, in images. Besides the basic steps introduced in the first example, advanced steps are applied and a procedure is provided that helps to interactively determine an appropriate pose range for the creation of the 3D shape model.

At the beginning of the program, you can select by commenting or uncommenting the corresponding code lines whether you want to use an already created 3D shape model or whether you want to create a new one. Further, the camera parameters are assigned to a variable and the 3D object model of the clamps is loaded.

```
ReCreateShapeModel3D := false
* ReCreateShapeModel3D := true
gen_cam_par_area_scan_division (0.01221, -2791, 7.3958e-6, 7.4e-6, 308.21, \
                                245.92, 640, 480, CamParam)
read_object_model_3d ('clamp_sloped', 'mm', [], [], ObjectModel3DID, \
                      DxfStatus)
```

The procedure inspect_object_model_3d can now be used to interactively visualize the model, set the reference pose, and determine further parameters for the creation of the 3D shape model. For example, selected images that were acquired before calling the procedure can be succesively added to the visualization by selecting the buttons Next Image or Previous Image. With the mouse, you can rotate the 3D object model and, after selecting the mouse mode Move Image, move the underlying image so that the object model matches the image (see Figure 11.2). The returned pose for each image can be added to the pose range by the button Add to Pose Range. When leaving the procedure by selecting Exit, the procedure returns values for a pose range that can be used to create a 3D shape model.

```
gen_empty_obj (Images)
for Index := 1 to 3 by 1
    read_image (Image, 'clamp_sloped/clamp_sloped_' + Index$'02')
    concat_obj (Images, Image, Images)
endfor
inspect_object_model_3d (Images, ObjectModel3DID, CamParam, RefRotX, \
                         RefRotY, RefRotZ, LongitudeMin, LongitudeMax, \
                         LatitudeMin, LatitudeMax, CamRollMin, CamRollMax, \
                         DistMin, DistMax, MinFaceAngle)
```

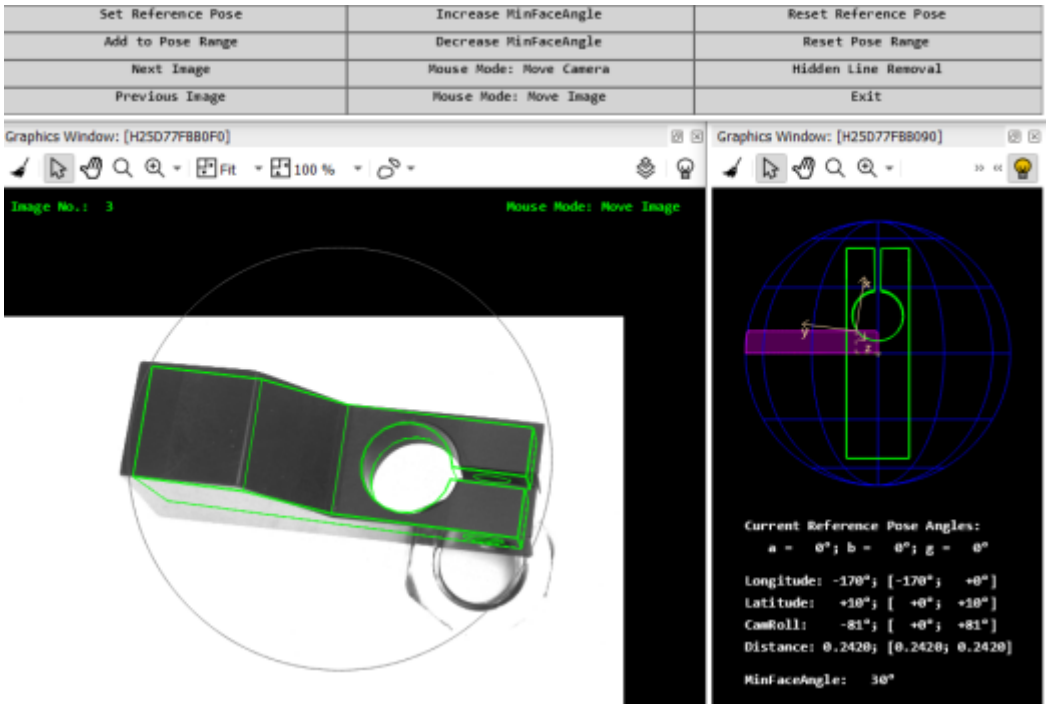


Figure 11.2: Interactive determination of the reference pose and the pose range for the 3D matching of clamps.

Additionally, within the procedure a `Hidden Line Removal` mode can be selected and the minimum face angle (`Increase MinFaceAngle`, `Decrease MinFaceAngle`) can be set. The latter is used, e.g., to suppress edges that are needed to approximate curved structures in the 3D object model but do not appear in the image and thus cannot be matched in the image (see Figure 11.3 on page 108).

Within the main procedure, the operator `create_shape_model_3d` creates a new 3D shape model if the variable `ReCreateShapeModel3D` was set to `true`. For this, the parameters determined by the procedure described above are used. The resulting 3D shape model is written into file using `write_shape_model_3d`. If an already existing 3D shape model is used (`ReCreateShapeModel3D` set to `false`) and the procedure was used mainly for visualization purposes, the existing 3D shape model is read from file using `read_shape_model_3d`.

```
if (ReCreateShapeModel3D)
    create_shape_model_3d (ObjectModel3DID, CamParam, RefRotX, RefRotY, \
                           RefRotZ, 'gba', LongitudeMin, LongitudeMax, \
                           LatitudeMin, LatitudeMax, 0, rad(360), DistMin, \
                           DistMax, 10, 'min_face_angle', MinFaceAngle, \
                           ShapeModel3DID)
    write_shape_model_3d (ShapeModel3DID, 'clamp_sloped_user.sm3')
else
    read_shape_model_3d ('clamp_sloped_35.sm3', ShapeModel3DID)
endif
```

For the actual 3D matching, all images that contain the clamps are read and for each image the operator `find_shape_model_3d` searches for instances of the 3D shape model in the image. With the resulting values, the edges of the 3D shape model are projected into the image using `project_shape_model_3d`. The numerical values of the pose are displayed using the procedure `display_match_pose`.

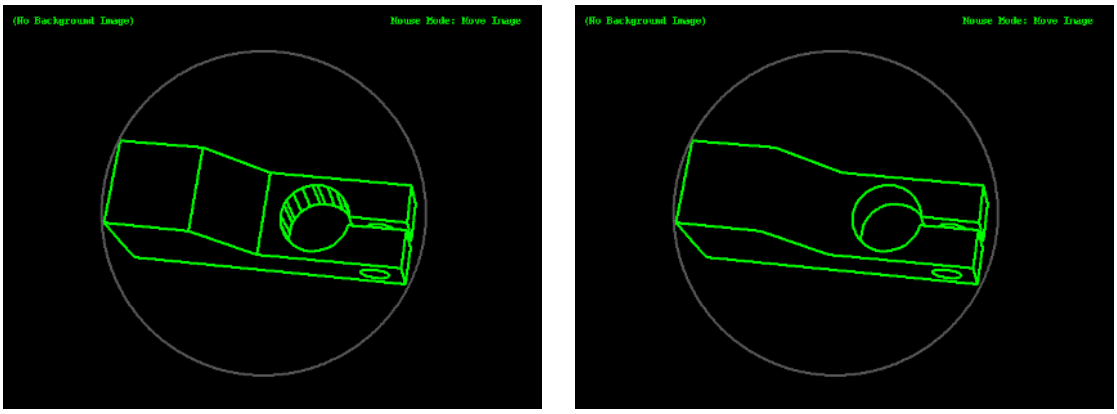


Figure 11.3: Different minimum face angles: (left) 8 degrees, (right) 45 degrees.

```
for ImageNo := 1 to 35 by 1
    read_image (Image, 'clamp_sloped/clamp_sloped_' + ImageNo$'02')
    find_shape_model_3d (Image, ShapeModel3DID, 0.7, 0.9, 5, ['num_matches', \
                         'pose_refinement'], [2, \
                         'least_squares_very_high'], Pose, CovPose, Score)
    for I := 0 to |Score| - 1 by 1
        PoseI := Pose[I * 7:I * 7 + 6]
        CovPoseI := CovPose[I * 6:I * 6 + 5]
        ScoreI := Score[I]
        project_shape_model_3d (ModelContours, ShapeModel3DID, CamParam, \
                                PoseI, 'true', 0.523599)
        display_match_pose (ShapeModel3DID, PoseI, WindowHandle)
    endfor
endfor
```

Inside the procedure `display_match_pose` the reference point and the camera parameters are queried by `get_shape_model_3d_params` to align the displayed strings in the image.

```
get_shape_model_3d_params (ShapeModel3DID, 'reference_point', \
                           ReferencePoint)
get_shape_model_3d_params (ShapeModel3DID, 'cam_param', CamParam)
pose_to_hom_mat3d (Pose, HomMat3D)
affine_trans_point_3d (HomMat3D, ReferencePoint[0], ReferencePoint[1], \
                       ReferencePoint[2], X, Y, Z)
project_3d_point (X, Y, Z, CamParam, Row, Column)
set_tposition (WindowHandle, Row, Column - 10)
write_string (WindowHandle, 'Pose:')
set_tposition (WindowHandle, Row + 15, Column)
write_string (WindowHandle, 'X: ' + (1000 * Pose[0])$'4.1f' + ' mm')
... etc. ...
```

11.3.2 Recognize Pipe Joints and Their Poses in a 3D Scene

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Robot-Vision/locate_pipe_joints_stereo.hdev`

This example shows how to apply surface-based 3D matching to find a 3D object, in particular a pipe joint, in a 3D scene given as a 3D object model.

First, the model is accessed. It is available as a CAD model in PLY format and is accessed with `read_object_model_3d`. From this 3D object model the corresponding surface model is created using `create_surface_model`.

```
read_object_model_3d ('pipe_joint', 'm', [], [], PipeJointOM3DID, Status)
create_surface_model (PipeJointOM3DID, 0.03, [], [], PipeJointSMID)
```

In the example, the 3D scene that builds the search data is accessed using multi-view stereo. Note that the following code shows only a part of the stereo reconstruction. The rest is explained in section 20.3.2 on page 220. Figure 11.4 shows the images that are used to reconstruct one of the 3D scenes.

```
for Index := 1 to NumImages by 1
    read_multi_view_stereo_images (Images, ImagePath, ImagePrefix, Index, \
                                   NumCameras)
    reconstruct_surface_stereo (Images, StereoModelID, PipeJointPileOM3DID)
```

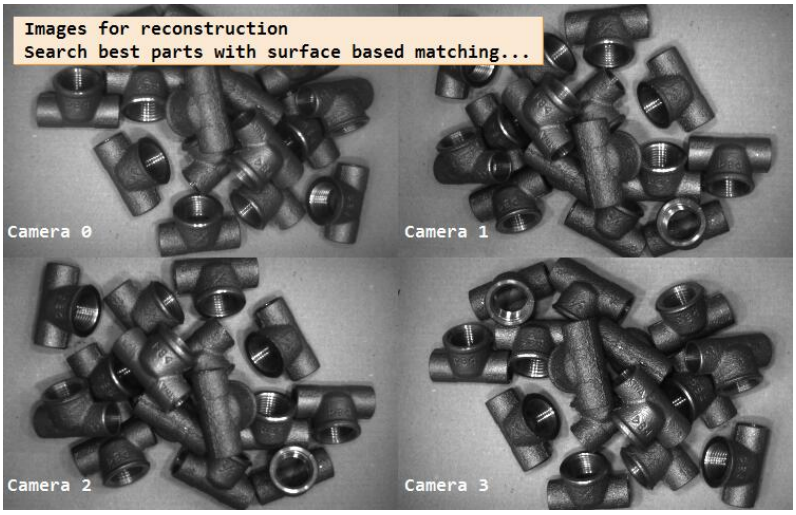


Figure 11.4: Images used for the multi-view stereo reconstruction of the search data (3D scene).

The matching is applied with `find_surface_model`. It searches the surface model in the 3D scene and returns the poses of the best matching model instances.

```
NumMatches := 5
Params := ['num_matches', 'pose_ref_scoring_dist_rel', \
           'scene_normal_computation']
Values := [NumMatches,0.02, 'mls']
    find_surface_model (PipeJointSMID, PipeJointPileOM3DID, 0.03, 0.05, \
                        MinScore, 'false', Params, Values, Poses, Scores, \
                        SurfaceMatchingResultID)
```

To visualize the result of each match the 3D object model that was accessed from the CAD model and that was prepared for visualization is transformed with the pose that was returned for the specific match. The transformed 3D object model is then projected into one of the images that were used to reconstruct the 3D scene that was used as search data (see figure 11.5).

```
    dev_display (Img)
    for MatchIndex := 0 to |Scores| - 1 by 1
        rigid_trans_object_model_3d (PipeJointOM3DID, PoseObjInWorld, \
                                     ObjectModel3DRigidTrans)
        project_object_model_3d (ModelContours, ObjectModel3DRigidTrans, \
                                 CamParam0, WorldPose0, ['data', \
                                 'hidden_surface_removal'], ['faces', \
                                 'true'])
        dev_display (ModelContours)
    endfor
```

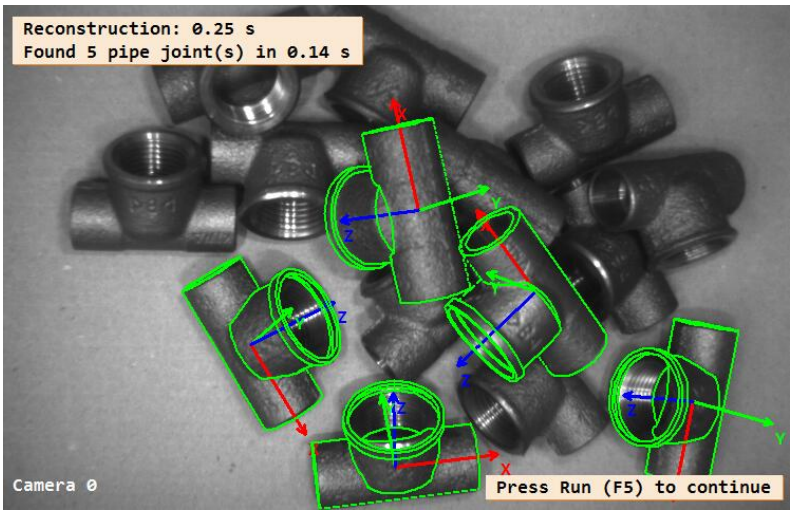


Figure 11.5: Projection of the five best matches into one of the multi-view stereo images.

11.4 Relation to Other Methods

11.4.1 Alternatives to 3D Matching

Matching (calibrated perspective deformable)

If the 3D pose of planar objects or object parts are searched for, the calibrated perspective, deformable matching is a fast alternative to 3D matching. There, no 3D model has to be provided, as only a 2D model is needed, which can easily be obtained from training images. As no 2D projections of a 3D model are computed, the calibrated perspective, deformable matching is significantly faster than the shape-based 3D matching. The calibrated perspective, deformable matching is described in the Solution Guide II-B in section 3.4 on page 91.

Matching (calibrated descriptor-based)

If the 3D pose of planar objects or object parts are searched for and the objects are textured in a way that distinctive points can be extracted, the calibrated descriptor-based matching is even faster than the calibrated perspective, deformable matching, especially for a large search space. On the other side, it is less accurate. The calibrated descriptor-based matching is described in the Solution Guide II-B in section 3.6 on page 108.

Chapter 12

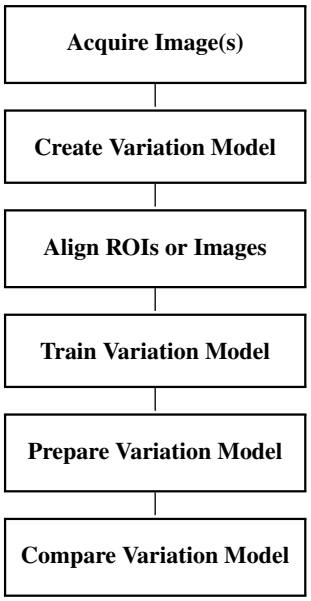
Variation Model

The main principle of a variation model is to compare one or more images to an ideal image to find significant differences. With this, you can, e.g., identify incorrectly manufactured objects by comparing them to correctly manufactured objects. The ideal image is often obtained by a training using several reference images. Besides the ideal image, the training derives information about the allowed gray value variation for each point of the image. This information is stored in a so-called variation image. Both images are used to create a variation model, to which other images can be compared.

The advantage of the variation model is that images can be directly compared by their gray values and the comparison is spatially weighted by the variation image.

12.1 Basic Concept

Variation Model mainly consists of the following parts:



12.1.1 Acquire Image(s)

Both for the training and for the comparison, images are acquired.

For detailed information see the description of this method on page 21.

12.1.2 Create Variation Model

First, you have to create the variation model used for the image comparison by the operator `create_variation_model`. It stores information that is successively added during the following steps.

12.1.3 Align ROIs or Images

For training the variation model, all training images must be placed in the same position and with the same orientation. Thus, before training the model the objects must be aligned. Similarly, the images that are to be compared to the variation model must be aligned.

How to perform alignment using shape-based matching is described in the Solution Guide II-B in section 2.5.3.2 on page 35.

12.1.4 Train Variation Model

The variation model can be trained by providing a set of images containing good samples. The operator `train_variation_model` uses the training images to calculate an ideal image and a corresponding variation image. The variation image represents the amount of gray value variation, i.e., the tolerance, at every point of the image.

If you do not want to train the model with multiple images, e.g., for memory reasons, you can also use a single image as ideal image, but then you need knowledge about the spatial distribution of the variations. If you know, e.g., that the acceptable tolerances are near the edges of your object, you can create the variation image by applying an edge filter to the ideal image. For this proceeding, the training step is skipped.

12.1.5 Prepare Variation Model

To prepare the variation model for the image comparison, the ideal image and the variation image are converted into two threshold images. If you have trained the model with several training images using `train_variation_model`, you prepare the model with the operator `prepare_variation_model`. If you use a single image as ideal image and create the corresponding variation image manually by filtering the ideal image, e.g., using `sobel_amp`, `edges_image`, or `gray_range_rect`, you have to use `prepare_direct_variation_model` for the preparation because the ideal image and the variation image are not yet connected to the variation model.

The obtained threshold images can be directly read out using the operator `get_thresh_images_variation_model`.

12.1.6 Compare Variation Model

You compare an image to the prepared variation model with the operator `compare_variation_model`. There, the two thresholds obtained during the preparation step (and stored in the variation model) are used to determine a region containing all points of the image that significantly differ from the model. Extended parameter settings for the comparison are available when using `compare_ext_variation_model` instead.

12.1.7 A First Example

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Print-Inspection/print_check.hdev`

An example for this basic concept is the following program. Here, the logos on the pen clips depicted in figure 12.1 are inspected using a variation model. The model is trained successively by a set of images containing correct prints. As the pen clips move from image to image, in each image the pen clip is located and aligned by shape-based matching (see also chapter chapter 10 on page 89 for an introduction to shape-based matching and Solution Guide II-B, section 2.5.3.3 on page 38 for further information about alignment using shape-based matching). After the training, the variation model is prepared for the comparison with other images.

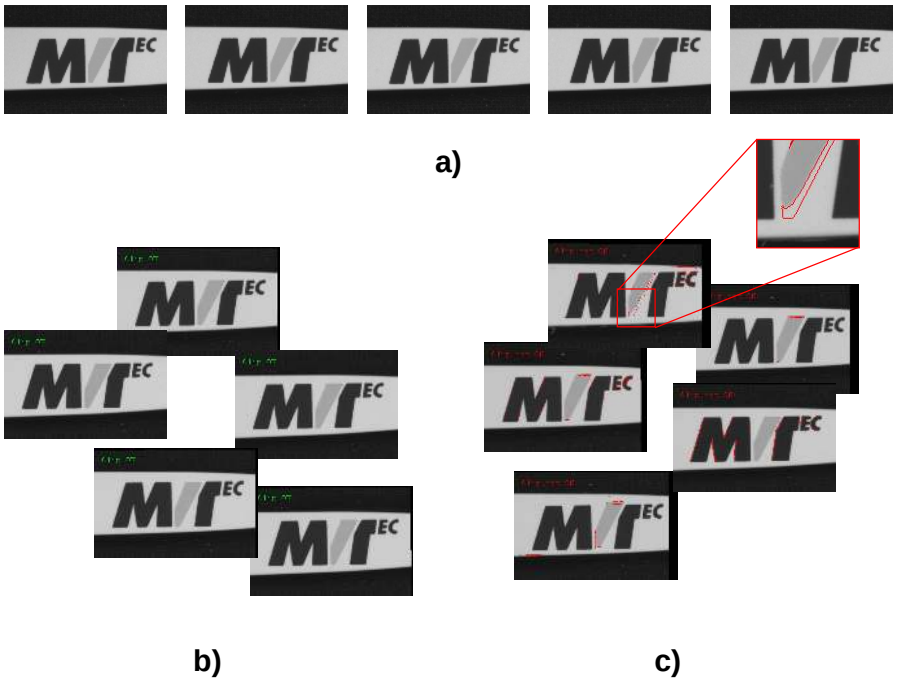


Figure 12.1: Inspection of the logo that is printed on the pen clip: (a) training images; (b) accepted images; (c) rejected images.

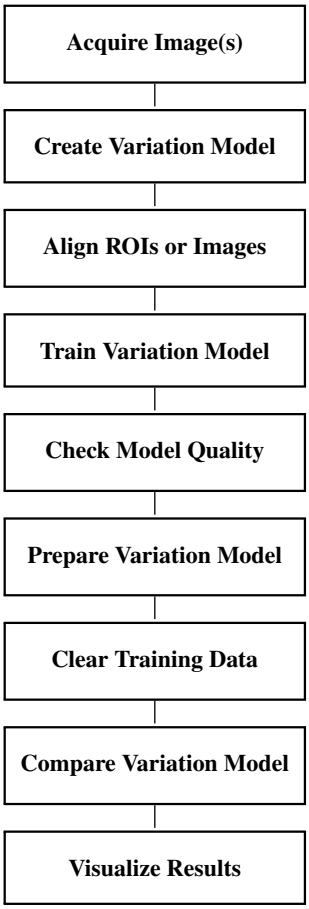
```
create_generic_shape_model (ShapeModelID)
train_generic_shape_model (ImageReduced, ShapeModelID)
find_generic_shape_model (ImageReduced, ShapeModelID, MatchResultID, \
                          Matches)
get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                HomMat2DModel)
create_variation_model (Width, Height, 'byte', 'standard', VariationModelID)
for I := 1 to 15 by 1
    read_image (Image, 'pen/pen-' + I$'02d')
    find_generic_shape_model (Image, ShapeModelID, MatchResultID, \
                              NumMatchResult)
    if (NumMatchResult == 1)
        get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                        HomMat2DMatch)
        hom_mat2d_invert (HomMat2DMatch, HomMat2DMatchInvert)
        hom_mat2d_compose (HomMat2DModel, HomMat2DMatchInvert, \
                           TransformationMatrix)
        affine_trans_image (Image, ImageTrans, TransformationMatrix, \
                            'constant', 'false')
        train_variation_model (ImageTrans, VariationModelID)
    endif
endfor
get_variation_model (MeanImage, VarImage, VariationModelID)
prepare_variation_model (VariationModelID, 20, 3)
```

During the inspection phase, also the images that are to be inspected are aligned by shape-based matching. Then, the images are compared to the variation model to check the print for errors. The erroneous regions are extracted and displayed.

```
find_generic_shape_model (Image, ShapeModelID, MatchResultID, \
                          NumMatchResult)
if (NumMatchResult == 1)
    get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                    HomMat2DMatch)
    hom_mat2d_invert (HomMat2DMatch, HomMat2DMatchInvert)
    hom_mat2d_compose (HomMat2DModel, HomMat2DMatchInvert, \
                       TransformationMatrix)
    affine_trans_image (Image, ImageTrans, TransformationMatrix, 'constant', \
                        'false')
    reduce_domain (ImageTrans, RegionROI, ImageReduced)
    compare_variation_model (ImageReduced, RegionDiff, VariationModelID)
    connection (RegionDiff, ConnectedRegions)
    select_shape (ConnectedRegions, RegionsError, 'area', 'and', 20, \
                  1000000)
endif
```

12.2 Extended Concept

Often more than the essential steps are necessary. In many cases, after the comparison, a visualization of the result is required. Additionally, you might want to check the quality of the training or you need to save memory. The advanced steps are described in the following sections.



12.2.1 Check Model Quality

After training a variation model with several images, the image of the ideal object and the corresponding variation image can be queried by the operator `get_variation_model` to check if the images used for the training all

contained similar objects. If the variation image contains large variations in areas that should exhibit no variations, this leads to the conclusion that at least one of the training images contained a bad object.

12.2.2 Clear Training Data

After the preparation of the variation model, you can reduce the amount of memory needed for the variation model by applying the operator `clear_train_data_variation_model`. But this is only recommended if you do not need to use the variation model for anything else than for the actual comparison anymore. A further training or the application of `get_variation_model` is not possible after deleting the training data.

12.2.3 Visualize Results

A typical visualization task is to display the image and mark the parts of the image that do not correspond to the model. These parts can be explicitly extracted by applying `connection` to separate the connected components of the region obtained by the image comparison and afterwards selecting the regions that are within a specific area range by `select_shape`.

For detailed information see the description of this method on page 223.

12.3 Programming Examples

This section gives a brief introduction to using HALCON for variation model.

12.3.1 Inspect a Printed Logo Using a Single Reference Image

Example: `%HALCONEXAMPLES%/solution_guide/basics/variation_model_single.hdev`

This example inspects the same logo as the first example, but instead of training the variation model with multiple images, a single image is used as ideal image.

From the ideal image a shape model is derived for the later alignment and the variation image is determined manually (see also chapter chapter 10 on page 89 for an introduction to shape-based matching and Solution Guide II-B, section 2.5.3.3 on page 38 for further information about alignment using shape-based matching). To obtain a variation image with tolerances at the edges of the object, the edges in the domain of the corresponding region of interest (`ImageReduced`) are extracted by the operator `edges_sub_pix`. These edges are converted into regions and the regions are enlarged by a dilation to create a generic image that contains the slightly enlarged boundaries of the object. As the conversion of subpixel-precise edges into pixel-precise regions leads to aliasing, the edges are scaled before applying the conversion. After the region processing, the generic image is zoomed back to the original size using a weighting function, which additionally smoothes the borders of the regions. A further smoothing is realized by the operator `binomial_filter`. The image is used now as variation image (see figure 12.2) and the variation model is prepared for the image comparison.

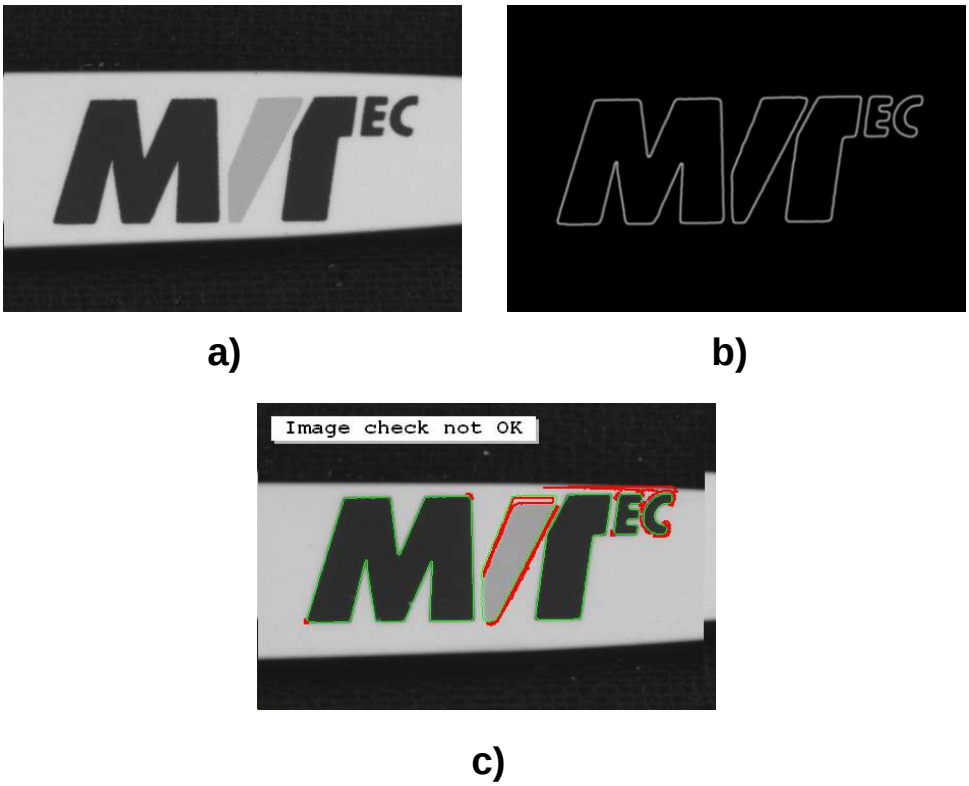


Figure 12.2: Model for the inspection of the pen clips: a) ideal image; b) variation image; c) inspected image with displayed errors.

```
create_generic_shape_model (ShapeModelID)
set_generic_shape_model_param (ShapeModelID, 'contrast_low', 40)
set_generic_shape_model_param (ShapeModelID, 'contrast_high', 50)
train_generic_shape_model (ImageReduced, ShapeModelID)
set_generic_shape_model_param (ShapeModelID, 'min_contrast', 40)
set_generic_shape_model_param (ShapeModelID, 'angle_start', rad(-10))
set_generic_shape_model_param (ShapeModelID, 'angle_end', rad(10))
edges_sub_pix (ImageReduced, Edges, 'sobel_fast', 0.5, 10, 20)
hom_mat2d_identity (HomMat2DIdentity)
hom_mat2d_scale (HomMat2DIdentity, 4, 4, 0, 0, HomMat2DScale)
affine_trans_contour_xld (Edges, ZoomedEdges, HomMat2DScale)
gen_image_const (VarImageBig, 'byte', 4 * Width, 4 * Height)
count_obj (ZoomedEdges, NEdges)
for i := 1 to NEdges by 1
    select_obj (ZoomedEdges, ObjectSelected, i)
    get_contour_xld (ObjectSelected, RowEdge, ColEdge)
    gen_region_polygon (Region1, RowEdge, ColEdge)
    dilation_circle (Region1, RegionDilation, 2.5)
    paint_region (RegionDilation, VarImageBig, VarImageBig, 255, 'fill')
endfor
zoom_image_size (VarImageBig, VarImageSmall, Width, Height, 'weighted')
binomial_filter (VarImageSmall, VarImage, 3, 3)
create_variation_model (Width, Height, 'byte', 'direct', VarModelID)
prepare_direct_variation_model (Image, VarImage, VarModelID, 15, 4)
```

During the inspection, in each image to be checked the boundary of the object is searched for to align the image to the ideal image. The aligned image is then compared to the variation model using compare_ext_variation_model. As the mode absolut is set, you can alternatively use the operator compare_variation_model. Differing regions of a certain size are obtained and stored in NDefects.

```
for i := 1 to 30 by 1
    read_image (Image, 'pen/pen-' + i$'02d')
    find_generic_shape_model (Image, ShapeModelID, MatchResultID, \
                              NumMatchResult)
    if (NumMatchResult != 0)
        get_generic_shape_model_result (MatchResultID, 'best', 'row', Row)
        get_generic_shape_model_result (MatchResultID, 'best', 'column', \
                                        Column)
        get_generic_shape_model_result (MatchResultID, 'best', 'angle', \
                                        Angle)
        vector_angle_to_rigid (Row, Column, Angle, ModelRow, ModelColumn, 0, \
                               HomMat2D)
        affine_trans_image (Image, ImageAffineTrans, HomMat2D, 'constant', \
                            'false')
        reduce_domain (ImageAffineTrans, LogoArea, ImageReduced1)
        compare_ext_variation_model (ImageReduced1, RegionDiff, VarModelID, \
                                     'absolute')
        connection (RegionDiff, ConnectedRegions)
        select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 10, \
                      99999)
        count_obj (SelectedRegions, NDefects)
    endif
endfor
```

12.3.2 Inspect a Printed Logo under Varying Illumination

Example: `%HALCONEXAMPLES%/solution_guide/basics/variation_model_illumination.hdev`

This example inspects caps of bottles using a variation model (see figure 12.3). The difficulty here is the changing illumination during the inspection. Like in the example described before, a single image is used as ideal image.

The variation model here is obtained by filtering the ideal image with the operator `sobel_amp`.

```
create_variation_model (Width, Height, 'byte', 'direct', VariationID)
sobel_amp (ModelImage, VarImage, 'sum_abs', 5)
prepare_direct_variation_model (ModelImage, VarImage, VariationID, [20, 25], \
                                [1.6, 1.6])
```

To compensate for the changing illumination, inside the procedure `get_grayval_range` the gray value range of the variation model is determined.

```
get_grayval_range (ModelImage, RegionROI, RegionForeground, \
                   RegionBackground, BackgroundGVModel, ForegroundGVModel)
```

Inside the procedure the ideal image is reduced to the domain of the region of interest that encloses the print on the cap. Then, the reduced image is split into foreground and background using `binary_threshold` and `difference`. Finally, for both regions the mean and standard deviation of the gray values is queried by the operator `intensity`.

```
reduce_domain (Image, RegionROI, ImageReduced)
binary_threshold (ImageReduced, RegionBackground, 'max_separability', \
                  'dark', UsedThreshold)
difference (RegionROI, RegionBackground, RegionForeground)
intensity (RegionForeground, Image, ForegroundGVal, DeviationFG)
intensity (RegionBackground, Image, BackgroundGVal, DeviationBG)
```

The inspection of the caps is realized inside the procedure `inspect_cap`.

```
    inspect_cap (rImage, RegionROI, WindowHandle, ModelID, VariationID, \
                 RowModel, ColumnModel, BackgroundGVModel, \
                 ForegroundGVModel)
```

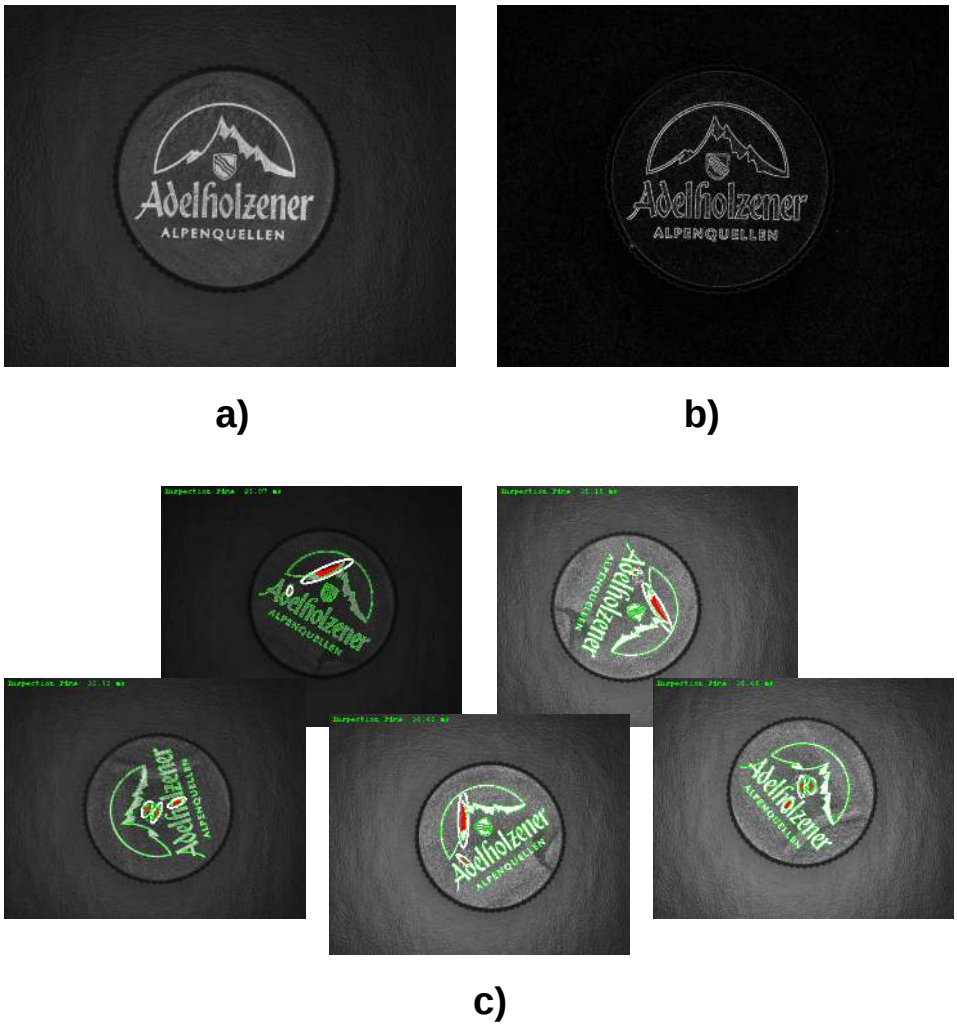


Figure 12.3: Model for the inspection of the caps: a) ideal image; b) variation image; c) results of image comparison for images taken under different illumination conditions.

There, in each image the object is searched for and aligned as described in the previous example. Then, the procedure `get_grayval_range` derives also the gray value range for the ROI of the object to be inspected, so that the image can be adapted to the gray value range of the variation model before comparing it to the variation model. Thus, also images that are taken under varying illumination can be compared. The mode for the image comparison using the operator `compare_ext_variation_model` is set to `light_dark`, so that separate regions for too bright and too dark image points are returned in `RegionDiff`.

```
get_grayval_range (ImageAffineTrans, RegionROI, RegionForegroundImage, \
                   RegionBackgroundImage, BackgroundImage, \
                   ForegroundImage)
Mult := (ForegroundGVModel - BackgroundGVModel) / (ForegroundImage -  \
        BackgroundImage)
Add := ForegroundGVModel - Mult * ForegroundImage
scale_image (ImageReduced, ImageScaled, Mult, Add)
compare_ext_variation_model (ImageScaled, RegionDiff, VariationID, \
                             'light_dark')
```

Chapter 13

Classification

Classification is the technical term for the assignment of objects to individual instances of a set of classes. The objects as well as the available classes are described by specific features, e.g., the color of a pixel or the shape of a region. To define the classes, the features have to be specified, e.g., by a training that is based on known objects. After the training, the classifier compares the features of the object with the features associated to the available classes and returns the class with the largest correspondence. Depending on the selected classifier, possibly additional information about the probabilities of the classes or the confidence of the classification is given.

Generally, two approaches for a classification of image data can be distinguished. One approach is based on a pure pixel classification and segments images based on color or texture. The other approach is more general and classifies arbitrary features, i.e., you can additionally classify regions based on region features like, e.g., shape, size, or color. For the programming examples introduced here, the focus is on the first approach. Note that actually the optical character recognition (OCR) provided by HALCON is a further classification approach, for which specific operators are provided. But these are described in more detail in the chapter OCR on page 183.

HALCON provides different classifiers. The most important classifiers are the neural network (multi-layer perceptron or MLP) classifier, the support vector machine (SVM) classifier, the Gaussian mixture model (GMM) classifier, and the k-nearest neighbor (k-NN) classifier. Additionally, a box classifier is available, but as the GMM classifier leads to comparable results and is more robust, here only the GMM classifier is described. Further, more 'simple' classifiers can be used for image segmentation. These comprise the classifiers for two-dimensional pixel classification with `class_2dim_sup` or `class_2dim_unsup`, and the n-dimensional pixel classification approach based on hyper-spheres (`class_ndim_norm`), which can be used, e.g., for an euclidean classification. These approaches are less flexible, so the focus of this chapter is on the MLP, SVM, GMM, and k-NN classifiers. If you want to use one of the 'simple' classifiers, have a look at the corresponding examples.

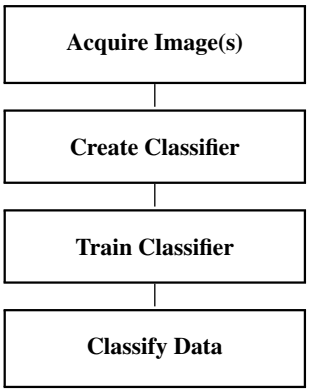
To decide which classifier to use for a specific task and to choose suitable parameters for the selected classification is rather challenging, as each classification task is different and thus needs different handling. HALCON provides the operators to apply a classification, and this chapter provides some general hints about their usage. Further, the most salient advantages and disadvantages of the different classifiers are summarized for the step Create Classifier, and minor differences between the classifiers, which are related to individual operators, are stated with the corresponding step descriptions or in the corresponding sections of the Reference Manual. Nevertheless, in contrast to other methods described in this manual, for classification no set of fixed rules for the selection of the suitable classifier and the selection of the parameters can be given. In many cases, you have to try out different classifiers for your specific task and with your specific training data. Independent on the selected approach, in almost any case you have to play with the parameters to get a satisfying result. Allow extra time for tests and do not despair if your first tests do not immediately lead to a satisfying result. Classification is complex!

13.1 Basic Concept

Classification consists of the following basic steps:

13.1.1 Acquire Image(s)

When classifying images, both for the generation of the training data and for the classification images must be acquired.



For detailed information see the description of this method on page 21.

13.1.2 Create Classifier

The first step of a classification is the creation of a new classifier. Here, you have to decide which classifier to apply for your specific classification task. The main advantages and disadvantages of the four classifiers (assuming an optimal tuning of the parameters) are as follows:

- Multi-Layer Perceptron: The MLP classifier leads to good recognition rates and is rather fast at classification. In exchange, the training is not as fast as for the SVM classification, especially for huge training sets. If the classification is time critical but the training can be applied offline, the MLP approach is a good choice. A rejection class is returned, but in comparison to the GMM classifier, it may be influenced by outliers, so that an additional, explicit training of a rejection class is recommended. Additionally, if you want to add additional training samples, you should not append a second training, but repeat the training with both the old and the new training samples.
- Support Vector Machines: Compared to the MLP classifier, the SVM classifier leads to slightly better recognition rates and is faster at training, especially for huge training sets. Additionally, a training of new training samples can be simply appended to a previous training. In exchange, the classification is not as fast as for the MLP approach. A rejection class is not obtained automatically.
- Gaussian Mixture Models: The advantage of the GMM classification is that, controlled by the parameter settings, a feature vector that does not match to one of the trained classes can be assigned to a rejection class. Additionally, you can apply a second training that is appended to the first training, e.g., in order to add new training samples. The disadvantage of the GMM classifier is that the recognition rates are not as good as the recognition rates obtained by the MLP or SVM approaches. Further, for the feature vector a length of up to 15 features is recommended, whereas for the MLP and SVM classifiers a feature vector with 500 features is still realistic.
- K-Nearest Neighbor: One advantage of the k-NN classifier is that it has only a few and very intuitive parameters. Also, the k-NN classifier works with very little training data. In principle, only a single sample per class is sufficient to get reasonable results. The training of the k-NN classifier is the fastest of all classifiers. That makes the k-NN classifier the first choice for automatic feature selection and quick evaluations. Disadvantages are that the classification is relatively slow and that the classification results are not as good as for the SVM and MLP classifiers.

Dependent on the selected classification approach, you create the classifier using `create_class_mlp`, `create_class_svm`, `create_class_gmm`, or `create_class_knn`.

When creating a classifier, the correct parameter settings are essential for the success of the later classification. These are very specific for the different approaches. To name just a few, the parameter `OutputFunction` should be set to `'softmax'` for an MLP classifier, as the classifier can also be used for regression and not only for classification. Additionally, the value for the parameter `NumHidden` has to be adjusted very carefully. In contrast to that, for the SVM approach, the parameter `KernelType` in most cases should be set to `'rbf'` and the values of the parameters `Nu` and `KernelParam` strongly influence the classification result and thus should be carefully adjusted. Further, different modes and specific algorithms for the preprocessing of the feature vector can be selected. For details, see the corresponding operator descriptions in the Reference Manual and the Solution Guide II-D.

13.1.3 Train Classifier

The training consists of two important steps: First, representative training samples, i.e., a set of feature vectors for each class, are added to the classifier using the operator add_sample_class_mlp, add_sample_class_svm, add_sample_class_gmm, or add_sample_class_knn. If you want to apply a pixel classification, i.e., you want to segment images into regions of different color or texture, you can alternatively add the samples using the operators add_samples_image_class_mlp, add_samples_image_class_svm, add_samples_image_class_gmm, or add_samples_image_class_knn. Then, you can immediately insert a single multi-channel image instead of many feature vectors (one per pixel).

In a second step, the classifier is trained by train_class_mlp, train_class_svm, train_class_gmm, or train_class_knn. Note that complex calculations are performed during the training. Thus, depending on the number of training samples and some approach-specific influences, e.g., the size of the MLP used for an MLP classification, the training can take from a few seconds to several hours.

13.1.4 Classify Data

For the general classification you use the operators classify_class_mlp, classify_class_svm, classify_class_gmm, or classify_class_knn. These use the trained classifier to return the classes that are most probable for the feature vector of the data to classify. In most cases, you need only the best class. Then, you set the parameter Num to 1. In a few cases, e.g., when working with overlapping classes, also the second best class may be of interest (Num=2).

Like for the step of adding samples to the training, for a pixel classification, i.e., if an image is to be segmented into regions of different color or texture classes, special operators are provided. For image segmentation, you can use classify_image_class_mlp, classify_image_class_svm, classify_image_class_gmm, or classify_image_class_knn. Again, these take images as input, instead of individual feature vectors for each pixel, and return regions as output. Note that for images with a maximum of three channels, you can speed up the image segmentation by using a classification that is based on look-up tables (LUT-accelerated classification). For further information, please refer to the Solution Guide II-D in section 6.1.7 on page 70.

Besides the most probable classes, the GMM, MLP, and k-NN classifiers return additional values to evaluate the results. The k-NN classifier returns a very intuitive rating that depends on the used classification method (e.g., the minimal distance from the training sample). The GMM classifier returns the probabilities of the classes, and the MLP classifier returns the confidence of the classification. Note that the probabilities for the GMM classification are relatively precise, whereas for the confidence of the MLP classification outliers are possible because of the way an MLP is calculated. For example, when classifying positions of objects (rows and columns build the feature vectors), the confidence may be high for positions that are far from the training samples and low in case of overlapping classes. For the SVM classification, no information about the confidence of the classification or the probabilities of the returned classes are given.

13.1.5 A First Example

An example for this basic concept is the following program, which segments the image depicted in figure 13.1 into regions of the three classes Sea, Deck, and Walls using a GMM pixel classification. Parts of the image that can not be clearly assigned to one of the specified classes are returned as a rejection class region.

First, the image is read and the sample regions for the classes Sea, Deck, and Walls are defined and listed in the tuple Classes.

```
read_image (Image, 'patras')
gen_rectangle1 (Sea, 10, 10, 120, 270)
gen_rectangle2 (Deck, [170,400], [350,375], [-0.56192,-0.75139], [64,104], \
                [26,11])
union1 (Deck, Deck)
gen_rectangle1 (Walls, 355, 623, 420, 702)
concat_obj (Sea, Deck, Classes)
concat_obj (Classes, Walls, Classes)
```

Then, a new classifier of type Gaussian Mixture Model is created with create_class_gmm. The classifier works for a feature vector with three dimensions (the three channels of the color image) and for three classes (Sea, Deck,

a) b) c)

Figure 13.1: Segmentation of a color image: (a) original image, (b) sample regions for the classes `Sea` (light gray), `Deck` (white), and `Walls` (gray), (c) regions returned by the classification (black marks the rejected regions).

and `Walls`). The tuple containing the regions that specify each class to be trained is assigned as a set of samples to the classifier using the operator `add_samples_image_class_gmm`. The samples are used now by the operator `train_class_gmm` to train the classifier.

```
create_class_gmm (3, 3, [1,10], 'full', 'none', 2, 42, GMMHandle)
add_samples_image_class_gmm (Image, Classes, GMMHandle, 2.0)
train_class_gmm (GMMHandle, 500, 1e-4, 'uniform', 1e-4, Centers, Iter)
```

Then, the whole image is segmented by `classify_image_class_gmm` into regions of the three trained classes and a rejection class, which consists of the parts that can not be assigned clearly to one of the trained classes. A rejection class is returned automatically for GMM and MLP classification. For the MLP approach the rejection class can be influenced by outliers, so it should be created explicitly like shown in the example `%HALCONEXAMPLES%/hdevelop/Segmentation/Classification/classify_image_class_mlp.hdev`.

```
classify_image_class_gmm (Image, ClassRegions, GMMHandle, 0.0001)
```

13.2 Extended Concept

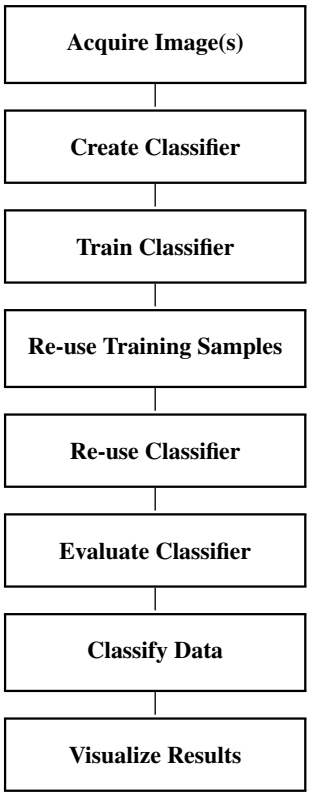
When we look more closely at Classification, further operations can be applied. For example, the training samples or the trained classifier can be stored to file to be re-used in a later step, or the trained classifier can be evaluated further.

13.2.1 Train Classifier

If a classification does not lead to a satisfying result, one way to enhance the recognition rate is to add further training samples to the classifier. For the SVM, GMM, and k-NN you can add new samples to the classifier and train the classifier again, so that only those classes are trained again for which new training samples were added (for details, see the corresponding sections in the Reference Manual). But if the run time of your application is not critical, we recommend to train the classifier anew, using both the old and the new training samples. For the MLP approach a new training should be applied in any case, as the result of appending a second training most likely is not satisfying.

You may want to access the individual training samples if you have the suspicion that one of them is classified incorrectly. To do so, you apply the operators `get_sample_class_mlp`, `get_sample_class_svm`, `get_sample_class_gmm`, or `get_sample_class_knn`. To get the number of available training samples, the operators `get_sample_num_class_mlp`, `get_sample_num_class_svm`, `get_sample_num_class_gmm`, or `get_sample_num_class_knn` can be applied.

If the training and the classification is applied in the same process, you can save memory by applying the operators `clear_samples_class_mlp`, `clear_samples_class_svm`, or `clear_samples_class_gmm` after the training. These clear the training samples from memory. This step is only recommended if you do not want to access the



individual samples in a later step. If the training process and the actual classification is separated (see the step Re-use Classifier), the clearing of the samples is not necessary as the training samples are not stored anyway (unless you store them explicitly, see the step Re-use Training Samples).

For the SVM approach, you can use `reduce_class_svm` to reduce the number of support vectors after the training. The returned classifier is faster than the originally used classifier.

13.2.2 Re-use Training Samples

In many cases it is necessary to access the samples used for the training in a later step, e.g., if you want to repeat the training with some additional samples or if you want to access individual samples to check their correctness. In most applications, you will separate the offline training and the online classification. Thus, in the classification phase the information about the individual samples is not implicitly available anymore. To nevertheless access the information, we recommend to write the used samples to file during the training phase with the operators `write_samples_class_mlp`, `write_samples_class_svm`, or `write_samples_class_gmm`. To read them from file again, you use the operators `read_samples_class_mlp`, `read_samples_class_svm`, or `read_samples_class_gmm`.

13.2.3 Re-use Classifier

In most applications, you will separate the offline training and the online classification. At the end of the training phase, you typically save the classifier to disk by `write_class_mlp`, `write_class_svm`, `write_class_gmm`, or `write_class_knn`. To re-use it for the classification phase, you read the stored classifier by `read_class_mlp`, `read_class_svm`, `read_class_gmm`, or `read_class_knn`.

If the training and the actual classification are separated, in the classification phase no information about the creation and training of the classifier is present anymore. To nevertheless get information about the parameters used for the creation of the classifier, the operators `get_params_class_mlp`, `get_params_class_svm`, `get_params_class_gmm`, or `get_params_class_knn` can be applied. If a preprocessing was used for the feature vectors, which was specified with the creation of the classifier, you can also query the information content of the preprocessed feature vectors of the training samples. For this, you use `get_prep_info_class_mlp`, `get_prep_info_class_svm`, or `get_prep_info_class_gmm`.

13.2.4 Evaluate Classifier

After training a classifier, a feature vector can be evaluated. If you need only its most probable class, or perhaps also the second best class, and the related probabilities or confidences, respectively, these are returned also when applying the actual classification (see Classify Data). But if all probabilities are needed, e.g., to get the distribution of the probabilities for a specific class (see the example description for `%HALCONEXAMPLES%/hdevelop/Classification/Neural-Nets/class_overlap.hdev` in the Programming Examples section), an evaluation is necessary. For the GMM approach, the evaluation with `evaluate_class_gmm` returns three different probability values: the a-posteriori probability (stored in the variable `ClassProb`), the probability density, and the k-sigma error ellipsoid. For the MLP approach, the result of the evaluation of a feature vector with `evaluate_class_mlp` is stored in the variable `Result`. The values of `ClassProb` and `Result` both can be interpreted as probabilities of the classes. The position of the maximum value corresponds to the class of the feature vector, and the corresponding value is the probability of the class. For the SVM classification, no information about the confidence of the classification or the probabilities of the returned classes are given.

13.2.5 Visualize Results

Finally, you might want to display the result of the classification.

For detailed information see the description of this method on page 223.

13.3 Programming Examples

This section gives a brief introduction to using HALCON for classification.

13.3.1 Inspection of Plastic Meshes via Texture Classification

Example: `%HALCONEXAMPLES%/hdevelop/Segmentation/Classification/novelty_detection_svm.hdev`

This example shows how to apply an SVM texture classification in order to detect deviations from the training data (novelties), in this case defects, in a textured material. The image depicted in figure 13.2 shows one of five images used to train the texture of a plastic mesh and the result of the classification for a mesh with defects.

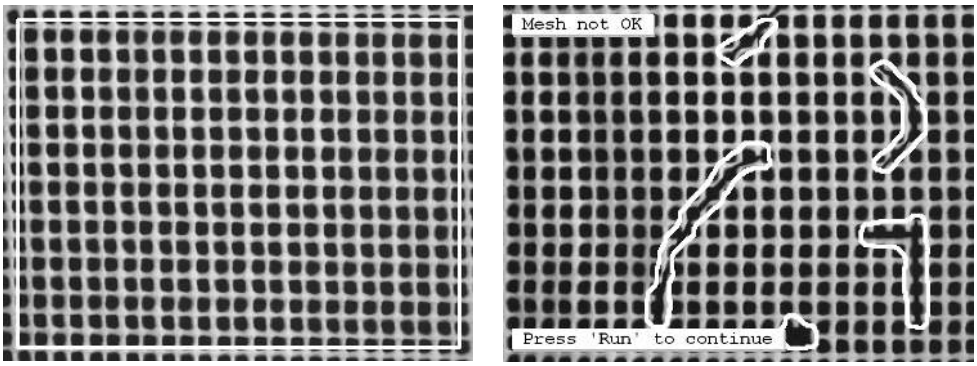


Figure 13.2: Inspection of a plastic mesh: (a) (zoomed) training image with correct texture (region of interest marked in white), (b) mesh with defects (texture novelties marked in white).

The program reads the first training image, defines a region of interest to avoid problems at the image border that would occur because of the later used texture filters and the specific structure of the mesh, and then creates the SVM classifier `SVMHandle` using `create_class_svm`. There, the mode `'novelty-detection'` is selected. With this mode, feature vectors that are not contained in the training samples are classified as a new class. Thus, novelties in the texture of the plastic mesh can be detected. For novelty detection, always the kernel type `'rbf'` has to be selected, but also for other modes it is recommended in most cases.

```
read_image (Image, 'plastic_mesh/plastic_mesh_01')
gen_rectangle1 (Rectangle, 10, 10, Height / 2 - 11, Width / 2 - 11)
create_class_svm (5, 'rbf', 0.01, 0.0005, 1, 'novelty-detection', \
                  'normalization', 5, SVMHandle)
```

Then, all five training images are read in a loop, zoomed, and transformed into texture images by the procedure gen_texture_image.

```
for J := 1 to 5 by 1
    read_image (Image, 'plastic_mesh/plastic_mesh_' + J$'02')
    zoom_image_factor (Image, ImageZoomed, 0.5, 0.5, 'constant')
    gen_texture_image (ImageZoomed, ImageTexture)
```

The procedure creates a texture image as follows: Five different texture filters (texture_laws with different filter types) are applied to the zoomed image and the resulting five images are used to build a five-channel image (compose5). Additionally, the five-channel image is smoothed.

```
texture_laws (Image, ImageEL, 'el', 5, 5)
texture_laws (Image, ImageLE, 'le', 5, 5)
texture_laws (Image, ImageES, 'es', 1, 5)
texture_laws (Image, ImageSE, 'se', 1, 5)
texture_laws (Image, ImageEE, 'ee', 2, 5)
compose5 (ImageEL, ImageLE, ImageES, ImageSE, ImageEE, ImageLaws)
smooth_image (ImageLaws, ImageTexture, 'gauss', 5)
```

After the creation of each texture image, it is added as training sample to the classifier by add_samples_image_class_svm.

```
    add_samples_image_class_svm (ImageTexture, Rectangle, SVMHandle)
endfor
```

After all training samples were added to the classifier, the training is applied by train_class_svm. To enhance the speed of the classification, the number of support vectors is reduced by reduce_class_svm, which leads to a new classifier named SVMHandleReduced.

```
train_class_svm (SVMHandle, 0.001, 'default')
reduce_class_svm (SVMHandle, 'bottom_up', 2, 0.001, SVMHandleReduced)
```

Now, the images that have to be inspected are read in a loop. Each image is again zoomed and transformed into a texture image. The image is reduced to the part described by the region of interest and the reduced image is classified using the operator classify_image_class_svm with the reduced classifier SVMHandleReduced.

```
for J := 1 to 14 by 1
    read_image (Image, 'plastic_mesh/plastic_mesh_' + J$'02')
    zoom_image_factor (Image, ImageZoomed, 0.5, 0.5, 'constant')
    gen_texture_image (ImageZoomed, ImageTexture)
    reduce_domain (ImageTexture, Rectangle, ImageTextureReduced)
    classify_image_class_svm (ImageTextureReduced, Errors, SVMHandleReduced)
```

The regions Errors that are returned by the classification and which describe the classes with the novelties, can now be visualized in the image. For that, morphological operators are applied to smooth the borders of the error regions, and connected components are extracted. Regions of a certain size are selected and the result of the texture inspection is visualized as text. In particular, the texture of the mesh is classified as being good, if the number of selected regions (NumErrors) is 0. Otherwise, the mesh is classified as being erroneous.

```
    opening_circle (Errors, ErrorsOpening, 3.5)
    closing_circle (ErrorsOpening, ErrorsClosing, 10.5)
    connection (ErrorsClosing, ErrorsConnected)
    select_shape (ErrorsConnected, FinalErrors, 'area', 'and', 300, 1000000)
    count_obj (FinalErrors, NumErrors)
    if (NumErrors > 0)
        disp_message (WindowHandle, 'Mesh not OK', 'window', 12, 12, 'red', \
                      'true')
    else
        disp_message (WindowHandle, 'Mesh OK', 'window', 12, 12, \
                      'forest green', 'true')
    endif
    if (J < 14)
        disp_continue_message (WindowHandle, 'black', 'true')
    endif
endfor
```

13.3.2 Classification with Overlapping Classes

Example: `%HALCONEXAMPLES%/hdevelop/Classification/Neural-Nets/class_overlap.hdev`

This artificial example shows how an MLP classification behaves for classes with overlapping feature vectors. Three overlapping classes are defined by 2D feature vectors that consist of the rows and columns of region points. The classes are depicted in figure 13.3.

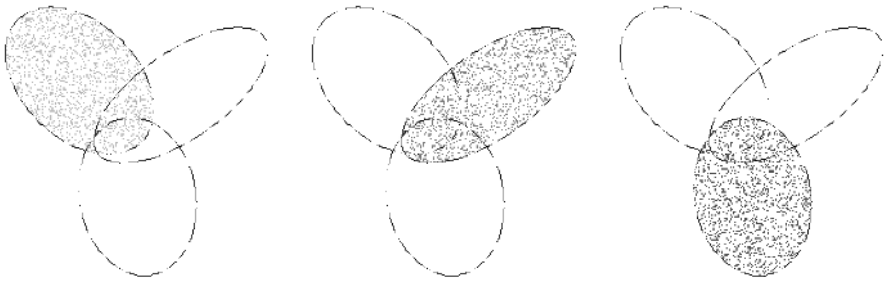


Figure 13.3: Three overlapping classes (feature vectors consist of rows and columns): (a) class1, (b) class 2, (c) class 3.

The classes are created as follows: For each class, an elliptic region is generated. Then, noise is added to a generic image, to which a threshold is applied to get three randomly distributed regions consisting of small dots. These regions are intersected with the elliptic regions of the individual classes. The result is an elliptic cluster of small dots for each class.

```
gen_ellipse (RegionClass1, 60, 60, rad(-45), 60, 40)
gen_ellipse (RegionClass2, 70, 130, rad(-145), 70, 30)
gen_ellipse (RegionClass3, 140, 100, rad(100), 55, 40)
gen_image_const (Image, 'byte', 200, 200)
add_noise_white (Image, ImageNoise1, 60)
add_noise_white (Image, ImageNoise2, 60)
add_noise_white (Image, ImageNoise3, 60)
threshold (ImageNoise1, RegionNoise1, 40, 255)
threshold (ImageNoise2, RegionNoise2, 40, 255)
threshold (ImageNoise3, RegionNoise3, 40, 255)
intersection (RegionClass1, RegionNoise1, SamplesClass1)
intersection (RegionClass2, RegionNoise2, SamplesClass2)
intersection (RegionClass3, RegionNoise3, SamplesClass3)
```

For each elliptic cluster, the positions of the region points are obtained by `get_region_points`. Each position builds a feature vector (`Rows[j]`, `Cols[j]`) that is added to the training by `add_sample_class_mlp`. Then, the classifier is trained with `train_class_mlp`.

```
concat_obj (SamplesClass1, SamplesClass2, Samples)
concat_obj (Samples, SamplesClass3, Samples)
create_class_mlp (2, 5, 3, 'softmax', 'normalization', 1, 42, MLPHandle)
for Class := 0 to 2 by 1
    select_obj (Samples, SamplesClass, Class + 1)
    get_region_points (SamplesClass, Rows, Cols)
    for J := 0 to |Rows| - 1 by 1
        add_sample_class_mlp (MLPHandle, real([Rows[J],Cols[J]]), Class)
    endfor
endfor
train_class_mlp (MLPHandle, 300, 0.01, 0.01, Error, ErrorLog)
```

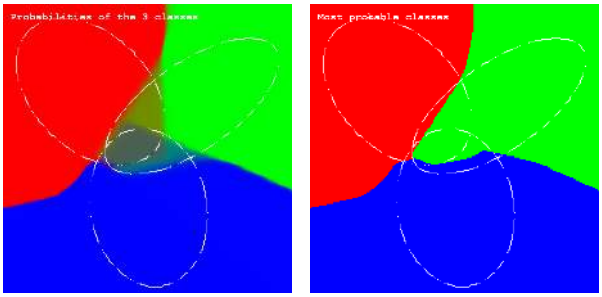
After the training, the classifier is evaluated. To visualize the probabilities of the classes, we create a probability image for each class. In a loop, the operator `evaluate_class_mlp` returns the probabilities for each pixel position. The first value of the returned tuple shows the probability of the pixel to belong to class 1, the second value shows the probability of the pixel to belong to class 2, etc. The probability value of each class is then used as gray value for the corresponding probability image. Additionally, the result of the classification that is returned by `classify_class_mlp`, i.e., the most probable class for each position, is visualized in a probability image. The probability images showing the probabilities of the individual classes are depicted in figure 13.4. In figure 13.5a the probabilities of the three classes are united into a three-channel (color) image with `compose3`. Figure 13.5b shows the result of the classification, i.e., the most probable class for each position in the image.

```
gen_image_const (ProbClass1, 'real', 200, 200)
gen_image_const (ProbClass2, 'real', 200, 200)
gen_image_const (ProbClass3, 'real', 200, 200)
gen_image_const (LabelClass, 'byte', 200, 200)
for R := 0 to 199 by 1
    for C := 0 to 199 by 1
        Features := real([R,C])
        evaluate_class_mlp (MLPHandle, Features, Prob)
        classify_class_mlp (MLPHandle, Features, 1, Class, Confidence)
        set_grayval (ProbClass1, R, C, Prob[0])
        set_grayval (ProbClass2, R, C, Prob[1])
        set_grayval (ProbClass3, R, C, Prob[2])
        set_grayval (LabelClass, R, C, Class)
    endfor
endfor
label_to_region (LabelClass, Classes)
compose3 (ProbClass1, ProbClass2, ProbClass3, Probs)
```

a) b) c)

Figure 13.4: Probabilities for the overlapping classes ("white"=most probable): (a) probability of class 1 (b) probability of class 2, (c) probability of class 3.

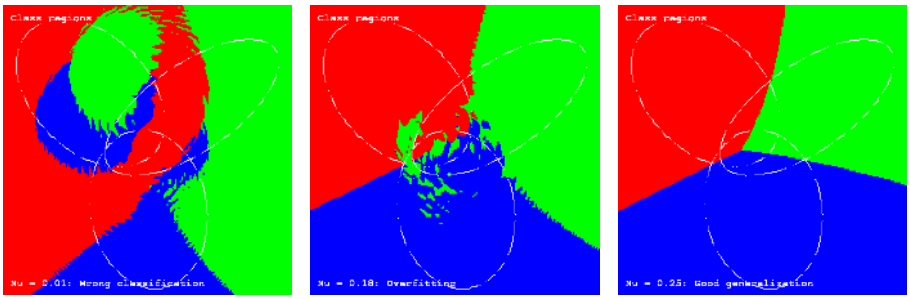
Similar examples, but with GMM and SVM classifiers instead of an MLP classifier are `%HALCONEXAMPLES%/hdevelop/Classification/Gaussian-Mixture-Models/class_overlap_gmm.hdev` and `%HALCONEXAMPLES%/hdevelop/Classification/Support-Vector-Machines/class_overlap_svm.hdev`. For the GMM example, additionally the k-sigma probability is calculated for each feature vector. This can be used to reject all feature vectors that do not belong to one of the three classes. For the SVM example, the importance of selecting suitable parameter values is shown. For SVM



a) b)

Figure 13.5: Probabilities for each pixel: (a) probabilities of the individual classes composed to a three-channel image, (b) confidences returned by the classification.

classification, especially the parameter value `Nu`, which specifies the ratio of outliers in the training data set, has to be selected sufficiently (see figure 13.6).



a) b) c)

Figure 13.6: For SVM classification, the value for the parameter Nu has to be selected carefully: (a) Nu =0.01 (too small, wrong classification), (b) Nu = 0.18 (still too small, overfitting), (c) Nu = 25 (good classification).

13.4 Relation to Other Methods

13.4.1 Methods that are Useful for Classification

Filtering

When applying a pixel classification, a preprocessing, in particular a filtering of the image may be convenient to minimize problems like noise, texture, or overlaid structures. Operators like `mean_image` or `gauss_filter` can be used to eliminate noise. A fast but slightly less perfect alternative to `gauss_filter` is `binomial_filter`. The operator `median_image` is helpful for suppressing small spots or thin lines and gray value morphology can be used to eliminate noise structures.

Region Of Interest (see description on page 25)

The concept of domains (the HALCON term for a region of interest) is useful for classification. Domains are used, e.g., to restrict the classification to the image part that has to be classified. For an overview on how to construct regions of interest and how to combine them with the image see the method Region Of Interest on page 25.

Texture Analysis (see description on page 143)

Texture analysis is a method for finding regular or irregular structures, e.g., on the surface of an object. For this not only the single gray values but also a larger pixel neighborhood is used. HALCON provides filters that emphasize or suppress specific textures. If your application includes objects with a textured surface it might be helpful to apply texture analysis first before segmenting objects using pixel classifiers.

13.4.2 Methods that are Using Classification

Color Processing (see description on page 131)
Classification is used for color processing. There, the feature vector for a pixel consists of the gray values connected to each channel of the image. The classification of the pixels due to their color can then be used, e.g., to segment an image into regions of different color classes. With the description of the color processing method you find the example `%HALCONEXAMPLES%/solution_guide/basics/color_pieces.hdev` which uses an MLP classification to check the completeness of colored game pieces.

OCR (see description on page 183)
Classification is used for optical character recognition (OCR). HALCON provides OCR-specific operators that classify previously segmented regions of images into character classes. The OCR-specific operators are available for the MLP, SVM, k-NN, and box classifier. The latter is not recommended anymore as MLP, SVM, and k-NN are more powerful.

13.4.3 Alternatives to Classification

Blob Analysis (see description on page 33)
Classification is time consuming and rather complex to handle. Thus, it should be applied mainly if other methods fail. One method that may also solve a classification problem is blob analysis. For example, scratches can be separated from a surface by a classification, but in most cases a comparable result can be obtained also by a segmentation of the image via blob analysis, which needs significantly less time and is easy to apply.

Matching (see description on page 89)
If 2D shapes with fixed outlines are to be classified, in many cases it is more convenient to apply a template matching instead of a classification. A classification may be used if not only the shape but also the color or texture of the object is needed to distinguish the objects of different classes.

Variation Model (see description on page 111)
If differences between images and a reference image are searched for, the variation model may be used. In contrast to, e.g., template matching, images can be directly compared by their gray values and the comparison is spatially weighted by the variation image.

13.5 Tips & Tricks

13.5.1 OCR for General Classification

The OCR-specific classification operators are developed for optical character recognition but are not limited to characters. Actually, assuming a suitable training, you can classify any shape of a region with OCR. In some cases, it may be more convenient to classify the shapes of regions using the OCR-specific operators instead of the classification operators described here.

13.6 Advanced Topics

13.6.1 Selection of Training Samples

The quality of a classification depends not only on the selected parameters, but also on the set of training samples used to train the classifier. Thus, for a good classification result, you should carefully select your training samples:

- Use as many training samples as possible.
- Use approximately the same number of samples for each class.
- Take care that the samples of a class show some variations. If you do not have enough samples with variations, you can artificially create more of them by slightly changing your available samples.

Chapter 14

Color Processing

The idea of color processing is to take advantage of the additional information encoded in color or multispectral images. Processing color images can simplify many machine vision tasks and provide solutions to certain problems that are simply not possible in gray value images. In HALCON the following approaches in color processing can be distinguished: First, the individual channels of a color image can be processed using standard methods like blob analysis. In this approach the channels of the original image have to be decomposed first. An optional color space transformation is often helpful in order to access specific properties of a color image. Secondly, HALCON can process the color image as a whole by calling specialized operators, e.g., for pixel classification. Advanced applications of color processing include lines and edges extraction.

a) b)

Figure 14.1: Simple color segmentation.

The example illustrated in figure 14.1 shows how to segment blue pieces of plasticine in a color image.

14.1 Basic Concept

Simple color processing, which is using the methods of blob analysis, mainly consists of three parts:

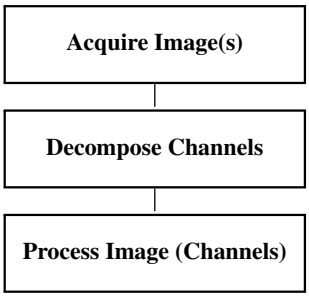
14.1.1 Acquire Image(s)

First, an image is acquired.

For detailed information see the description of this method on page 21.

14.1.2 Decompose Channels

In order to be able to process the individual channels, RGB color images have to be split up into a red, green, and blue channel by using the operator `decompose3`.



14.1.3 Process Image (Channels)

Depending on the application, the individual channels can be processed using standard methods described in this chapter. One of the most frequently used methods is blob analysis (see Blob Analysis on page 33).

14.1.4 A First Example

An example for this basic concept is the following program, which belongs to the example explained above.

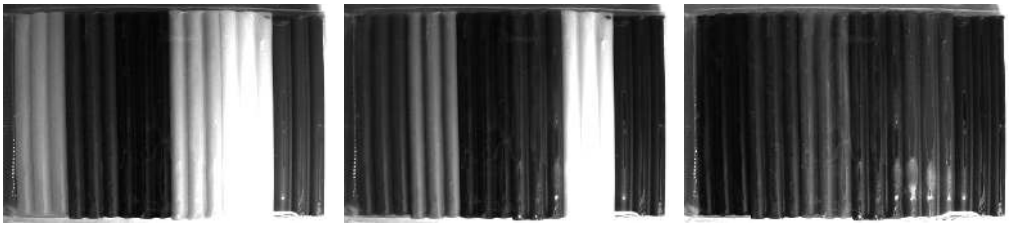


Figure 14.2: Color image decomposed to its red, green, and blue channels.

Here, an RGB image is acquired from file. The image is split into its channels using `decompose3`. The red and green channels are subtracted from the blue channel using `sub_image`. The purpose of this process is to fade out pixels with high values in the other channels, leaving pure blue pixels only. Using `threshold`, the blue pixels with a certain intensity are selected.

```
read_image (Image, 'plasticine')
decompose3 (Image, Red, Green, Blue)
sub_image (Blue, Red, RedRemoved, 1, 0)
sub_image (RedRemoved, Green, RedGreenRemoved, 1, 0)
threshold (RedGreenRemoved, BluePixels, 10, 255)
```

14.2 Extended Concept

In many cases the processing of color images will be more advanced than in the above example. Depending on the actual application, the order of the following steps may vary, or some steps may be omitted altogether.

14.2.1 Demosaick Bayer Pattern

If the acquired image is a Bayer image, it can be converted to RGB using the operator `cfa_to_rgb`. The encoding type of the Bayer pattern (the color of the first two pixels of the first row) must be known (see figure 14.3).

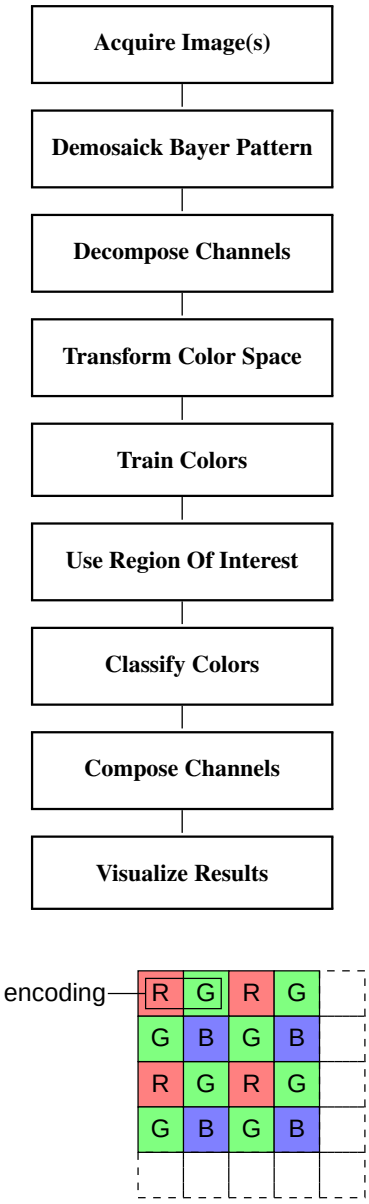


Figure 14.3: Sample Bayer pattern and corresponding encoding.

14.2.2 Transform Color Space

The RGB color space is not always the most appropriate starting point to process color images. If this is the case, a transformation to a different color space might be useful. HALCON supports many important color spaces. Namely, the HSV and HSI color spaces are favorable to select distinct colors independent of their intensity. Therefore, color segmentations in these color spaces are very robust under varying illumination. The i1i2i3 color space qualifies for color classification, whereas the cielab color space is a close match to human perception. Typical operators for color transform are `trans_from_rgb` and `trans_to_rgb`. The operators `create_color_trans_lut` and `apply_color_trans_lut` are, however, faster for time-consuming transformations. How time-consuming a transformation is, depends on both the color space and the used hardware. The example `%HALCONEXAMPLES%/hdevelop/Filters/Color/color_trans_lut.hdev` shows how to use the "faster" operators.

14.2.3 Train Colors

In order to do color classification the colors that need to be distinguished have to be trained. There are different approaches for full color classification including, e.g., Gaussian mixture models (GMM), multilayer perceptron

(MLP), and support vector machine (SVM) classification. For further information about classification, see Classification on page 119.

14.2.4 Use Region Of Interest

Color processing can be sped up by using a region of interest. The more the region in which the segmentation is performed can be restricted, the faster and more robust it will be.

For detailed information see the description of this method on page 25.

14.2.5 Classify Colors

The colors trained in the previous step are used in subsequent images to do the actual classification.

14.2.6 Compose Channels

Any number of channels can be joined to a multi-channel image using the operators `compose2` through `compose7`, or `append_channel`. This way, channels that were processed separately can be composed back to color images for visualization purposes.

14.2.7 Visualize Results

Finally, you might want to display the images, the regions, and the features.

For detailed information see the description of this method on page 223.

14.3 Programming Examples

This section gives a brief introduction to using HALCON for color processing.

14.3.1 Robust Color Extraction

Example: `%HALCONEXAMPLES%/solution_guide/basics/color_simple.hdev`

The object of this example is to segment the yellow cable in a color image in a robust manner.

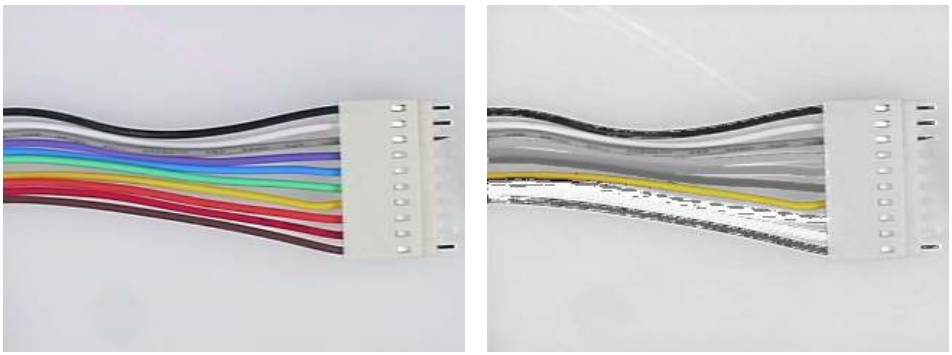


Figure 14.4: Segmentation of a specific color.

Here, an RGB image is acquired from file. The image is split into its channels using `decompose3`. Afterwards, a color space transformation from RGB to HSV is performed using `trans_from_rgb`. This transformation converts the image channels into the separate components hue, saturation and intensity. In the next steps the operator `threshold` selects all pixels with a high saturation value, followed by `reduce_domain` in the hue channel which

effectively filters out pale colors and grays. A histogram of the remaining saturated (vivid) colors is displayed in figure 14.5. Each peak in this histogram corresponds to a distinct color. The corresponding color band is shown below the histogram. Finally, the last `threshold` selects the yellowish pixels.

```
read_image (Image, 'cable' + i)
decompose3 (Image, Red, Green, Blue)
trans_from_rgb (Red, Green, Blue, Hue, Saturation, Intensity, 'hsv')
threshold (Saturation, HighSaturation, 100, 255)
reduce_domain (Hue, HighSaturation, HueHighSaturation)
threshold (HueHighSaturation, Yellow, 20, 50)
```

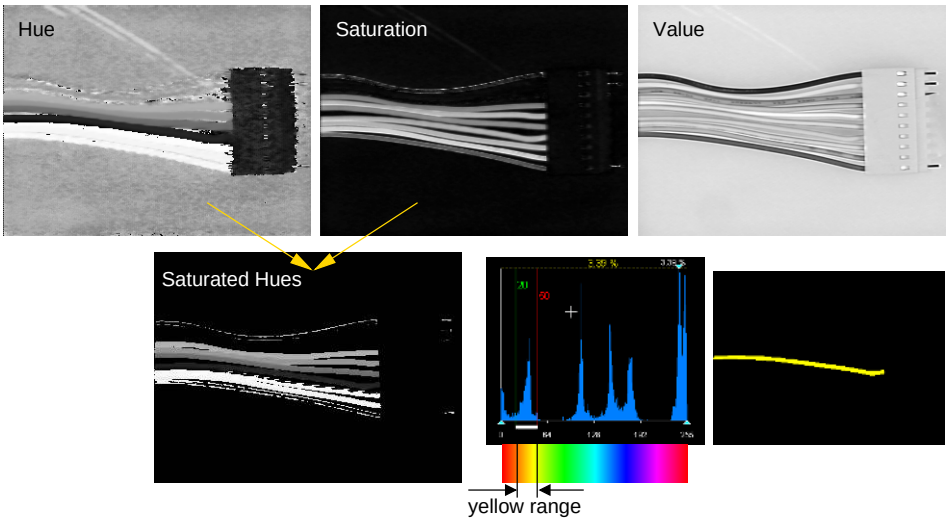


Figure 14.5: Segmentation in HSV color space.

Finding the proper threshold margins is crucial in applications like this. In HDevelop the Gray Histogram tool can be used to determine the values interactively. See the HDevelop User's Guide for more information. To generate the color band shown in figure 14.5, use the following code snippet:

```
gen_image_gray_ramp (Hue, 0, 1, 128, 32, 128, 256, 64)
gen_image_proto (Hue, White, 255)
trans_to_rgb (Hue, White, White, Red, Green, Blue, 'hsv')
compose3 (Red, Green, Blue, MultiChannelImage)
```

14.3.2 Sorting Fuses

Example: `%HALCONEXAMPLES%/solution_guide/basics/color_fuses.hdev`

In this example different types of fuses are classified using color images. The applied method is similar to the previous example. A training image has been used to specify ranges of hue for the fuse types that need to be distinguished. The determined ranges are hard-coded in the program.

```
FuseColors := ['Orange', 'Red', 'Blue', 'Yellow', 'Green']
FuseTypes := [5, 10, 15, 20, 30]
* HueRanges: Orange 10-30, Red 0-10...
HueRanges := [10, 30, 0, 10, 125, 162, 30, 64, 96, 128]
```

A sequence of images is acquired from file, converted to the HSV color space, and reduced to contain only saturated colors just like in the previous example. As already mentioned, color selection in this color space is pretty stable under changing illumination. That is why the hard-coded color ranges are sufficient for a reliable classification. However, it has to be kept in mind that a certain degree of color saturation must be guaranteed for the illustrated method to work.

Figure 14.6: Simple classification of fuses by color with varying illumination.

```
decompose3 (Image, Red, Green, Blue)
trans_from_rgb (Red, Green, Blue, Hue, Saturation, Intensity, 'hsv')
threshold (Saturation, Saturated, 60, 255)
reduce_domain (Hue, Saturated, HueSaturated)
```

The classification iterates over the fuse types and checks for sufficiently large areas in the given hue range. This is done using blob analysis. Afterwards, an additional inner loop labels the detected fuses.

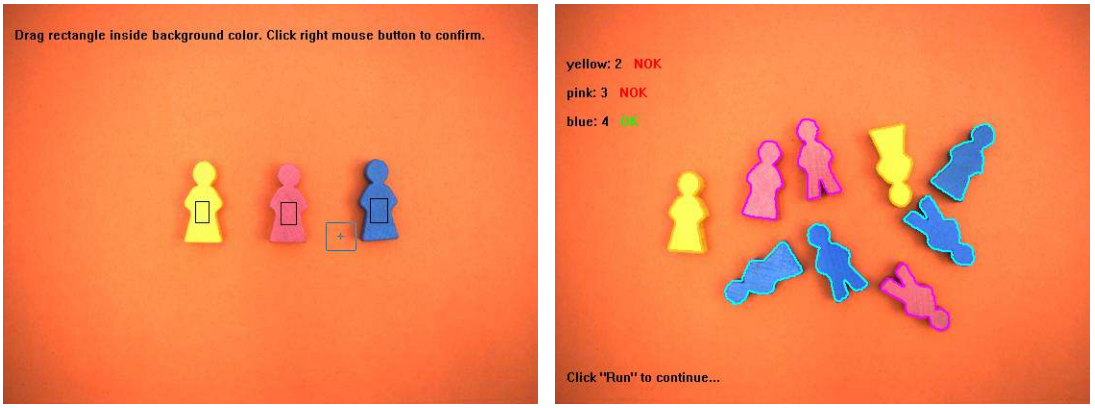
```
for Fuse := 0 to |FuseTypes| - 1 by 1
    threshold (HueSaturated, CurrentFuse, HueRanges[Fuse * 2], \
               HueRanges[Fuse * 2 + 1])
    connection (CurrentFuse, CurrentFuseConn)
    fill_up (CurrentFuseConn, CurrentFuseFill)
    select_shape (CurrentFuseFill, CurrentFuseSel, 'area', 'and', 6000, \
                  20000)
    area_center (CurrentFuseSel, FuseArea, Row1, Column1)
    dev_set_color ('magenta')
    for i := 0 to |FuseArea| - 1 by 1
        set_tposition (WH, Row1[i], Column1[i])
        write_string (WH, \
                      FuseColors[Fuse] + ' ' + FuseTypes[Fuse] + ' Ampere')
    endfor
    set_tposition (WH, 24 * (Fuse + 1), 12)
    dev_set_color ('slate blue')
    write_string (WH, FuseColors[Fuse] + ' Fuses: ' + |FuseArea|)
endfor
stop ()
```

14.3.3 Completeness Check of Colored Game Pieces

Example: `%HALCONEXAMPLES%/solution_guide/basics/color_pieces.hdev`

Completeness checks are very common in machine vision. Usually, packages assembled on a production line have to be inspected for missing items. Before this inspection can be done, the items have to be trained. In the example presented here, a package of game pieces has to be inspected. The game pieces come in three different colors, and the package should contain four of each type. The pieces themselves can be of slightly different shape, so shape-based matching is not an option. The solution to this problem is to classify the game pieces by color. The method applied here is a classification using neural nets (MLP classification).

In the training phase an image is acquired, which contains the different types of game pieces. The task is to specify sample regions for the game pieces and the background using the mouse (see Figure 14.7a). This is accomplished by looping over the `draw_rectangle1` and `gen_rectangle1` operators to draw and create the corresponding regions. The tuple `Classes`, which will be used for the actual training, is extended each time.



(a) (b)

Figure 14.7: Example MLP classification: (a) Training, (b) Result.


```
read_image (Image, ImageRootName + '0')
    for I := 1 to 4 by 1
        dev_display (Image)
        dev_display (Classes)
        disp_message (WindowHandle, \
                      ['Drag rectangle inside ' + Regions[I - 1] + ' color', \
                     'Click right mouse button to confirm'], 'window', \
                      24, 12, 'black', 'false')
        draw_rectangle1 (WindowHandle, Row1, Column1, Row2, Column2)
        gen_rectangle1 (Rectangle, Row1, Column1, Row2, Column2)
        concat_obj (Classes, Rectangle, Classes)
    endfor
```

Once the classes are specified, a multilayer perceptron is created using `create_class_mlp`. With the operator `add_samples_image_class_mlp` the training samples from the image are added to the training data of the multilayer perceptron. The actual training is started with `train_class_mlp`. The duration of the training depends on the complexity and sizes of the training regions.

```
create_class_mlp (3, 7, 4, 'softmax', 'normalization', 3, 42, MLPHandle)
add_samples_image_class_mlp (Image, Classes, MLPHandle)
disp_message (WindowHandle, 'Training...', 'window', 100, 12, 'black', \
              'false')
train_class_mlp (MLPHandle, 400, 0.5, 0.01, Error, ErrorLog)
```

After the training has finished, subsequent images are acquired and classified using `classify_image_class_mlp`. The operator returns a classified region.

```
for J := 0 to 3 by 1
    read_image (Image, ImageRootName + J)
    classify_image_class_mlp (Image, ClassRegions, MLPHandle, 0.5)
```

The returned result is processed further using blob analysis. Each class of the classified region (with the exception of the background class) is accessed using `copy_obj`. The regions of each class are split up using `connection` and reduced to regions of a relevant size (`select_shape`). The remaining few lines of code calculate the number of game pieces found for each class and make a decision whether the result was OK or not.

```
        for Figure := 1 to 3 by 1
            copy_obj (ClassRegions, ObjectsSelected, Figure, 1)
            connection (ObjectsSelected, ConnectedRegions)
            select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', \
                          400, 99999)
            count_obj (SelectedRegions, Number)
            dev_set_color (Highlight[Figure - 1])
            dev_display (SelectedRegions)
            OutString := Regions[Figure - 1] + ': ' + Number + '   '
            dev_set_color ('green')
            disp_message (WindowHandle, OutString, 'window', \
                          24 + 30 * Figure, 12, 'black', 'false')
            if (Number != 4)
                disp_message (WindowHandle, 'Not OK', 'window', \
                              24 + 30 * Figure, 120, 'red', 'false')
            else
                disp_message (WindowHandle, 'OK', 'window', \
                              24 + 30 * Figure, 120, 'green', 'false')
            endif
        endfor
    endfor
```

To illustrate the advantage of using color information, and to compare the classification results, the example program runs an additional training and classification on a converted gray image. As can be seen in figure 14.8 only the yellow region is detected faithfully.

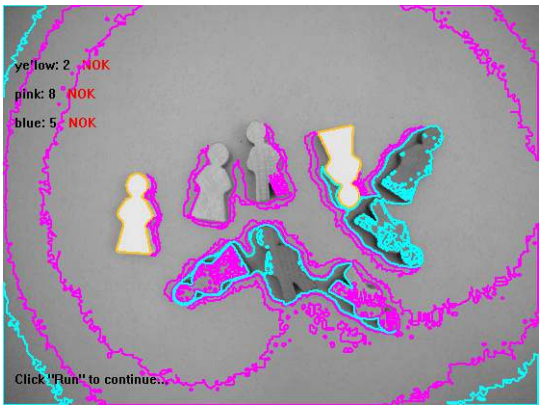


Figure 14.8: Poor classification result when only using the gray scale image.

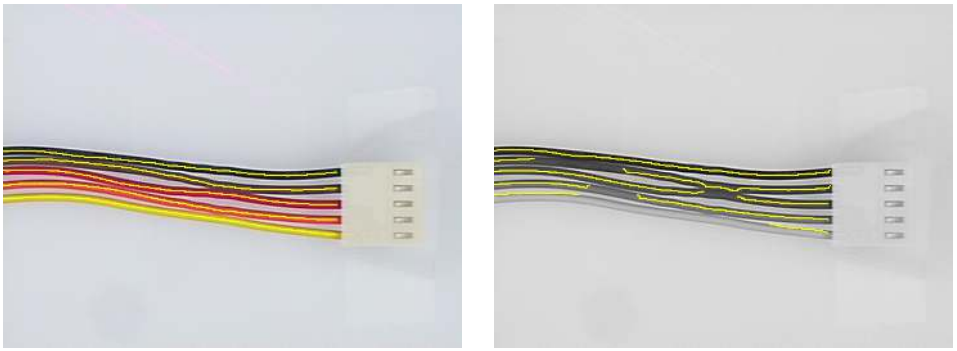
```
        rgb1_to_gray (Image, GrayImage)
        compose3 (GrayImage, GrayImage, GrayImage, Image)
```

For more complex (and time-consuming) classifications, it is recommended to save the training data to the file system using `write_class_mlp`. Later, the saved data can be restored using `read_class_mlp`. As an alternative to the MLP classification you can also apply a classification based on support vector machines (SVM). The corresponding operators are `create_class_svm`, `add_samples_image_class_svm`, `train_class_svm`, `classify_image_class_svm`, `write_class_svm`, and `read_class_svm`. In order to speed up the classification itself, you can apply a segmentation based on look-up tables (for LUT-accelerated classification, please refer to the Solution Guide II-D in section 6.1.7 on page 70) or use a different classification method like Euclidean classification (see `%HALCONEXAMPLES%/solution_guide/basics/color_pieces_euclid.hdev`).

14.3.4 Inspect Power Supply Cables

Example: `%HALCONEXAMPLES%/hdevelop/Filters/Lines/lines_color.hdev`

The task of this example is to locate and inspect the power supply cables depicted in figure 14.9.



a) b)

Figure 14.9: (a) Original color image with cable centers extracted using the color line extractor; (b) corresponding results when using the gray value image.

The input for the program are sample images of colored power supply cables. The task is to extract the centers of each cable together with the width. This is performed using the subpixel-precise color line extractor. To remove irrelevant structures, contours that are too short are removed.

```
lines_color (Image, Lines, 3.5, 0, 12, 'true', 'false')
select_contours_xld (Lines, LongLines, 'contour_length', 450, 100000, 0, \
                     0)
```

The cable width is determined by accessing the line width attribute. For display purposes, a contour is generated for each side.

```
count_obj (LongLines, Number)
gen_empty_obj (EdgesL)
gen_empty_obj (EdgesR)
for K := 1 to Number by 1
    select_obj (LongLines, Line, K)
    get_contour_xld (Line, Row, Col)
    get_contour_attrib_xld (Line, 'angle', Angle)
    get_contour_attrib_xld (Line, 'width_right', WidthR)
    get_contour_attrib_xld (Line, 'width_left', WidthL)
    EdgeRR := Row + cos(Angle) * WidthR
    EdgeRC := Col + sin(Angle) * WidthR
    EdgeLR := Row - cos(Angle) * WidthL
    EdgeLC := Col - sin(Angle) * WidthL
    gen_contour_polygon_xld (EdgeR, EdgeRR, EdgeRC)
    gen_contour_polygon_xld (EdgeL, EdgeLR, EdgeLC)
    concat_obj (EdgesL, EdgeL, EdgesL)
    concat_obj (EdgesR, EdgeR, EdgesR)
endfor
```

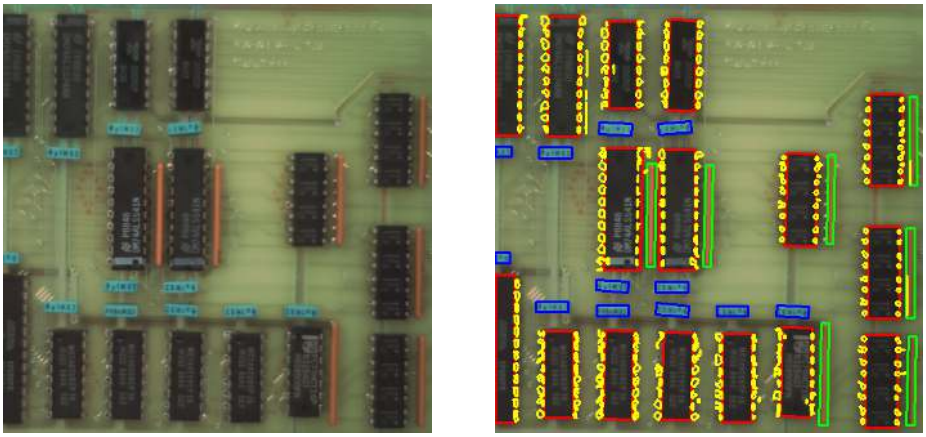
To compare this result with the classical line extraction approach (see Edge Extraction (Subpixel-Precise) on page 63), a line extractor is also applied to the gray value image. The result is depicted in figure 14.9b. Here, it becomes obvious how hard it is to extract the cable using the luminance only.

```
rgb1_to_gray (Image, GrayImage)
lines_gauss (GrayImage, LinesGray, 3.5, 0.0, 0.7, 'dark', 'true', \
             'bar-shaped', 'false')
```

14.3.5 Locating Board Components by Color

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Completeness-Check/ic.hdev`

The task of this example is to locate all components on the printed circuit board depicted in figure 14.10.



a) b)

Figure 14.10: (a) Original image; (b) extracted ICs, resistors, and capacitors.

The input data is a color image, which allows locating components like capacitors and resistors very easily by their significant color: Using a color space transformation, the hue values allow the selection of the corresponding components. The following code extracts the resistors; the extraction of the capacitors is performed along the same lines.

```
decompose3 (Image, Red, Green, Blue)
trans_from_rgb (Red, Green, Blue, Hue, Saturation, Intensity, 'hsv')
threshold (Saturation, Colored, 100, 255)
reduce_domain (Hue, Colored, HueColored)
threshold (HueColored, Red, 10, 19)
connection (Red, RedConnect)
select_shape (RedConnect, RedLarge, 'area', 'and', 150.000000, 99999.000000)
shape_trans (RedLarge, Resistors, 'rectangle2')
```

The extraction of the ICs is more difficult because of the bright imprints, which do not allow a simple thresholding in one step. Instead of this, dark areas are selected first, which are then combined using a dilation.

```
threshold (Intensity, Dark, 0, 50)
dilation_rectangle1 (Dark, DarkDilation, 14, 14)
connection (DarkDilation, ICLarge)
```

After this, the segmentation is repeated inside the thus extracted connected components.

```
add_channels (ICLarge, Intensity, ICLargeGray)
threshold (ICLargeGray, ICDark, 0, 50)
shape_trans (ICDark, IC, 'rectangle2')
```

To locate the contact points, small ROIs are generated on the left and right side of each IC.

```
dilation_rectangle1 (IC, ICDilation, 5, 1)
difference (ICDilation, IC, SearchSpace)
dilation_rectangle1 (SearchSpace, SearchSpaceDilation, 14, 1)
union1 (SearchSpaceDilation, SearchSpaceUnion)
```

Inside these areas, locally bright spots are detected.

```
reduce_domain (Intensity, SearchSpaceUnion, SearchGray)
mean_image (SearchGray, SearchMean, 15, 15)
dyn_threshold (SearchGray, SearchMean, PinsRaw, 5.000000, 'light')
connection (PinsRaw, PinsConnect)
fill_up (PinsConnect, PinsFilled)
select_shape (PinsFilled, Pins, 'area', 'and', 10, 100)
```

14.4 Tips & Tricks

14.4.1 Speed Up

Many online applications require maximum speed. Because of its flexibility, HALCON offers many ways to achieve this goal. Here, the most common methods are listed.

- Of the color processing approaches discussed in this section, the simplest is also the fastest (decompose color image and apply blob analysis). If you want to do color classification, you should consider using `class_ndim_norm` or `class_2dim_sup` for maximum performance.
- Regions of interest are the standard method to increase the speed by processing only those areas where objects need to be inspected. This can be done using pre-defined regions but also by an online generation of the regions of interest that depend on other objects found in the image.
- By default, HALCON performs some data consistency checks. These can be switched off using `set_check`.
- By default, HALCON initializes new images. Using `set_system` with the parameter `"init_new_image"`, this behavior can be changed.

14.5 Advanced Topics

14.5.1 Color Edge Extraction

Similar to the operator `edges_image` for gray value images (see Edge Extraction Using Edge Filters on page 55), the operator `edges_color` can be applied to find pixel-precise edges in color images. For a subpixel-precise edge extraction in color images, the operator `edges_color_sub_pix` is provided. It corresponds to the gray value based operator `edges_sub_pix` (see Edge Extraction (Subpixel-Precise) on page 63). Processing the detected color edges is the same as for gray value images (see Contour Processing on page 79).

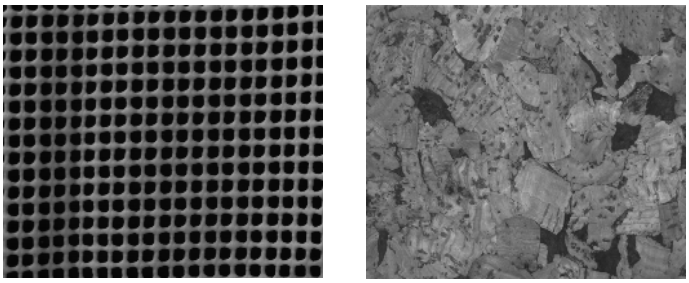
14.5.2 Color Line Extraction

Similar to color edge extraction, HALCON supports the detection of lines in color images using `lines_color`. For processing the detected lines, please refer to Edge Extraction (Subpixel-Precise) on page 63 and Contour Processing on page 79.

Chapter 15

Texture Analysis

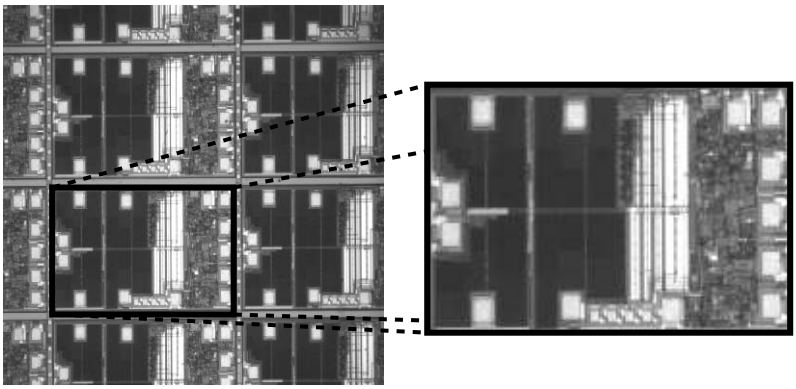
The concept of texture analysis is based on the use of an object's structural features. In order to thoroughly understand the suitable application of texture analysis, it is important to first comprehend texture itself. Texture designates the structural characteristics of a surface which present themselves as gray value variations in an image. The regularity of a texture varies from absolutely regular to irregular. The basis of a regular texture is an ideal grid structure whereas irregular textures or statistical structures are based upon a random function.



a) b)

Figure 15.1: a) Regular texture; b) irregular texture.

Textures are composed of so-called texels, a short form for texture elements, which can easily be recognized in a regular texture. Irregular textures also have recurring elements, which are, however, more difficult to recognize. When working with irregular textures the measures of a texel have to be approximated if necessary. A texel is the smallest recurring structure as depicted in figure 15.2.



a) b)

Figure 15.2: Texture elements of extremely regular texture: a) original image; b) texel.

Texture analysis is a very useful method for all tasks which cannot simply be solved as their gray value structure is too complex. It can help you to solve two tasks:

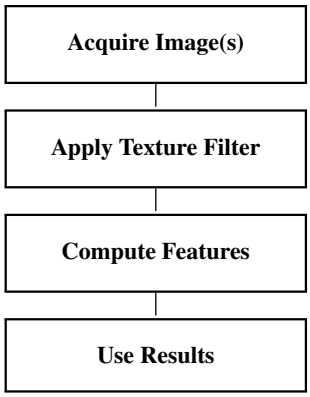
- Where is a certain texture located?
- What kind of texture is it?

The first task is typically solved by filtering the image with filters that enhance or suppress certain textures. The filtered image is then segmented into regions with similar texture, either with standard blob analysis or by classifying each pixel. The second task is typically solved by computing texture features for the whole image or for individual regions. Based on these features, the image or region is then classified. As a preprocessing, texture filters can be applied to enhance or suppress certain textures.

Alternatively to this 'traditional' approach, you can use a texture inspection model to analyze the texture. The texture inspection model involves a training process which extracts texture features automatically. For more information, please refer to the chapter "Inspection ▷ Texture Inspection" of the Reference Manual.

15.1 Basic Concept

A simple texture analysis consists of the following steps:



15.1.1 Acquire Image(s)

First, an image is acquired.

For detailed information see the description of this method on page 21.

15.1.2 Apply Texture Filter

With a texture filter, specific textural structures can be emphasized or suppressed. HALCON's standard texture filter is `texture_laws`.

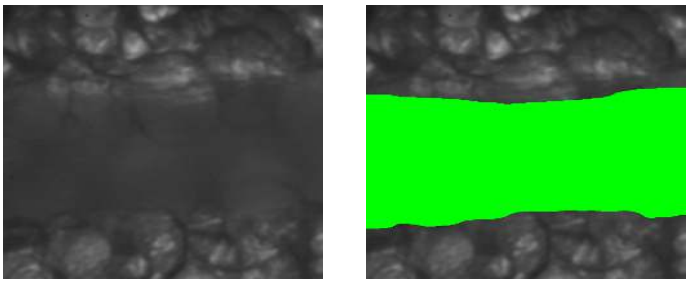
15.1.3 Compute Features

The standard HALCON operator for computing texture features is `gen_cooc_matrix`, which generates the cooccurrence matrix of an image. Other feature related operators are `entropy_gray` for calculating the entropy or the anisotropy of an image or region. These calculations are computed on the original image. To determine the amount of a feature, the intensity, with a threshold, the task is performed on the image after filtering.

15.1.4 A First Example

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Measuring-2D/vessel.hdev`

The example for this basic concept clarifies how texture analysis allows the segmentation of textured images which are impossible to segment using classical segmentation methods like thresholding.



a) b)

Figure 15.3: a) Original image; b) segmented vessel.

The example illustrated in figure 15.3 shows how to segment a blood vessel in a gray value image using texture analysis.

In the gray value image a), a blood vessel has to be segmented and measured. In order to perform this task, `texture_laws` enhances the structure of the texture by extracting structural elements of a certain direction as well as their frequency. Then `mean_image` with the filter size of one texture element, smoothes the image. `binary_threshold` is then able to segment the vessel. Now that the vessel is segmented from its environment in image b), it can easily be measured.

```
read_image (Image, 'vessel')
texture_laws (Image, Texture, 'el', 2, 5)
mean_image (Texture, Energy, 211, 61)
binary_threshold (Energy, Vessel, 'smooth_histo', 'dark', UsedThreshold)
```

15.2 Extended Concept

It will, however, sometimes be necessary to perform more advanced steps during texture analysis in order to achieve a state where the image can be, depending on the kind of analysis undertaken, segmented or classified.

15.2.1 Rectify Image(s)

If the camera looks at an angle onto the object to inspect, it may be necessary to rectify the image to remove lens distortion and perspective distortion.

Detailed information about rectifying images can be found in the Solution Guide III-C in section 3.4 on page 80.

15.2.2 Scale Down Image(s)

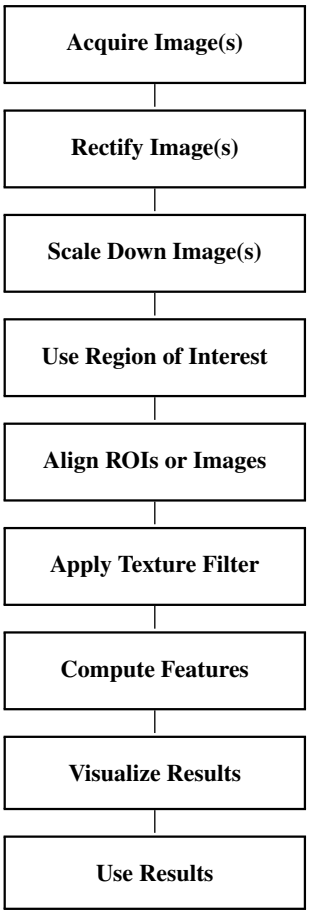
When both the texel and its smallest substructure are relatively large, you can downscale the image with the operator `zoom_image_factor` to speed up the texture analysis.

You can also create a pyramid of downscaled images by calling `zoom_image_factor` multiple times and then apply the texture filtering or feature computation to the image pyramid. Thus, texture analysis is performed at multiple scales as is shown in example `classify_wood.hdev` on page 150.

15.2.3 Use Region of Interest

A region of interest can be created to reduce the image domain for which the texture analysis will be performed. This will reduce the processing time. In particular when computing texture features, the region of interest can be the result of a previous segmentation.

For detailed information see the description of this method on page 25.



15.2.4 Align ROIs or Images

If you are dealing with a regular texture, you can exploit this fact by rotating the image such that the texture appears aligned to the image axes. Then, both texture filtering and texture features can be applied or computed more specifically.

How to perform alignment using shape-based matching is described in the Solution Guide II-B in section 2.5.3.2 on page 35.

15.2.5 Apply Texture Filter

The operator `texture_laws` actually provides a family of texture filters. More information can be found in section 15.7 on page 158. After calling `texture_laws`, you should smooth the image by using `mean_image` with the mask size set to the size of the texel.

For certain images with simple textures, the operator `dyn_threshold` can be a good alternative to a regular filter as it is faster. Before using this operator, you must call `mean_image` with the mask size set to twice the texel size.

15.2.6 Compute Features

If you want to use the cooccurrence matrix corresponding to different directions, it is more efficient to generate the matrix with `gen_cooc_matrix` and then compute the features with `cooc_feature_matrix`.

Besides the texture features based on the cooccurrence matrix, you can also use the entropy and anisotropy of an image, which are computed by the operator `entropy_gray`, respectively. After the filtering has been performed, the intensity of a feature in the image can be computed using a threshold.

All operators can compute the texture features either for the complete image or for image regions. More information about texture features can be found in section 15.6 on page 153.

As an advanced texture feature, you can also compute the distribution of the image gradient, which shows how the gray values change by first filtering the image with `sobel_amp` and then computing the histogram with `gray_histo_abs`.

15.2.7 Visualize Results

To check the results of texture analysis, you might want to display the images, the regions, and the features. To visualize the cooccurrence matrix, you must compute the matrix and its features in separate steps using `gen_cooc_matrix` and `cooc_feature_matrix`.

Often, details in the texture images and the cooccurrence matrix do not show up clearly in the original gray value image. In such a case, you can try to change the look-up table (LUT).

For detailed information see the description of this method on page 223.

15.2.8 Use Results

The filtered image is typically used as input for segmentation tasks, either by Blob Analysis on page 33 or by (pixel) Classification on page 119. Described examples are `vessel.hdev` on page 144, `novelty_detection_svm.hdev`, and `novelty_detection_dyn_threshold.hdev`. The computed texture features are typically used as input for general Classification on page 119. A described example is `classify_wood.hdev` on page 150.

15.3 Programming Examples

15.3.1 Detect Defects in a Texture with Novelty Detection

Example: `%HALCONEXAMPLES%/hdevelop/Segmentation/Classification/novelty_detection_svm.hdev`

A detailed description of this example can be found in the chapter Classification on page 119.

The task of this example is to detect defects in a texture that do not correspond to the texture of trained good objects. For this, it uses the novelty detection mode of the SVM classifier. In the example `novelty_detection_dyn_threshold.hdev`, which is described in the next section, the same task is solved with the operator `dyn_threshold`.

The raw errors returned by the SVM are postprocessed to remove insignificant parts of the detected errors. The texture image is generated in a procedure: It is a five-channel image that contains the result of applying five different Laws filters, which basically correspond to first and second derivatives, and smoothing them sufficiently. Laws filters, included in the HALCON operator `texture_laws`, are very useful whenever it is necessary to enhance certain structures in a texture as becomes clear in figure 15.4.

```
texture_laws (Image, ImageEL, 'el', 5, 5)
texture_laws (Image, ImageLE, 'le', 5, 5)
texture_laws (Image, ImageES, 'es', 1, 5)
texture_laws (Image, ImageSE, 'se', 1, 5)
texture_laws (Image, ImageEE, 'ee', 2, 5)
compose5 (ImageEL, ImageLE, ImageES, ImageSE, ImageEE, ImageLaws)
smooth_image (ImageLaws, ImageTexture, 'gauss', 5)
```

The figure 15.4 shows a plastic mesh without defects on the left side and one that is damaged on the right side. Starting with the original image, first the Laws filter is applied which enhances horizontal structures, followed by the Laws filter le, stressing vertical structures. The damages in the filtered images d) to f) are quite clearly visible. For more information about the Laws filter read in section 15.7 on page 158 later in this chapter.

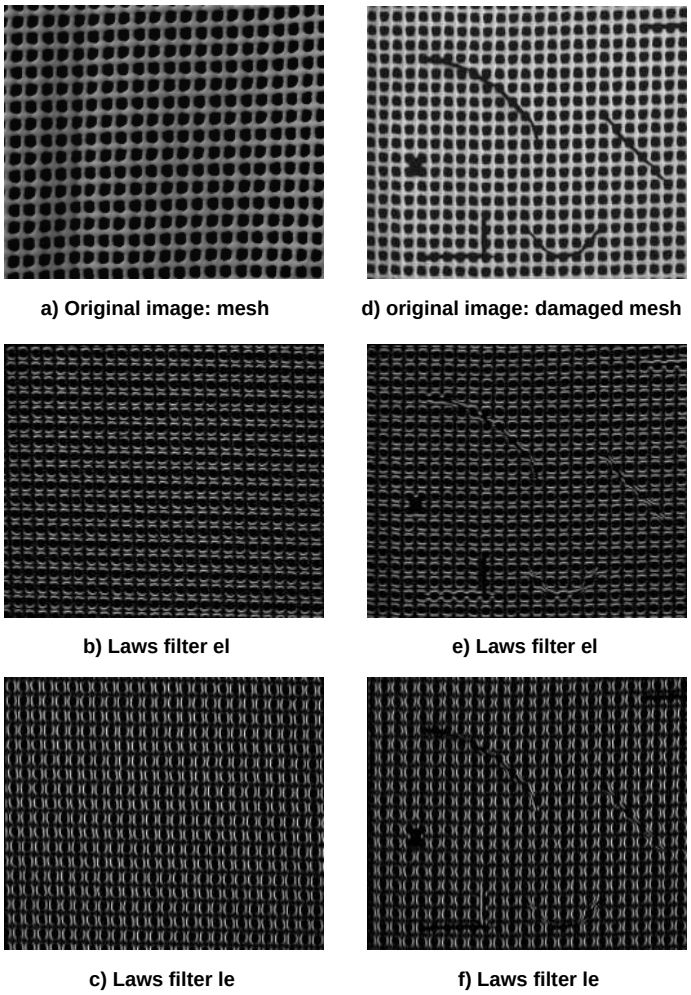


Figure 15.4: Images a) to c) mesh without defects; images d) to f) damaged mesh.

15.3.2 Detect Defects in a Web Using Dynamic Thresholding

Example: `%HALCONEXAMPLES%/solution_guide/basics/novelty_detection_dyn_threshold.hdev`

The task of this example is also to detect defects in a texture that do not correspond to the texture of trained good objects similar to the example `novelty_detection_svm.hdev` on page 124.

The task of this example is to detect defects in a mesh using the operator `dyn_threshold`. In this way, the operator can be used to find textures that differ from the rest of the image. `dyn_threshold` is an operator that is easy to handle. Together with a smoothing filter like `mean_image`, it can extract objects that differ locally from their neighborhood. Note that the operator `dyn_threshold` can only be used successfully for texture analysis if the defects can be detected as darker or brighter then the rest of the texture.

First, the image is smoothed using the HALCON operator mean_image. The mash size determines the size of the extracted objects: The larger the mask size is chosen, the larger the found regions become. As a rule of thumb, the mask size should be about twice the diameter of the objects to be extracted. Subsequently, dyn_threshold is performed and connected regions are looked for. The parameter 'area' of the operator select_shape makes it possible to find regions that differ in size, e.g., that are too small. Found errors are finally counted and displayed.

```
read_image (Image, 'plastic_mesh/plastic_mesh_' + J$'02')
mean_image (Image, ImageMean, 49, 49)
dyn_threshold (Image, ImageMean, RegionDynThresh, 5, 'dark')
connection (RegionDynThresh, ConnectedRegions)
select_shape (ConnectedRegions, ErrorRegions, 'area', 'and', 500, 99999)
count_obj (ErrorRegions, NumErrors)
```

If the number of the errors exceeds zero, the message 'Mesh not OK' is displayed. Otherwise the web is undamaged and 'Mesh OK' appears as is shown in figure 15.5.

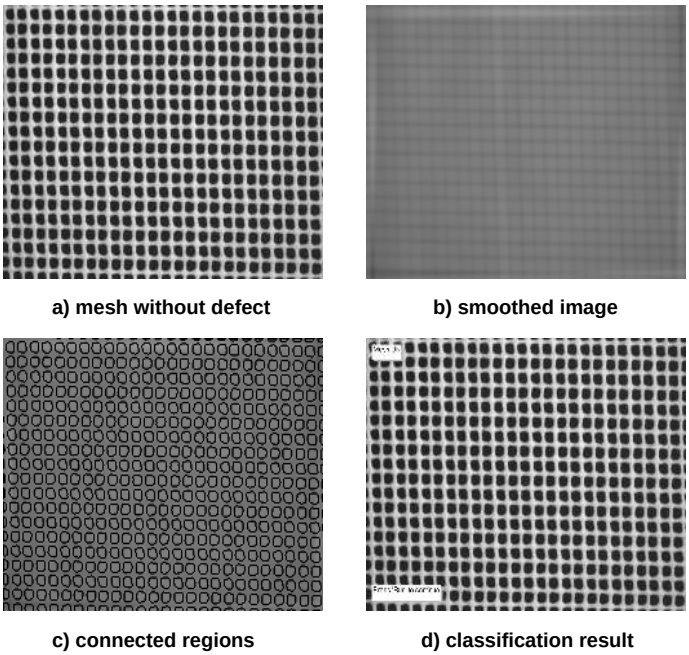


Figure 15.5: a) Faultless mesh b) smoothed image c) connected regions d) classification: Mesh OK.

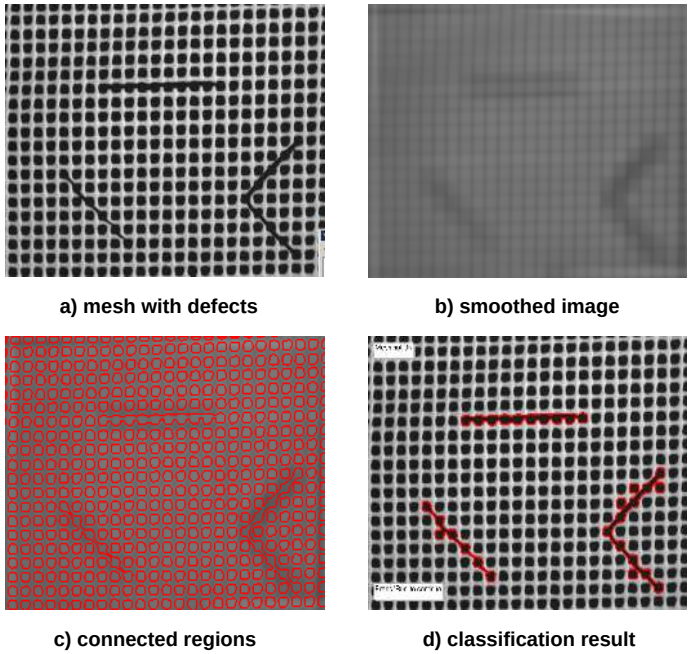


Figure 15.6: a) Defective mesh b) smoothed image c) connected regions d) classification: Mesh not OK.

15.3.3 Classification of Different Types of Wood

Example: `%HALCONEXAMPLES%/solution_guide/basics/classify_wood.hdev`

The objective of this example is to classify different types of wood according to their surface texture.

First the different classes of wood, as shown in figure 15.7, are specified.

```
Classes := ['apple', 'beech', 'cherry', 'maple', 'oak', 'walnut']
```

Then, the training is performed, which means that several images are read and trained for each type of wood. Therefore, the SVM learns different wood textures and compares each new wood image to the existing images. It checks which class is most similar and displays the assigned class.

For each image, features are computed in a procedure and then passed to the operator `classify_image_class_mlp`.

```
gen_features (Image, FeatureVector)
classify_class_mlp (MLPHandle, FeatureVector, 2, FoundClassIDs, Confidence)
```

The procedure gen_features calls the actual feature extraction procedure and then downsamples the image and calls the second procedure with the smaller image again.

```
procedure gen_features (Image, FeatureVector)
gen_sobel_features (Image, FeatureVector, FeatureVector)
zoom_image_factor (Image, Zoomed1, 0.5, 0.5, 'constant')
gen_sobel_features (Zoomed1, FeatureVector, FeatureVector)
```

The procedure gen_sobel_features calculates multiple texture features. First the cooccurrence matrix is computed for the directions 0 and 90 degrees.

```
procedure gen_sobel_features (Image, Features, FeaturesExtended)
* Coocurrence matrix for 90 deg:
cooc_feature_image (Image, Image, 6, 90, Energy, Correlation, Homogeneity, \
                    Contrast)
```

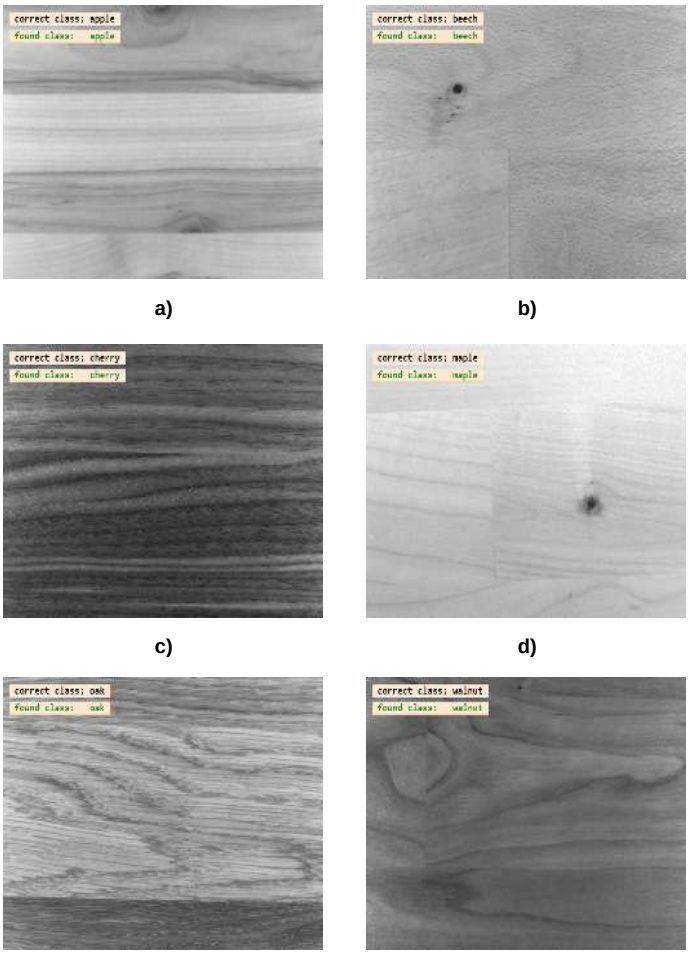


Figure 15.7: Result of wood classification: a) apple, b) beech, c) cherry, d) maple, e) oak, f) walnut .

Furthermore, within the procedure the gray value edges are extracted with the operator `sobel_amp`. The bigger the gray value difference between neighboring pixels is, the brighter these areas are shown in the resulting image. This way, the filtering enables to find and highlight structures in an image. As a feature derived from the sobel image, the absolute gray value histogram of the edge amplitude image is computed.

```
sobel_amp (Image, EdgeAmplitude, 'sum_abs', 3)
gray_histo_abs (EdgeAmplitude, EdgeAmplitude, 8, AbsoluteHistoEdgeAmplitude)
```

All calculated features are appended to the input feature vector and returned in the output feature vector.

```
FeaturesExtended := [Features,Energy,Correlation,Homogeneity,Contrast]
FeaturesExtended := [FeaturesExtended,AbsoluteHistoEdgeAmplitude]
```

As already noted, the texture features are computed on multiple pyramid levels by zooming the images with a factor of 0.5 for each level. This makes it possible to analyze a bigger neighborhood. In the example, only 2 pyramid levels are used; to use more for a more complex task, you can activate the corresponding lines. Note, however, that you must then train the classifier anew.

```
zoom_image_factor (Image, Zoomed1, 0.5, 0.5, 'constant')
gen_sobel_features (Zoomed1, FeatureVector, FeatureVector)
* zoom_image_factor (Zoomed1, Zoomed2, 0.5, 0.5, 'constant')
* gen_sobel_features (Zoomed2, FeatureVector, FeatureVector)
FeatureVector := real(FeatureVector)
```

Similarly, the procedure gen_sobel_features contains deactivated code to compute more features: the entropy, anisotropy, and the absolute gray value histogram of the image.

```
* entropy_gray (Image, Image, Entropy, Anisotropy)
* gray_histo_abs (Image, Image, 8, AbsoluteHistoImage)
* FeaturesExtended := [FeaturesExtended,Entropy,Anisotropy]
* FeaturesExtended := [FeaturesExtended,AbsoluteHistoImage]
```

Note again that if you use more features you must train the classifier anew.

15.4 Relation to Other Methods

15.4.1 Methods that are Using Texture Analysis

Blob Analysis (see description on page 33)
Blob analysis identifies pixels of relevant objects (also called foreground) by their gray value in an image. Whenever objects cannot simply be segmented by separating dark pixels from bright pixels, those objects can possibly be identified by their texture. Therefore, it is useful to perform a texture analysis first and then use the results to segment the objects.

Classification (see description on page 119)
Classification is the technical term for the assignment of objects to individual instances of a set of classes and can be a useful method to follow the texture analysis if the object is characterized by a particular texture. To define the classes, this texture has to be specified, e.g., by a training that is based on known objects. After the training, the classifier compares the features of the object with the features associated to the available classes and returns the class with the largest correspondence.

15.5 Advanced Topics

15.5.1 Fast Fourier Transform (FFT)

Fast Fourier transform, short FFT, is another option for texture analysis. This method makes use of the different frequencies in textures. HALCON's FFT exactly determines the amount of each frequency in the image. It enables the construction of any type of linear filters which means that application-oriented filter designs are possible. There is also a huge set of predefined filters as well as advanced Gabor texture filters. With HALCON's FFT it is possible, e.g., to emphasize any structure of a special texture (see figure 15.8). A convolution can be performed by transforming the image and the filter into the frequency domain, multiplying the two results, and transforming the result back into the spatial domain. Therefore the two reasons for using FFT are:

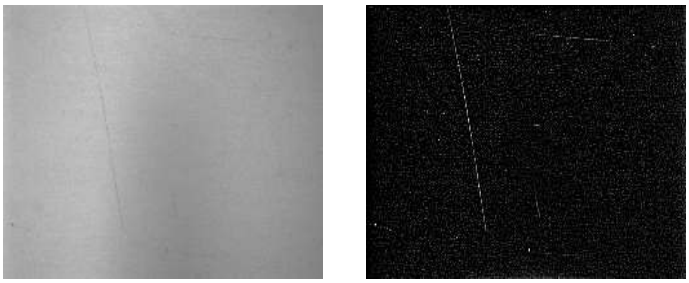
- For bigger filter sizes it is more efficient to apply the filters in Fourier domain as FFT always takes the same time for any mask size, unlike normal filters.
- It is a great advantage that filters can be customized to remove or emphasize specific frequencies from the image.

Standard operators for FFT are:

- `fft_generic`: a generic version of the complex Fourier transform,
- `rft_generic`: real-valued Fourier transform,
- `fft_image`, a shortcut for the forward transformation with `fft_generic`
- and `fft_image_inv`, a shortcut to the backward transformation with `fft_generic`.

15.5.2 Texture Analysis in Color Images

By combining texture analysis with Color Processing on page 131, even more tasks can be solved efficiently. Typically, the color image is first split into color channels and possibly transferred into another color space. Then, texture filtering is applied and/or texture features are computed.



a) b)

Figure 15.8: a) Original image b) with FFT enhanced defect structure.

15.6 More Information About Texture Features

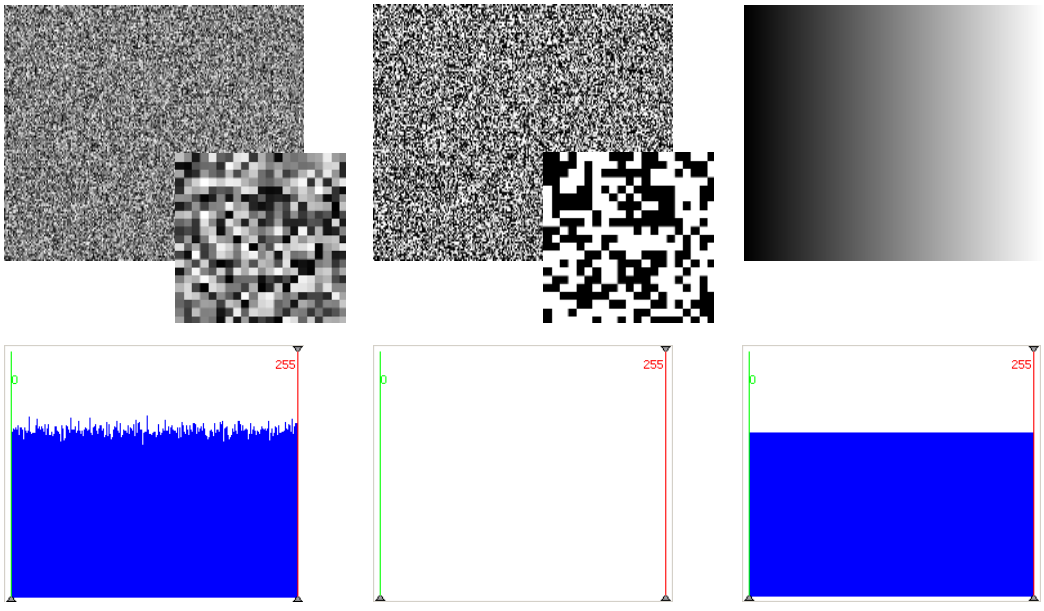
15.6.1 Entropy and Anisotropy (`entropy_gray`)

Entropy and anisotropy describe the gray value distribution of an image independent of the position of the pixels.

The entropy for white noise, as shown in image a) of figure 15.9, is high as white noise comprises of an equal number for all gray values.

Even though binary noise (image b) of figure 15.9 looks similar to white noise at first sight, it is in fact quite different because it only consists of two gray values: black and white. Therefore, the entropy is very low.

Image c) of figure 15.9 shows a gray ramp and image a) of figure 15.9 shows white noise. Both have the same value for entropy even though they look very different. Both images include, however, the same frequency for each gray value. This shows that the position of pixels within an image has no influence on the image's entropy.



a) entropy = 8 b) entropy = 1 c) entropy = 8

Figure 15.9: a) White noise; b) binary noise; c) gray ramp.

Even though real images do not usually show such extreme values, the examples in figure 15.10 still highlight the connection between gray value distribution (as is presented in the histogram) and the entropy value.

Anisotropy determines the symmetry of the gray value distribution. A value of -0,5 means that the histogram curve is symmetric. The value is higher if the image has more bright parts and lower if it has more dark parts as becomes clear from images a) and b) of figure 15.11.

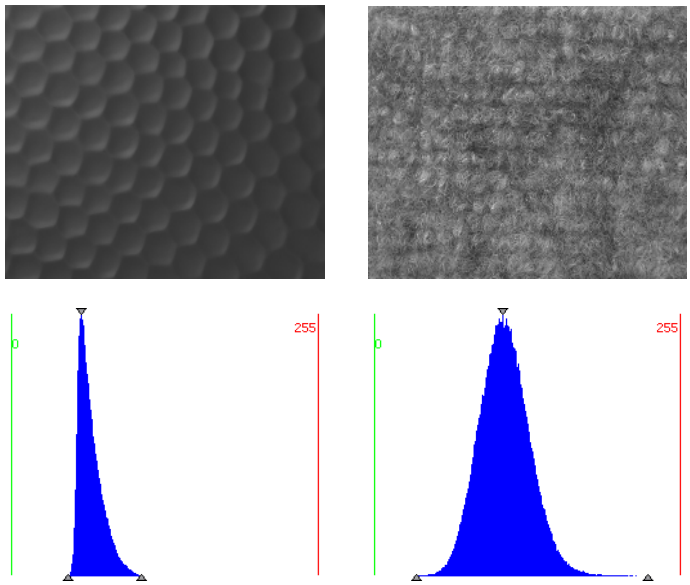


Figure 15.10: Entropy values of a) honeycomb structure and b) dust filter.

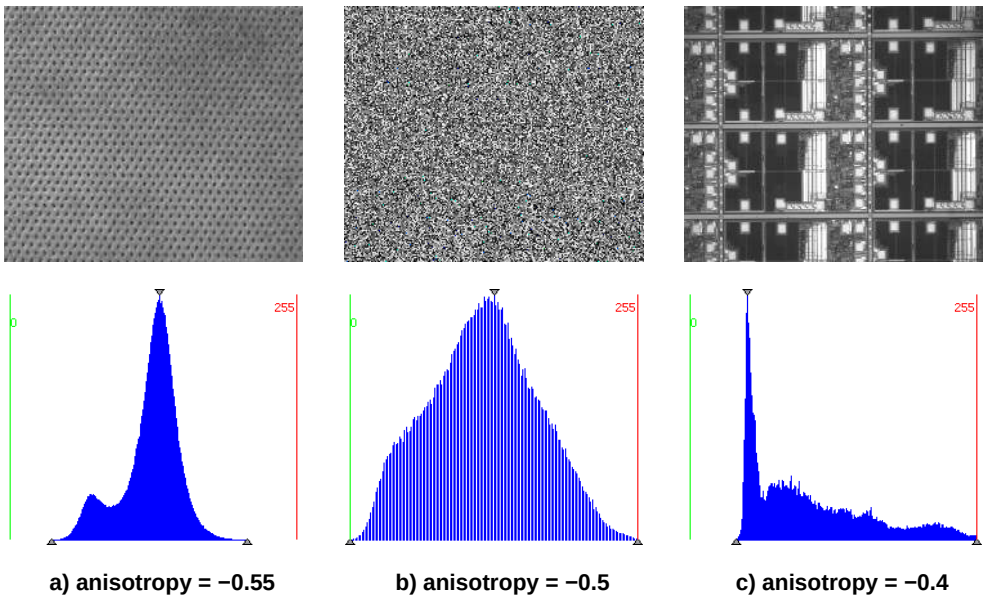


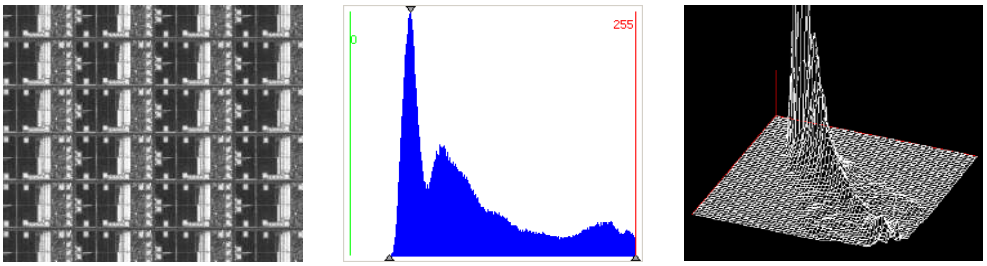
Figure 15.11: Anisotropy in images with different gray value distributions.

15.6.2 Cooccurrence Matrix (`gen_cooc_matrix`)

The cooccurrence matrix describes the relationship between the gray value of a pixel and the gray values of its neighbors. The matrix elements contain the probability that two gray values appear next to each other. Slow gray value changes are indicated by high values along the diagonal of the principal axis. Strong contrasts lead to entries far away from the diagonal of the principal axis. The connection between gray values in the image, in the histogram, and finally in the cooccurrence matrix is shown in images a), b), and c) of figure 15.12. Computing the cooccurrence matrix, you can select which neighbors are evaluated. Based on the cooccurrence matrix, four features can be computed:

- energy,
- correlation,

- homogeneity, and
- contrast.



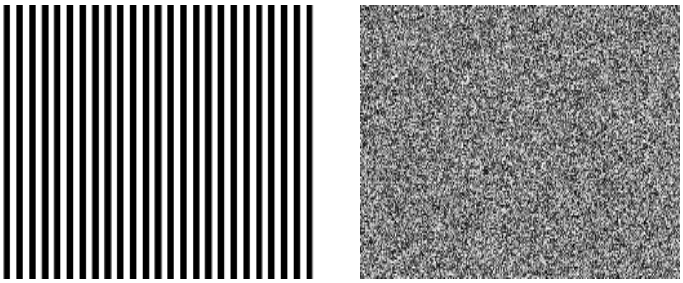
a) b) c)

Figure 15.12: a) Original image; b) gray value histogram; c) cooccurrence matrix.

15.6.3 Features of the Cooccurrence Matrix

Energy

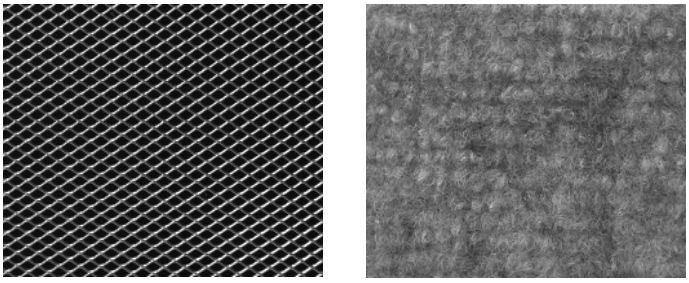
Energy is actually the short form for *uniformity of energy* and indicates if all kinds of gray value combinations are present or if certain gray value combinations are dominant. An image with a constant gray value has the highest energy.



a) energy = 0.45 b) energy = 0.0002

Figure 15.13: Energy values.

Image a) in figure 15.13 has a high energy because there are only few gray value combinations, which change always according to the same scheme. Image b) has little energy because the changes between pixels differ, i.e., there is no uniformity.



a) energy = 0.088 b) energy = 0.005

Figure 15.14: Energy values of a) canvas and b) dust filter.

The textures in figure 15.14 show how parts that contain pixels of the same gray value contribute to the higher energy of image a), whereas gray values in image b) are almost constantly changing and therefore lead to a low energy.

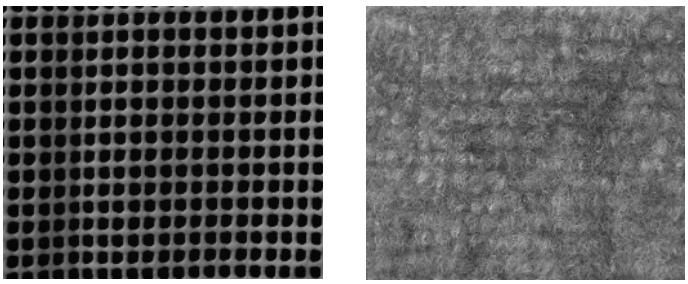
Correlation

Correlation defines the measure of dependency between neighboring values. If the matrix shows a thin line from top left to right bottom, this means that the image is blurry, which is why the values are so close together. A wide, roundish progress indicates a focused image. Contrasts and little or no correlation are indicated by pixels far away from the main diagonal. The question of correlation between pixels can however sometimes depend on the image's orientation as is depicted in figure 15.15.

a) original image b) correlation 0° = −1.0 c) correlation 90° = 1.0

Figure 15.15: Correlation values for different image orientations. Note that images b) and c) are depicted in pseudocolors to enhance the perceptibility of the cooccurrence matrix.

The two correlation values of the same image are very different depending on the rotation of the image. 0° causes a low correlation whereas 90° has a high one. In figure 15.16 the correlation values show that image a) with extreme gray value changes has a lower value than image b) with a texture that consists of gray values which are quite similar to each other.

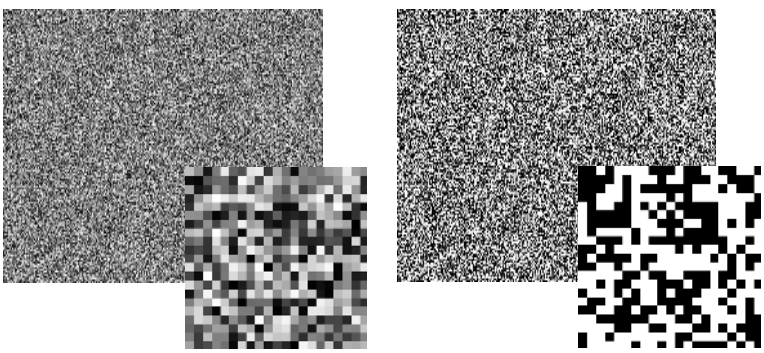


a) correlation = 0.0959 b) correlation = 0.795

Figure 15.16: Correlation values of a) mesh and b) dust filter.

Homogeneity

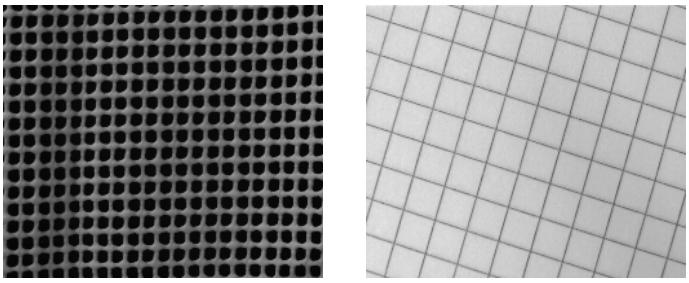
Homogeneity is high if large areas in an image have the same gray values.



a) homogeneity = 0.47 b) homogeneity = 0.5

Figure 15.17: Homogeneity values.

Image a) of figure 15.17 does not include any groups of pixels with the same gray values which makes it very inhomogeneous. Image b) with white noise includes areas, a feature which makes it more homogeneous. A unicolored image has the highest homogeneity value.



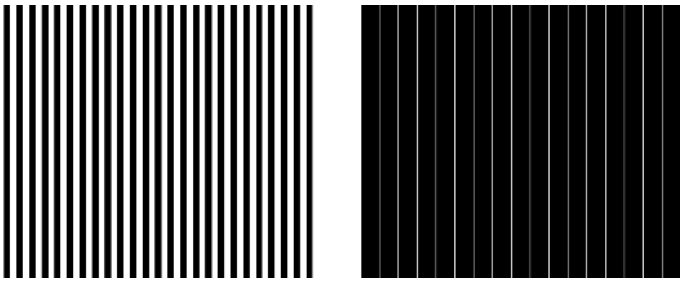
a) homogeneity = 0.65 **b) homogeneity = 0.66**

Figure 15.18: Homogeneity values of a) mesh and b) grid.

Realistic images, as shown in figure 15.18, clarify these homogeneity rules. Even though both images look partially homogeneous (they both contain bright and dark parts of a certain size but also include some changes from dark to bright) there are little differences. The mesh image is less homogeneous because the structure is three dimensional and therefore little deviations in brightness can cause slightly different gray values on the white mesh. Image b), however, contains bigger parts with a same gray value (the white background) and is also only two dimensional.

Contrast

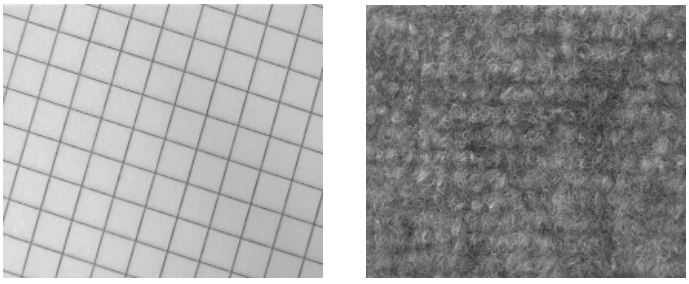
The contrast value between two pixels is multiplied and squared. High differences lead to steep edges and a high value.



a) contrast = 3969 **b) contrast = 203,9**

Figure 15.19: Contrast values.

The example images in figure 15.19 depict lines that run in different distances from each other. The image with the denser lines also has a higher contrast. This shows that the contrast value includes the frequency as well as the intensity.



a) contrast = 86.2 **b) contrast = 10.8**

Figure 15.20: Contrast values of a) grid and b) dust filter.

In figure 15.20, the grid structure in image a) has a higher contrast than the dust filter in image b) because the gray value changes in image a) are more extreme than the gray value changes in image b).

15.7 More Information About Texture Filtering

15.7.1 The Laws Filter (`texture_laws`)

By applying a Laws filter you can enhance or suppress textural structures in an image based on their spatial frequency, i.e., the size of the texels.

The operator `texture_laws` applies texture transformations (according to K. I. Laws) to an image. This is done by convolving the input image with a special filter mask of either 3x3, 5x5, or 7x7 pixels. For most of the filters the resulting gray values must be modified by a shift. This makes the different textures in the output image more comparable to each other, provided suitable filters are used.

The name of the filter is composed of the letters of the two vectors used, where the first letter specifies a convolution in the column direction, while the second letter specifies a convolution in the row direction. The letters themselves specify the convolution. You can select "l", "e", "s", "r", "w", "o" with "l" emphasizing low and "o" emphasizing high frequencies.

Therefore, the second letters detecting low frequencies to high frequencies are "l", "e", "s", "r", "w" and "o". The example figures figure 15.21 and figure 15.22 show how the two common texture filters "le" and "ls" work on two different textures.

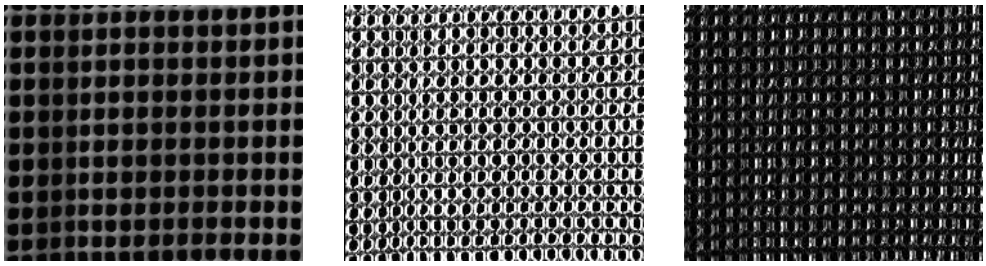


Figure 15.21: a) Mesh (original image) b) Laws filter le and b) Laws filter ls.

The Laws filter "le" looks for low frequencies in the image. As can be seen in figure 15.21, there are a few low frequencies in the image. The Laws filter "ls", however, looks for high frequencies, which occur very rarely in the image in places where gray values change from black to white and vice versa. As a result, the filtered image looks quite dark.

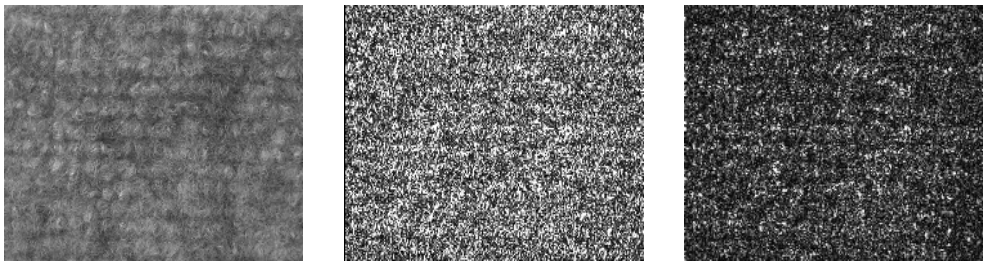


Figure 15.22: a) Dust filter (original image) b) Laws filter le and b) Laws filter ls.

The filtered images in figure 15.22 show that the original image contains low frequencies as well as high frequencies as it responds to both the "le" and "ls" filter. The Laws filter is therefore useful to distinguish textures which differ in their frequencies in a certain direction.

Chapter 16

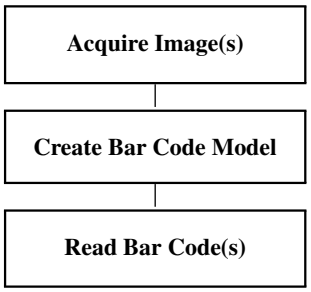
Bar Code

The idea of bar code reading is quite easy. You initialize a bar code model and then execute the operator for reading bar codes. Within this operator, you specify the desired bar code type. That way, you can read different bar code types by successively applying the operator with different parameters for the type, but without the need to create a separate model for each bar code type. The result of the reading is a region that contains the bar code and the decoded string.

The advantage of the HALCON bar code reader is its ease of use. No advanced experience in programming or image processing is required. Only a few operators in a clear and simple order are applied. Furthermore, the bar code reader is very powerful and flexible. An example for this is its ability to read an arbitrary number of bar codes of the same code type in any orientation even if parts are missing.

16.1 Basic Concept

Bar code reading consists mainly of these steps:



16.1.1 Acquire Image(s)

For the online part, i.e., during reading only, images must be acquired.

For detailed information see the description of this method on page 21.

16.1.2 Create Bar Code Model

You create a bar code model with the operator `create_bar_code_model`. The returned handle provides all necessary information about the structure of the bar code. In most cases, you do not have to adjust any parameters.

16.1.3 Read Bar Code(s)

Bar codes are read using the operator `find_bar_code`. Within this operator, you specify the model and the bar code type that you search for. The operator returns the regions and the decoded strings of all bar codes of the specified type that are found in the image or in the specified region of interest.

16.1.4 A First Example

As an example for this basic concept, here is a very simple program, that reads the EAN 13 bar code depicted in figure 16.1.

Figure 16.1: Reading a bar code.

A test image is acquired from file and the bar code model is created with `create_bar_code_model`. Then, the operator `find_bar_code` (with `CodeType` set to `'EAN-13'`) returns the region and the decoded string for the found bar code.

```
read_image (image, 'barcode/ean13/ean1301')
create_bar_code_model ([], [], BarCodeHandle)
find_bar_code (image, SymbolRegions, BarCodeHandle, 'EAN-13', \
               DecodedDataStrings)
```

16.2 Extended Concept

In some cases, bar code reading can be more advanced than in the example above. For example, a preprocessing or rectification of the image as well as the visualization of results can be necessary.

16.2.1 Use Region Of Interest

Bar code reading can be sped up by using a region of interest. The more the region in which the code is searched can be restricted, the faster and more robust the search will be.

For detailed information see the description of this method on page 25.

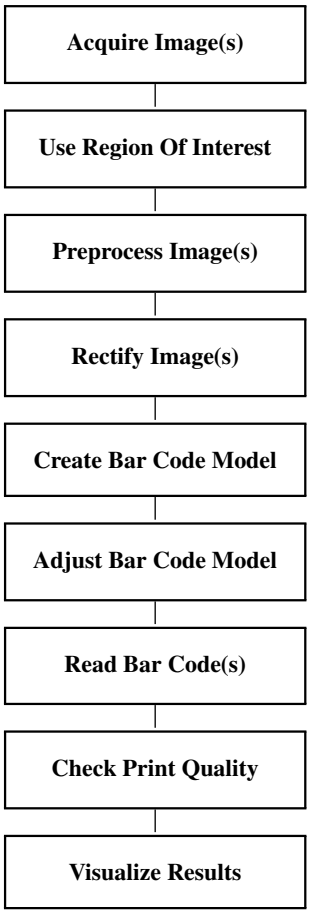
16.2.2 Preprocess Image(s)

There are various possibilities of preprocessing bar codes. Some of the most common ones are listed below.

If a bar code image is a bit blurry, `emphasize` can be used to emphasize high frequency areas of the image and, therefore, let it appear sharper.

The operator `zoom_image_factor` enlarges a bar code that is too small to be read.

In case of low resolution (< 2.0 pixels element size), operators like `emphasize`, `zoom_image_factor`, or `mean_image` should not be used for preprocessing, as crucial information could be lost after the image manipulation. See the section on `'small_elements_robustness'` in the operator reference of `set_bar_code_param` for details.



Low contrast images can be enhanced by scaling the gray value range with `scale_image` (or easier with the external convenience procedure `scale_image_range`).

HALCON expects bar codes to be printed dark on a light background. To read light bar codes on a dark background you must first invert the image using the operator `invert_image`.

16.2.3 Rectify Image(s)

The bar code reader can handle perspective distortions up to a certain extent. For very strong distortions, it might be necessary to rectify the image before applying the bar code reader.

How to perform the rectification for a bar code that is printed radially on a CD is shown in the description of the example program `circular_barcode.hdev` on page 171. Detailed information about compensating distortions caused, e.g., by non-perpendicularly mounted cameras, can be found in the Solution Guide III-C in section 3.4 on page 80.

16.2.4 Create Bar Code Model

Sometimes, a better result can be obtained when adjusting parameters like the range for the element size (`'element_size_min'`, `'element_size_max'`). All available parameters can be changed either here, or in a later step using `set_bar_code_param`.

16.2.5 Adjust Bar Code Model

In most cases no adaptation of parameters is necessary for reading bar codes with HALCON. If the bar code reading fails, however, you can adjust the following parameters with the operator `set_bar_code_param` before applying `find_bar_code`.

Adjust Bar Code and Element Size

In particular, you can adjust the bar code size and the range of the bar code element's width with the parameters

- 'barcode_width_min',
- 'barcode_width_max',
- 'barcode_height_min',
- 'element_size_min', and
- 'element_size_max'.

The description of the example barcode_typical_cases.hdev on page 168 shows how an adaption of the element's size can improve the decoding.

Determine Check Character

For code types having a check character, you can specify whether the check character is used to check the result or if it is returned as part of the decoded string ('check_char').

Adapting the Thresholds for the Extraction of Element Edges

Two threshold parameters control the detection of edges within a bounding box. 'meas_thresh' calculates a relative threshold by detecting the highest and lowest gray values within a rough bar code region and using this dynamic range as a basis for edge detection. 'meas_thresh' works well unless images contain lots of noise. It also fails if the code is a stacked code, which, due to its structure, contains a white patch within the bounding box. In both cases the calculated gray-value range of the images is small. This results in an unreasonably small value for 'meas_thresh' and can therefore cause the detection of false edges.

This, however, is prevented by 'meas_thresh_abs', which sets an absolute threshold of 5.0 for all images. This means that the difference between the brightest and the darkest value of the element border has to be more than 5.0 to be classified as an edge. Therefore, no false edges are detected in images with a small dynamic range.

Sometimes, a low threshold is needed, e.g., if an image contains parts with a high dynamic range as well as parts with a low one, due to inhomogeneous illumination. Then, 'meas_thresh_abs' must be turned off, by setting it to 0.0 before decreasing the value of 'meas_thresh'.

Speedup

There are two cases in which you can improve the performance. The first case concerns images with too many false candidates, which are scanned with the default value of 10 scanlines each, whereas a valid bar code itself is usually decoded after one or two scans. Reducing the number of scanlines therefore leads to a significant speedup. Note that if the bar code cannot be detected after reducing the number of scanlines, the number has to be increased again. The second case concerns images with stacked bar codes (currently RSS-14 Stacked, RSS-14 Stacked Omnidirectional, and RSS Expanded Stacked). The reader evaluates by default the necessary number of scanlines, 20 for RSS-14 Stacked and 55 for RSS Expanded Stacked, which may be higher than necessary. If an RSS Expanded Stacked code does not have the maximum possible number of rows, it is useful to specify the number of scanlines with the parameter 'num_scanlines' (the reader uses 5 scanlines per row). Consequently, the performance is increased. Both cases are demonstrated in the example %HALCONEXAMPLES%\hdevelop\Identification\Bar-Code\barcode_param_num_scanlines.hdev. Please refer to the Reference Manual for further information. If the number of bar codes per image is known, decoding time can be decreased by specifying this number with the parameter 'stop_after_result_num' of the operators set_bar_code_param or set_bar_code_param_specific. This way, the detection of false codes after all existing bar codes have already been read can be prevented and, as a consequence, processing can be sped up.

Increase Bar Code Detection Robustness

The parameter 'min_identical_scanlines' specifies how many successfully decoded scanlines with identical result are required to decode a bar code. If this parameter is not set, a bar code is considered as decoded after the first scanline was successfully decoded. It can be useful to increase the value of this parameter when working either with an image containing lots of noise or with various bar codes to prevent the detection of false bar codes. The use of the parameter is demonstrated in the example %HALCONEXAMPLES%/hdevelop/Identification/Bar-Code/barcode_param_scanning_control.hdev.

In some cases it might be useful to decode a bar code based on the majority of all scanlines instead of the numbers of identical scanlines, e.g., if false decode results shall be reduced. The parameter 'majority_voting' specifies the selection mode for the decode result. If this parameter is

not set, a bar code is decoded successfully if the minimal number of scanlines was decoded as described above. If this parameter is set to 'true', a majority voting scheme is used to select between different scanline results and the overall result is decoded by the majority of all scanlines. The use of the parameter is demonstrated in the example %HALCONEXAMPLES%/hdevelop/Identification/Bar-Code/barcode_param_majority_voting.hdev. Please refer to the Reference Manual for further information.

Enforce Verification of the Quiet Zone

HALCON is able to read bar codes even if their quiet zones are violated. However, the parameter 'quiet_zone' can be used to handle quiet zone violations more rigidly. The use of this parameter is demonstrated in the example %HALCONEXAMPLES%/hdevelop/Identification/Bar-Code/barcode_quiet_zone.hdev.

Adapt Tolerance for Start/Stop Matching Criteria

The 'start_stop_tolerance' specifies the tolerance for detecting the start- and stop pattern. A less tolerant criterion increases the robustness against false detections, but might reduce the general detection rate. This criterion can be activated by setting the parameter to 'low'. A 'high' tolerance is recommended if the boundary of the symbol in the image is clearly visible, the image is not noisy, and the symbol does not contain bar codes of different types. Note that the parameter is currently only implemented for Code 128. How to adapt the parameter 'start_stop_tolerance' to avoid misdetection is demonstrated in the example %HALCONEXAMPLES%/hdevelop/Identification/Bar-Code/barcode_param_scanning_control.hdev.

To query the values that are currently used for the individual parameters, you can apply the operator get_bar_code_param.

Determine Bar Code Parameters by Training In order to determine the best parameter settings, an automatical training can be performed. The advantage of this training is an increased robustness of the bar code reader. Furthermore, the decoding time can be reduced as a side effect. Parameter settings can be determined for 'element_size_min', 'element_size_max', 'orientation', 'meas_thresh', and 'meas_thresh_abs'. To activate the training mode, use the operator create_bar_code_model to create a model for the training and set the generic parameter 'train' to a tuple of parameter names that should be trained or 'all' to train all bar code parameters that are available for the training. To determine all parameters with given properties, e.g., 'trained', the operator query_bar_code_params can be used. How to perform the training is demonstrated in the example %HALCONEXAMPLES%\hdevelop\Identification\Bar-Code\barcode_training.hdev. Please refer to the Reference Manual for further information.

16.2.6 Read Bar Code(s)

The operator find_bar_code allows you to either find one bar code type or several bar code types in one call. When reading one code type, choose the type - if it is known - as 'CodeType'.

Autodiscrimination

Autodiscrimination describes the simultaneous decoding of multiple bar code types. It is activated by setting 'CodeType' to 'auto' or by choosing the expected bar code types. Note, however, that each allowed bar code type increases the runtime of the operator and too many bar code types may decrease the reliability of the decoding. You should at least exclude all definitely not occurring bar code types that are scanned before the first expected bar code type occurs or, even better, just scan for the expected bar code types (see figure 16.2). For more information on autodiscrimination including the decoding order, please refer to the documentation of find_bar_code in the Reference Manual. Furthermore, the example %HALCONEXAMPLES%\hdevelop\Identification\Bar-Code\barcode_autodiscrimination.hdev demonstrates the practical use of autodiscrimination. The operators set_bar_code_param_specific and get_bar_code_param_specific allow to set parameters of the bar code model specifically for certain bar code types, which may especially be interesting for autodiscrimination.

Restrict the Reading to a Part of the Image

If the code always appears approximately in the same position within every image, you can speed up your application using the operator decode_bar_code_rectangle2 instead of the operator find_bar_code. Rather than performing a time-consuming search for candidate regions, decode_bar_code_rectangle2 scans the provided region directly for bar codes. The position of the code is given in an arbitrarily oriented rectangle. Note that the direction is important. It needs to be perpendicular to the elements.

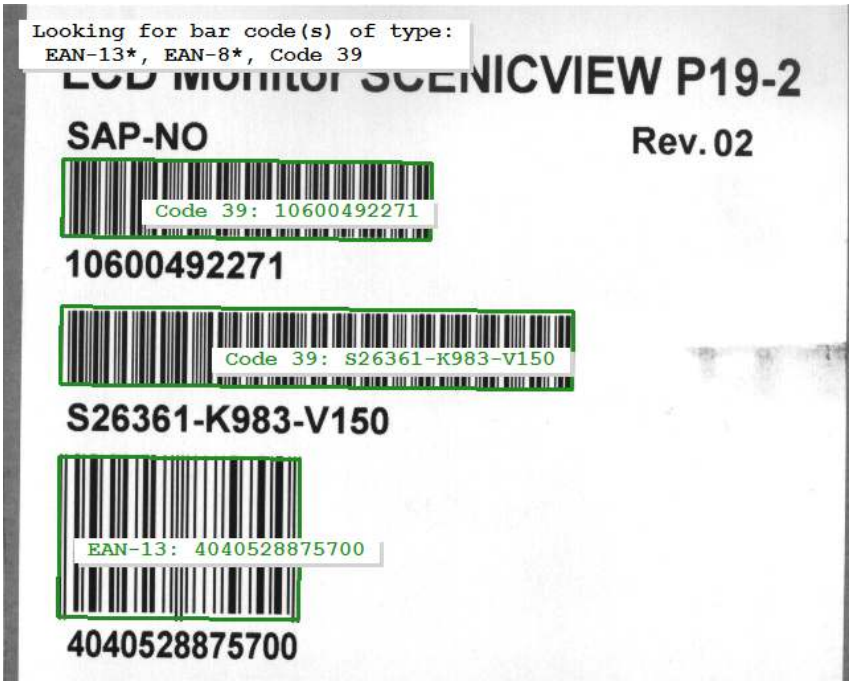


Figure 16.2: Using the 'autodiscrimination' feature, different bar codes within one image - in this case two codes of the type Code 39 and one EAN-13 - can automatically be recognized and read.

Small Elements Robustness

If `find_bar_code` or `decode_bar_code_rectangle2` are not able to decode a sufficient number of scanlines, the operators apply an additional decoding attempt (as long as `'small_elements_robustness'` was not set to `'false'` and `'element_size_min'` is set smaller than `2.0`). This approach can help to read the code in case the resolution of the elements is rather low.

For more information on this decoding step see the section on `'small_elements_robustness'` in the operator reference of `set_bar_code_param`).

Access Results

After reading the bar codes with `find_bar_code`, you can explicitly query the regions of the decoded bar codes or their candidates with the operator `get_bar_code_object`. The decoded strings or the underlying reference data (including, e.g., the start/stop characters) can be queried with the operator `get_bar_code_result`.

Diagnostic Status Messages

In order to control why scanlines are not successfully detected or why the code reading fails completely, diagnostic status messages in human readable format can be returned (see figure 16.3). With the operator `get_bar_code_result` set to `'status'` or `'status_id'` and supplying a candidate handle, diagnostic status messages for each possible scanline in the candidate region are returned in a tuple. Use the operator `get_bar_code_object` with the parameter `'scanlines_all'` to query the corresponding scanlines or `'scanlines_valid'` to query all valid scanlines. If you want to further process the scanline information, the `'_plain'`-variants of the parameters mentioned above, i.e., `'scanlines_all_plain'` and `'scanlines_valid_plain'` can be used. They return the detected or valid scanlines as XLD contours. Note that the status of the scanlines of a bar code can only be evaluated if the operator `find_bar_code` or the operator `decode_bar_code_rectangle2` was called before while in `'persistence'` mode. The `'persistence'` mode allows the storage of intermediate results during bar code decoding.

Also note that status information about the scanlines after the additional decoding step regarding small elements robustness (described above) needs to be retrieved using `'status_small_elements_robustness'`, while `'status'` and `'status_id'` still return messages concerning the decoding status without the additional approach.

Determining the status increases runtime and should therefore only be performed if the debugging information is needed. The example `%HALCONEXAMPLES%\hdevelop\Identification\Bar-Code\barcode_status.hdev` demonstrates the use of the bar code parameters `'status'` and `'status-id'`.

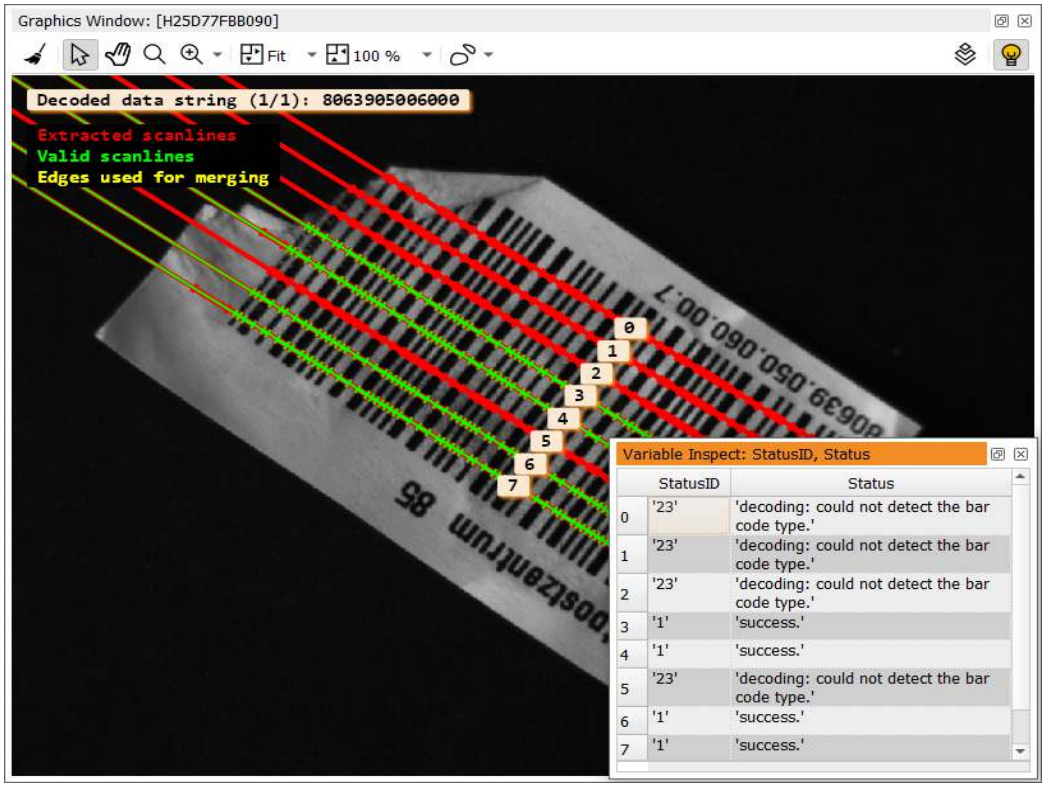


Figure 16.3: Status messages can be used to solve problems that might occur when reading bar codes.

What if the Code Reading Fails?

If the code reading fails, an error message is returned. Depending on the message we have several suggestions on how to solve the problem.

If regions are not found properly, you can either improve the image quality itself, which helps to solve many problems, or - if that is not possible - adjust the software to better detect your code. If you know the location of your bar code, you can use the operator `decode_bar_code_rectangle2` to specify the location instead of searching the whole image (see above).

Another reason why regions are not detected can be an element size that is not adjusted to the actual size of the bar code - which is especially a problem, if the code is very big or very small. You can solve the problem by adjusting the parameters `element_size_min` or `element_size_max`. How to adapt these parameters and handle other typical cases is described in the example `%HALCONEXAMPLES%\solution_guide\basics\barcode_typical_cases.hdev`, which is documented in this chapter (see the example descriptions in section 16.3 on page 168). This example presents some typical cases of problems that can occur while reading bar codes and presents some easy solutions.

The error messages (see the table of error messages on page 166) which lead to a failure of code reading begin either with 'edges', 'decoding' or with 'check'. These categories represent the state within the decoding process in which the code reading failed. First, edges of the bars, i.e. the white-to-black and black-to-white transitions from background to bar and bar to background, are detected. Then the code is decoded according to the specified symbology and finally several tests ('check') are performed to ensure a correct result.

The errors with the codes (0),(8),(9) and (13) ('unknown status', 'decoding: internal error when estimating the maximum string length', 'decoding: internal error when decoding a single character' and 'decoding: error decoding the reference to a human readable string') are internal errors that should not occur. If they do, however, occur, please contact your local HALCON distributor.

Warning messages (see page 167) occur either with a success message or with an error message. Warnings do not prevent a successful code reading but are merely an indication for possible (future) problems.

Error Message(s)	Possible Solution
edges: not enough edges detected.	Inspect the region by querying region and scanlines with the operators get_bar_code_object and get_bar_code_result.
edges: not enough edges for a start, a stop and at least one data character.	
edges: too many edges detected.	
edges: center of scanline not within domain.	Change the ROI or use the operator full_domain to search the whole image. If the position of the code is known, the operator decode_bar_code_rectangle2 can be used to specify the exact position.
decoding: could not find stop character.	Check the found bar code region or increase the number of scanlines with the parameter num_scanlines of the operator set_bar_code_param(_specific).
decoding: could not find start and stop characters.	
decoding: number of wide bars of a single character is not equal to 2.	Check if the right code type was specified as 'CodeType' for the operator find_bar_code.
decoding: invalid encoding pattern.	
decoding: invalid mix of character sets.	
decoding: error decoding the reference to a human readable string.	
decoding: could not detect center guard pattern.	
decoding: could not detect left and/or right guard patterns.	
decoding: could not detect add-on guard pattern.	
decoding: could not detect enough finder patterns.	
decoding: no segment found.	
check: detected EAN-13 bar code type instead of specified type.	
check: checksum test failed.	Check if the code even has a checksum. If it does not, set the parameter 'check_char' of the operator set_bar_code_param_specific to 'absent'.
check: check of add-on symbol failed.	Check if the code has an add-on symbol. If it does not, change the 'CodeType' of find_bar_code to the non-add-on variant.
check: symbol region overlaps with another symbol region.	The whole region is skipped because another symbol region that overlaps with this region has already been decoded. This is no error. Set 'stop_after_result_num' of the operator set_bar_code_param or set_bar_code_param_specific to the number of expected codes.

16.2.7 Check Print Quality

If you are interested in checking the print quality of your bar code symbols, you can use the HALCON operator get_bar_code_result, which grades the symbol according to the standard ISO/IEC 15416:2016. For simple 1D bar codes the print quality is described in a tuple with nine elements:

- Overall Quality,
- Decode,
- Symbol Contrast,
- Minimal Reflectance,
- Minimal Edge Contrast,

Warning Message(s)	Possible Solution
White spaces too wide.	To improve image quality and prevent errors, please refer to the example program %HALCONEXAMPLES%\solution_guide\basics\barcode_typical_cases.hdev.
White spaces too narrow.	
Bars too wide.	
Bars too narrow.	
Possible saturation of gray values.	
No composite component found.	Check if the composite code is completely within the image.

- Modulation,
- Defects,
- Decodability,
- and Additional Requirements.

Note that the names and the order of the grades can be queried with the convenience option (query_isoiec15416_labels).

For composite bar codes, the print quality is described in a tuple with 24 elements, including components of both bar code print quality grading and data code print quality grading as described for the method Data Code on page 175 or in more detail in the Solution Guide II-C in section 6 on page 45.

When investigating the reason for quality defects, it can be useful to query the data that was used to calculate the values for the print quality elements. This is possible with the parameter `quality_isoiec_15416_values` which returns a tuple with the raw values for the following grades: Symbol Contrast, Minimal Reflectance, Minimal Edge Contrast, Modulation, Defects, and Decodability for simple bar codes, and Symbol Contrast, Minimal Reflectance, Minimal Edge Contrast, Modulation, Defects, Decodability, Rap Contrast, Rap Minimal Reflectance, Rap Minimal Edge Contrast, Rap Modulation, Rap Defects, Rap Decodability, Codeword Yield, and Unused Error Correction for composite bar codes. For grades that are excluded from these lists the operator reports 'N/A'.

For further information on checking print quality of composite bar codes or regular bar codes, please refer to the description of the operator `get_bar_code_result` in the reference manual.

16.2.8 Visualize Results

Finally, you might want to display the images, the bar code regions, and the decoded content.

For detailed information see the description of this method on page 223.

16.3 Programming Examples

This section shows how to use the bar code reader.

16.3.1 How to Read Difficult Barcodes

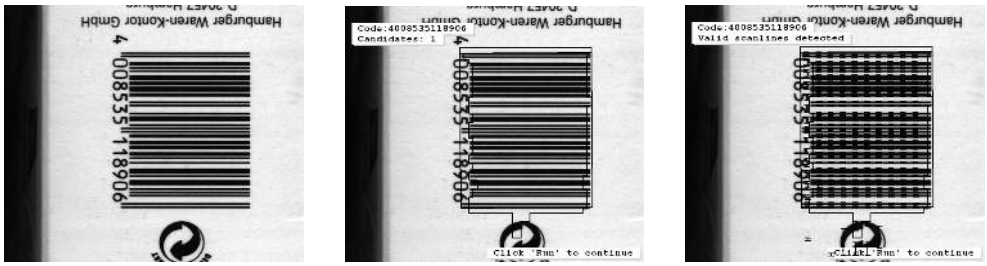
Example: `%HALCONEXAMPLES%/solution_guide/basics/barcode_typical_cases.hdev`

When reading bar codes, there are a few conditions you have to consider. These conditions might have to do with the code itself, for which you may, for example, have to use different parameters because it is very big or small, or with the image acquisition environment affecting the image quality. Even though you should always aim for achieving very good images, for technical reasons, it might sometimes not be possible to reach a perfect image quality.

The following example can help you identify simple obstacles, find out at which stage of the bar code reading process they occur, and it subsequently offers suggestions for solving the problem.

The different cases within the example are based on typical defects. There are no problems with reading the bar code for case 0. It just presents the two stages at which intermediate results can be obtained (see figure 16.4).

You can query candidate regions to find out if the reader detects a bar code candidate at all with the local procedure `disp_bar_code_candidates`. To query scanlines, it is necessary that a candidate region is found. You can observe the scanlines with the local procedure `disp_bar_code_scanlines`. Both procedures use the operator `get_bar_code_object`, setting either the parameter `'candidate_regions'`, `'scanlines_all'` or `'scanlines_valid'`, respectively.



a) b) c)

Figure 16.4: a) Original bar code image b) A candidate is detected. c) Valid scanlines are detected.

Case 1 deals with a bar code in a very low contrast image (see figure 16.5). Subsequently, no candidates are found. However, just one preprocessing step is necessary to overcome this obstacle. The knowledge that there is a defined gray value difference between the code bars and the background leads to the solution. The single gray values lie, however, very close together which makes it impossible for the reader to distinguish the code from the background. The contrast can simply be improved by scaling the gray value range of the image with the convenience procedure `scale_image_range` and therefore achieving a bright background and a black bar code again. To obtain the scaling range, it is useful to check the gray histogram of the image.

```
scale_image_range (Image, ScaledImage, 0, 20)
```

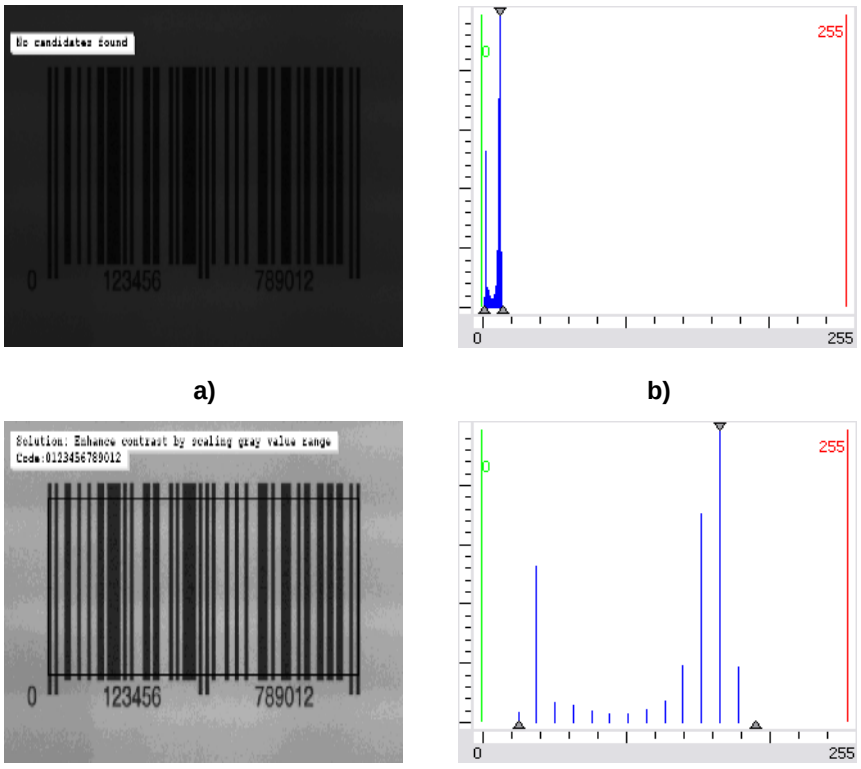


Figure 16.5: a) A bar code with a very low contrast. b) Histogram of the low contrast image. c) Scaling gray values make it possible to decode the bar code. d) The histogram of this image shows how the gray values have been scaled.

After scaling the image, the code can be read and both candidate and scanlines are found.

Case 2 shows another problem that is connected with illumination. The light distribution in the image is inhomogeneous (see figure 16.6). `'disp_bar_code_candidates'` does not detect the "whole" candidate first, because the contrast diminishes in the darker parts of the image. Decreasing the value of `'meas_thresh'` makes it possible to detect and read the whole bar code. It is, however, necessary to disable `'meas_thresh_abs'` first, because `'meas_thresh_abs'` sets a default threshold of 5.0, which is too high in this case.

```
set_bar_code_param (BarCodeHandle, 'meas_thresh_abs', 0.0)
set_bar_code_param (BarCodeHandle, 'meas_thresh', 0.02)
```

a) b)

Figure 16.6: a) Bar code with inhomogeneous illumination b) Decreasing `'meas_thresh'` makes the code readable.

16.3.2 Reading a Bar Code on a CD

Example: %HALCONEXAMPLES%/hdevelop/Applications/Bar-Codes/circular_barcode.hdev

Figure 16.7 shows an image of a CD, on which a bar code is printed radially. The task is to read this circular bar code. Because the bar code reader cannot read this kind of print directly, first the image must be transformed such that the elements of the bar code are parallel.

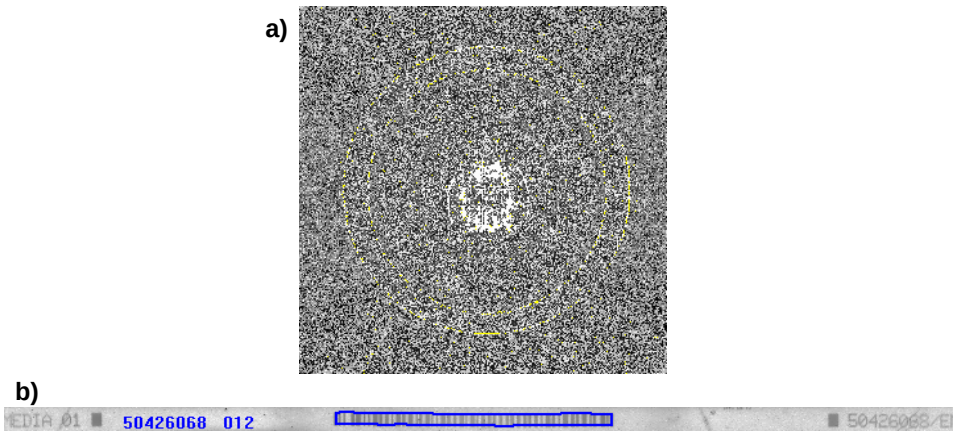


Figure 16.7: (a) Original image with segmented ring; (b) rectified ring with decoded bar code.

The first step is to segment the dark ring on which the bar code is printed. This is performed using threshold followed by closing_circle and connection. This segmentation returns all dark areas in the image, including the dark ring. To select the ring, select_shape is used with corresponding values for the extent.

```
threshold (Image, Region, 0, 100)
closing_circle (Region, Region, 3.5)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, Ring, ['width', 'height'], 'and', [550, \
              550], [750, 750])
```

After that, the parameters of the outer and the inner circle are determined. The outer circle can be determined directly using shape_trans. The inner circle is more complicated to extract. Here, it is calculated by creating the complement region of the ring with appropriate dimensions and then selecting its inner part. With smallest_circle, the parameters of the inner and outer circle are determined.

```
shape_trans (Ring, OuterCircle, 'outer_circle')
complement (Ring, RegionComplement)
connection (RegionComplement, ConnectedRegions)
select_shape (ConnectedRegions, InnerCircle, ['width', 'height'], 'and', \
              [450, 450], [650, 650])
smallest_circle (Ring, Row, Column, OuterRadius)
smallest_circle (InnerCircle, InnerRow, InnerColumn, InnerRadius)
```

The parameters for inner and outer circle are the input for the polar transform (polar_trans_image_ext), which transforms the image inside the ring into a rectangular area. Then, the image is inverted to obtain dark bar code elements on a light background (see figure 16.7b).

```
WidthPolar := 1440
HeightPolar := round(OuterRadius - InnerRadius - 10)
polar_trans_image_ext (Image, PolarTransImage, Row, Column, rad(360), 0, \
                       OuterRadius - 5, InnerRadius + 5, WidthPolar, \
                       HeightPolar, 'bilinear')
invert_image (PolarTransImage, ImageInvert)
```

To read the bar code, first a model is created with create_bar_code_model. As the bar code elements are very thin, parameters are adjusted with the operator set_bar_code_param. In particular, the parameter 'ele-

ment_size_min' is set to 1.5 and the parameter 'meas_thresh' is set to 0.1. Then, find_bar_code finds and decodes the bar code, which is of the specified code type 'Code 128'.

```
create_bar_code_model ([], [], BarCodeHandle)
set_bar_code_param (BarCodeHandle, 'element_size_min', 1.5)
set_bar_code_param (BarCodeHandle, 'meas_thresh', 0.3)
find_bar_code (ImageZoomed, SymbolRegions, BarCodeHandle, 'Code 128', \
               DecodedDataStrings)
```

Finally, the region of the bar code that was returned by find_bar_code is transformed back to the original shape of the bar code by polar_trans_region_inv and is displayed in the image.

```
polar_trans_region_inv (SymbolRegions, CodeRegionCircular, Row, Column, \
                        rad(360), 0, OuterRadius - 5, InnerRadius + 5, \
                        WidthPolar, HeightPolar, Width, Height, \
                        'nearest_neighbor')
dev_display (CodeRegionCircular)
```

16.3.3 Checking Bar Code Print Quality

Example: %HALCONEXAMPLES%/hdevelop/Identification/Bar-Code/print_quality_isoiec15416.hdev

This example checks the print quality of bar codes, according to the ISO/IEC 15416:2016 standard, for contrast, minimal reflectance, modulation, minimal edge contrast, defects, decodability and additional requirements.

This program will be explained exemplarily, using the "Contrast" grade, which checks whether the range between the minimal and the maximal value in the reflectance profile is lower or equal to 0.5 of the maximal reflectance value. Otherwise, a value of 0 is assigned as you can see in figure 16.8. Furthermore, the program returns raw values for some quality grades to investigate the reason for quality defects.

First, the data code is detected in the image, then the operator get_bar_code_result checks the print quality of the code.

```
find_bar_code (ImageDefect, SymbolRegions, BarCodeHandle, 'EAN-13', \
               DecodedDataStrings)
get_bar_code_result (BarCodeHandle, 0, 'quality_isoiec15416_float_grades', \
                     Quality)
```

Finally, the result is displayed in a message.

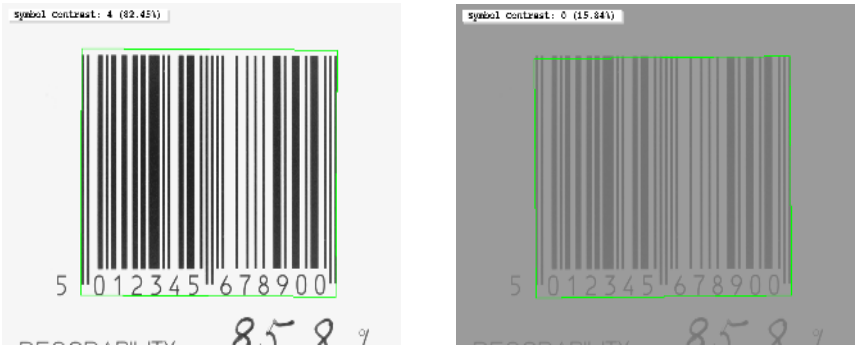
```
min_max_gray (SymbolRegions, ImageDefect, 0, Min, Max, Range)
Contrast := Range / 255
dev_display (ImageDefect)
dev_set_color ('green')
dev_display (SymbolRegions)
grade_message_text (Labels, Quality, QualityValues, GRADE_CONTRAST, Message)
disp_message (WindowHandle, Message, 'window', 10, 12, 'black', 'true')
```

The program repeats these checks for the print quality features listed above. For more information concerning composite codes view the example %HALCONEXAMPLES%/hdevelop/Identification/Bar-Code/composite_print_quality_isoiec15416.hdev.

16.4 Relation to Other Methods

16.4.1 Alternatives to Bar Code

OCR (see description on page 183)
With some bar codes, e.g., the EAN 13, the content of the bar code is printed in plain text below the elements. Here, OCR can be used, e.g., to check the consistency of the reading process.



a) b)

Figure 16.8: a) The bar code with a high contrast gets the grade 4 and a raw value of 82.45%. b) The defective bar code with a very low contrast gets the grade 0 and a raw value of 15.84%.

16.5 Advanced Topics

16.5.1 Use Timeout

With the operator `set_bar_code_param`, you can set a timeout. Then, the operator `find_bar_code` will return at the latest after the specified time. This mechanism is demonstrated for Matching (Shape-Based) in the example `%HALCONEXAMPLES%/hdevelop/Matching/Shape-Based/set_shape_model_timeout.hdev`.

Chapter 17

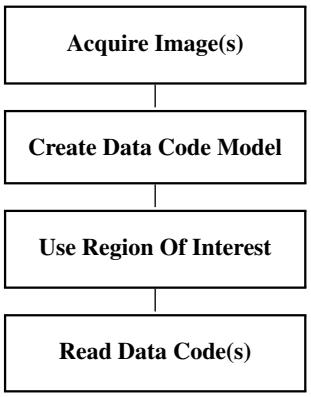
Data Code

Data codes are a special kind of two-dimensional patterns that encode text and numbers. HALCON is able to read the most popular data codes: Data Matrix ECC 200, QR Code, Micro QR Code, Aztec Code, PDF417, and DotCode. Depending on the code type, the symbols consist of multiple dots or small squares, which encode information according to the individual code specifications. Because of the special design of the codes, they can be decoded even if some parts are disturbed.

The advantage of the HALCON data code reader is its ease of use. No advanced experience in programming or image processing is required. Only a few operators in a clear and simple order need to be applied. Furthermore, the data code reader is very powerful and flexible. Examples for this are its ability to read codes in many print styles and the possibility to automatically learn optimal parameters.

17.1 Basic Concept

Data code reading consists mainly of these steps:



17.1.1 Acquire Image(s)

For the online part, i.e., during reading, images are acquired.

For detailed information see the description of this method on page 21.

17.1.2 Create Data Code Model

First, you create a data code model with the operator `create_data_code_2d_model`. This model provides the reader with all necessary information about the structure of the code. For normal printed codes only the name needs to be provided and HALCON will select suitable default parameters. For special cases, you can modify the model by passing specific parameters.

17.1.3 Read Data Code(s)

To read a data code, just one operator is needed: `find_data_code_2d`. It will locate one or more data codes and decode the content.

17.1.4 A First Example

As an example for this basic concept, here a very simple program, which reads the data code on the chip depicted in figure 17.1, is discussed.

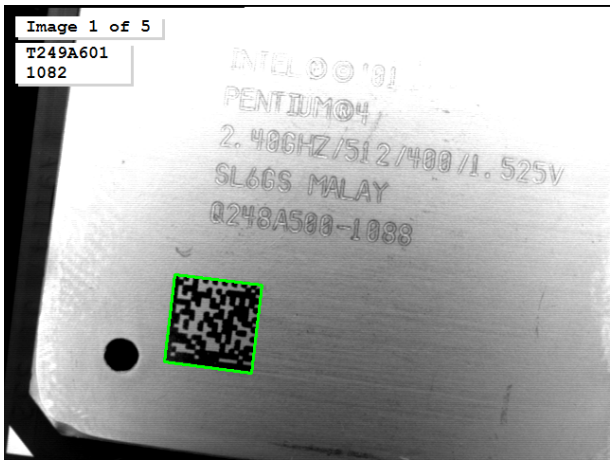


Figure 17.1: Reading a data code.

After reading an image from file, the data code model is generated by calling `create_data_code_2d_model`. As the only required parameter value, the code name `'Data Matrix ECC 200'` is specified.

```
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
```

Then, the data code is read with the operator `find_data_code_2d`.

```
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)
```

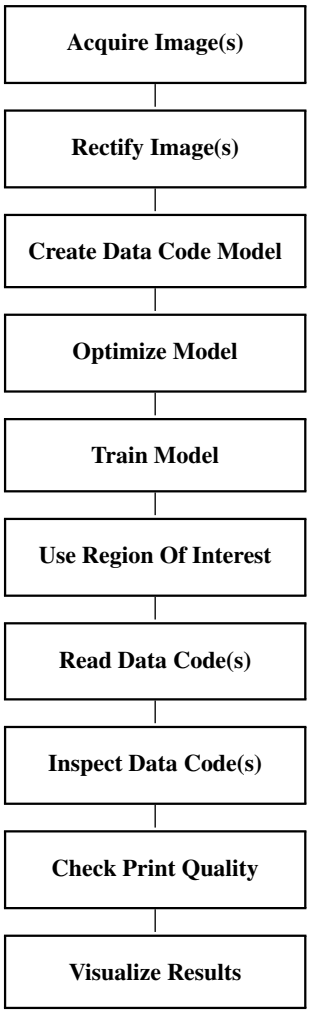
17.2 Extended Concept

In some cases, data code reading can be more advanced than in the example above. Reasons for this are, e.g., parameter optimization for improved execution time. Furthermore, preprocessing like rectification or the visualization of results might be required. The following sections give a brief overview. More detailed information can be found in the Solution Guide II-C.

17.2.1 Acquire Image(s)

Optionally, additional images can be acquired for parameter optimization (see the description of the step Optimize Model on page 178).

For detailed information see the description of this method on page 21.



17.2.2 Rectify Image(s)

HALCON's data code reader is robust against image distortions up to a certain limit. But if the data code is printed on a cylindrical surface or if the camera is tilted relative to the surface, it might be necessary to rectify the image before applying the data code reader.

Detailed information about rectifying images can be found in the Solution Guide III-C in section 3.4 on page 80.

17.2.3 Create Data Code Model

The operator `create_data_code_2d_model` expects the name of the desired code and optionally parameters to specify the geometry and radiometry as input. By default, a parameter set is used that is suitable for data codes that fulfill the following requirements:

- The code must be printed dark on light,
- the contrast must be bigger than 30,
- the sizes of symbol and modules are in a certain range (which depends on the selected symbol type),
- there is no or only a small gap in between neighboring modules of matrix codes (for PDF417 no gap is allowed),
- for QR Codes, additionally all three position detection patterns must be visible,
- for Micro QR Codes, the single position detection pattern must be visible, and
- for Aztec Codes, the position detection pattern must be visible as well.

This parameter set is also used if you specify the value 'standard_recognition' for the parameter GenParamValues. In contrast, if you specify the value 'enhanced_recognition', a parameter set is used that detects codes that do not follow the rules given above. If you choose the value 'maximum_recognition' even more data codes will be detected. However, using the parameter sets of enhanced mode and maximum mode possibly results in a longer processing time.

17.2.4 Optimize Model

Using the default parameters, the data code reader is able to read a wide range of codes. For non-standard codes the parameters can be adapted accordingly. For this, the operator set_data_code_2d_param is used.

The easiest way is to use the parameter value 'enhanced_recognition', which uses a model that is able to find a very wide range of print styles. An alternative is to specify parameter values separately to adapt the model to the conditions of the used print style.

If a data code symbol is not detected although it is well visible in the image, check whether the symbol's appearance complies with the model. In particular, have a look at the polarity ('polarity': dark-on-light or light-on-dark), the module size ('module_size' and 'module_shape') and the minimum contrast ('contrast_min'). In addition, the parameters 'module_gap' (allowed gap between modules), 'symbol_size', and 'slant_max' (angle variance of the legs of the finder pattern) should be checked. The current settings of all applicable parameters can be queried by the operator get_data_code_2d_param.

All possible parameter values can be checked in the Reference Manual. Besides this, they can also be queried with the operator query_data_code_2d_params.

As an alternative, you can train the model (see below).

17.2.5 Train Model

Instead of modifying the model parameters manually as described above, you can also let HALCON train the model automatically using the operator find_data_code_2d. All you need to do is to call this operator with the parameter values 'train' and 'all'. Then, HALCON will search for the best parameters needed to extract the given code. It is recommended to apply this to multiple example images to ensure that all variations are covered.

As an alternative, you can execute the finder with normal parameters and request the features of the found symbols with get_data_code_2d_results. These values can then be used to change the model with set_data_code_2d_param.

17.2.6 Use Region Of Interest

Reading data codes can be sped up by using a region of interest. The more the region in which codes are searched can be restricted, the faster and more robust the search will be.

For detailed information see the description of this method on page 25.

17.2.7 Read Data Code(s)

The operator find_data_code_2d returns for every successfully decoded symbol the surrounding XLD contour in SymbolXLDs, a handle to a result structure, which contains additional information about the symbol as well as about the search and decoding process (ResultHandles), and the string that is encoded in the symbol (DecodedDataStrings). With the result handles and the operators get_data_code_2d_results and get_data_code_2d_objects, additional data about the extraction process can be accessed.

17.2.8 Inspect Data Code(s)

Using the handles of the successfully decoded symbols returned by find_data_code_2d, you can request additional information about the symbol and the finding process using the operators get_data_code_2d_results and get_data_code_2d_objects. This is useful both for process analysis and for displaying.

In addition, information about rejected candidates can also be queried by requesting the corresponding handles with `get_data_code_2d_results` using, e.g., the parameter values `'all_undecoded'` and `'handle'`.

The operator `get_data_code_2d_results` gives access to several alphanumerical results that were calculated while searching and reading the data code symbols. Besides basic information like the dimensions of the code, its polarity, or the found contrast, also the raw data can be accessed.

The operator `get_data_code_2d_objects` gives access to iconic objects that were created while searching and reading the data code symbols. Possible return values are the surrounding contours or the regions representing the foreground or background modules.

17.2.9 Check Print Quality

If your first aim is not to quickly read the 2D data code symbols but to check how good they were printed, you can query the print quality of a symbol (except for DotCode symbols) in accordance to the standards ISO/IEC 15415:2011 or AIM DPM-1-2006. For details, see the Solution Guide II-C in section 6 on page 45.

17.2.10 Visualize Results

Finally, you might want to display the images, the data code regions, and the decoded content.

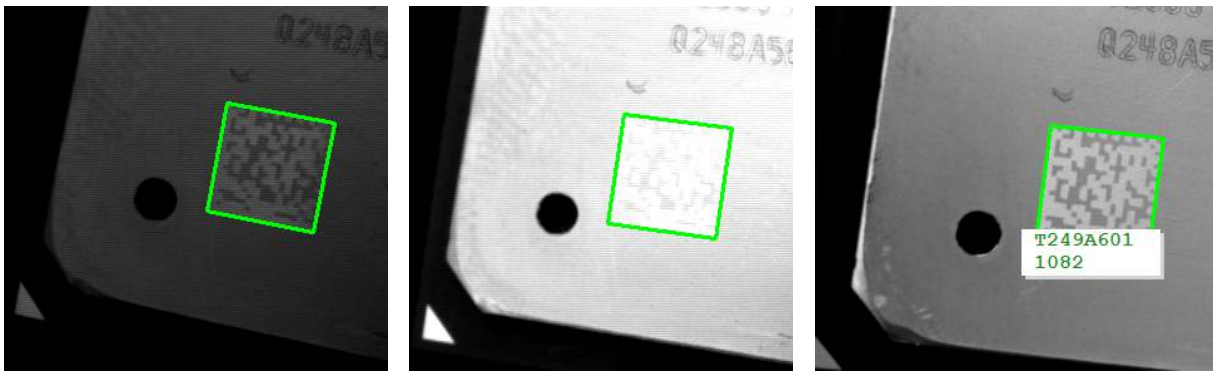
For detailed information see the description of this method on page 223.

17.3 Programming Examples

This section gives a brief introduction to the programming of the data code reader.

17.3.1 Training a Data Code Model

Example: `%HALCONEXAMPLES%/hdevelop/Identification/ecc200_training.hdev`



a) b) c)

Figure 17.2: (a) Dark training image; (b) bright training image; (c) read code.

In this example we show how easy it is to train a data code model, here to allow changes in the illumination of the images. To prepare the reading of the data codes, the following major steps are performed: First, a model for 2D data codes of type ECC 200 is created with `create_data_code_2d_model`.

```
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
```

Then, two sample images are loaded and passed to `find_data_code_2d` with the parameter value `'train'` to get the optimal parameters for finding the data code.

```
* Dark image
read_image (Image, 'datacode/ecc200/ecc200_cpu_007')
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, 'train', 'all', \
                   ResultHandles, DecodedDataStrings)
* Bright image
read_image (Image, 'datacode/ecc200/ecc200_cpu_008')
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, 'train', 'all', \
                   ResultHandles, DecodedDataStrings)
```

Inside the `for`-loop, images are read and `find_data_code_2d` is applied to read the code from the image. For further information, we also measure the time needed to run the operator `find_data_code_2d`. After that, the XLD contour of the symbol, the decoded data string, and the runtime are visualized.

```
for Index := 7 to 16 by 1
    read_image (Image, ImageFiles + Index$'.2d')
    dev_display (Image)
    *
    * Find and decode the data codes and measure the runtime
    count_seconds (T1)
    find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                       ResultHandles, DecodedDataStrings)
    count_seconds (T2)
    Time := 1000 * (T2 - T1)
    *
    * Display the results
    TitleMessage := 'Image ' + (Index - 6) + ' of ' + (ImageNum - 6)
    ResultMessage := 'Data code found in ' + Time$'.1f' + ' ms'
    display_found_data_codes (SymbolXLDs, WindowHandle, DecodedDataStrings, \
                              TitleMessage, ResultMessage, 'forest green', \
                              'black')
endfor
```

17.3.2 Reading 2D Data Codes on Chips

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Data-Codes/ecc200_optimized_settings.hdev`

This example program reads 2D data codes of type ECC200, which like the example described before are engraved in chips (see figure 17.3).

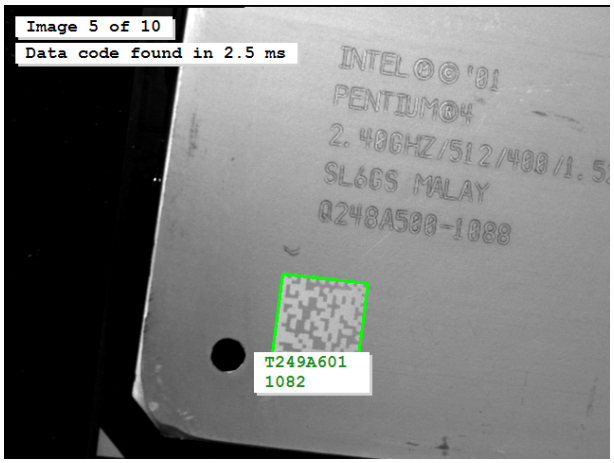


Figure 17.3: Decoded data code.

The example shows how to set optimized parameters for efficient data code reading. The code printed on a chip is always light on dark in this application and has a given size and number of modules. Also, the contrast is within a predefined range. By specifying these values for the model, the execution can be sped up significantly.

```
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
set_data_code_2d_param (DataCodeHandle, 'symbol_size', 18)
set_data_code_2d_param (DataCodeHandle, ['module_size_min', \
                        'module_size_max'], [4, 7])
set_data_code_2d_param (DataCodeHandle, 'module_gap', 'no')
set_data_code_2d_param (DataCodeHandle, 'polarity', 'light_on_dark')
set_data_code_2d_param (DataCodeHandle, 'mirrored', 'no')
set_data_code_2d_param (DataCodeHandle, 'contrast_tolerance', 'high')
set_data_code_2d_param (DataCodeHandle, 'candidate_selection', 'extensive')
set_data_code_2d_param (DataCodeHandle, 'module_grid', 'any')

find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], ResultHandles, \
                   DecodedDataStrings)
```

17.4 Advanced Topics

17.4.1 Use Timeout

With the operator `set_data_code_2d_param`, you can set a timeout. Then, the operator `find_data_code_2d` will return at the latest after the specified time. This mechanism is demonstrated for Matching (Shape-Based) in the example `%HALCONEXAMPLES%/hdevelop/Matching/Shape-Based/set_shape_model_timeout.hdev`.

Chapter 18

OCR (character classification)

Optical Character Recognition (OCR) is the technical term for reading, i.e., identifying symbols. In HALCON, OCR is defined as the task to assign an interpretation to regions of an image. This chapter considers methods where these regions typically represent single characters and therefore we consider this as reading single symbols. A different approach is introduced in Chapter Deep OCR on page 209.

In an offline phase, the characters are trained by presenting several samples for each character. In the online phase, the image is segmented to extract the regions representing the characters and then the OCR reader is applied to get the interpretation for each character.

Figure 18.1 shows the principal steps. The first part is offline and consists of collecting training samples and, after that, applying the training. The online part consists of extracting the characters and then reading them.

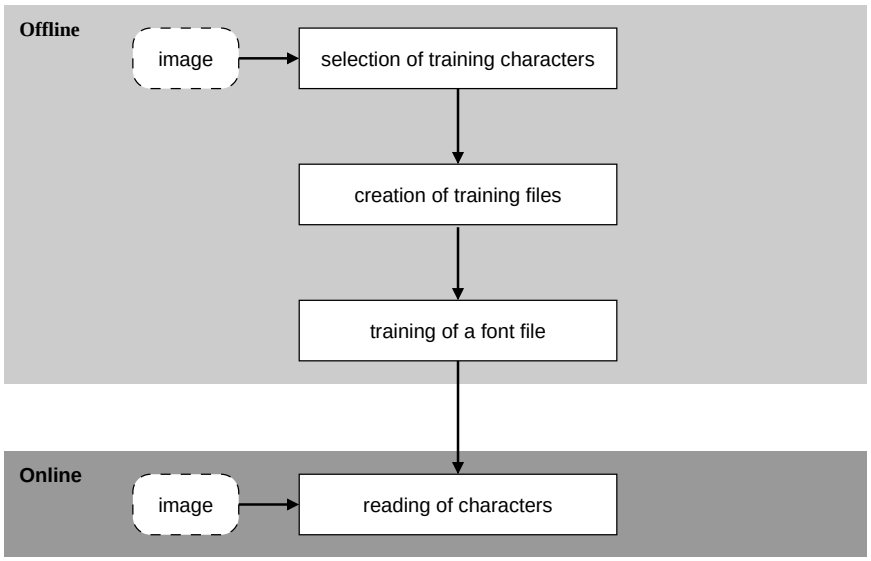


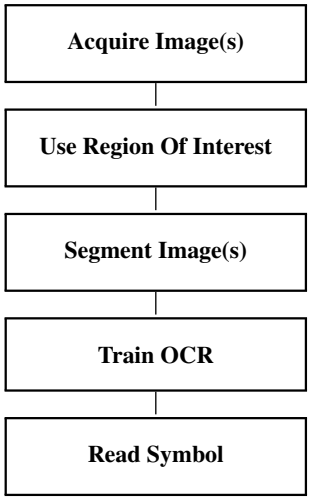
Figure 18.1: Main steps of OCR.

The advantage of OCR is the flexibility of the training, which allows to select features optimized for an application. Furthermore, you can choose between different classifiers that are based on latest technologies and provide the best possible performance.

As a further advantage, HALCON provides you with a set of pretrained fonts, which are based on a large amount of training data from various application areas. These fonts allow you to read text in documents, on pharmaceutical or industrial products, dot prints, and even handwritten numbers. Furthermore, HALCON includes pretrained fonts for OCR-A and OCR-B, and a generic font based on Convolutional Neural Networks (CNNs).

18.1 Basic Concept

The OCR is split into two major parts: training and reading. Each of these major parts requires additional preparation steps:



18.1.1 Acquire Image(s)

Both for the generation of training data and for the OCR itself images must be acquired.

For detailed information see the description of this method on page 21.

18.1.2 Segment Image(s)

Both for the training samples and for the online reading process, characters must be extracted from the image. This step is called segmentation. This means that the OCR operators like `do_ocr_single_class_svm` do not search for the characters within a given region of interest, but expect a segmented region, which then will be classified.

If the samples for training are taken from real application images, the same segmentation method will be applied for both training and reading. If the training images are more "artificial", a simpler method might be used to segment the training images.

18.1.3 Train OCR

The training consists of two important steps: First, a number of samples for each character is selected and stored in so-called training files. In the second step, these files are input for a newly created OCR classifier.

As already noted, HALCON provides pretrained fonts, i.e., ready-to-use classifiers, which already solve many OCR applications. These fonts can be found in the subdirectory `ocr` of the directory where you installed HALCON.

18.1.4 Read Symbol

For reading, you only have to read the trained classifier from disk, segment the characters from the image, and use the segmented characters as input for one of the reading operators that will be discussed later.

18.1.5 A First Example

An example for this basic concept is the following program, which uses one of the pretrained fonts provided by HALCON to read the numbers in the image depicted in figure 18.2.

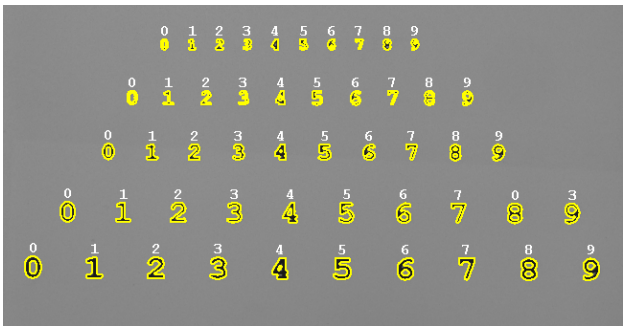


Figure 18.2: Applying a pretrained classifier.

First, the pretrained font `Document_0-9` is read using `read_ocr_class_mlp`. As no file extension is specified, it is searched for a file with the MLP specific extension ".omc" or with the extension ".fnt".

```
FontFile := 'Document_0-9_NoRej'
read_ocr_class_mlp (FontFile, OCRHandle)
```

Then, the numbers are segmented using `threshold` and `connection`. Because the order is irrelevant here, no further processing is applied.

```
read_image (Image, 'numbers_scale')
threshold (Image, Region, 0, 125)
connection (Region, Characters)
```

Finally, the numbers are read in a for-loop. The operator `do_ocr_single_class_mlp` takes the single region, the image, and the OCR handle as input. As a result the best and the second best interpretation together with the confidences are returned.

```
count_obj (Characters, Number)
dev_set_color ('white')
for i := 1 to Number by 1
    select_obj (Characters, SingleChar, i)
    do_ocr_single_class_mlp (SingleChar, Image, OCRHandle, 2, Class, \
                             Confidence)
endfor
```

Note that this examples was chosen for didactic reasons only. It clearly shows the two main parts of each OCR application: segmentation and classification. Typically, the Automatic Text Reader (see section 18.2.6 on page 187) should be used for OCR applications, because it is much easier to use. The Automatic Text Reader combines the two steps segmentation and classification into one call of the operator `find_text`. The above example can then be reduced to a few lines of code (see section 18.3.3 on page 193).

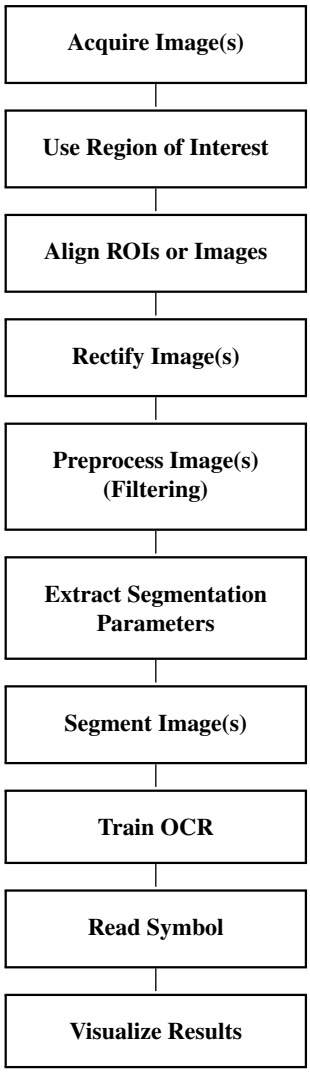
18.2 Extended Concept

When we look more closely at the OCR, many possibilities for adaptation to specific applications become apparent. For example, we have to consider an efficient way of collecting samples as well as correct parameters for the training. In the online phase, an optimal segmentation method is required to extract all characters in a robust manner.

18.2.1 Use Region of Interest

The OCR can be sped up by using a region of interest. The more the region in which the characters are searched can be restricted, the faster and more robust the search will be.

For detailed information see the description of this method on page 25.



18.2.2 Align ROIs or Images

Because the reading of characters is not invariant to rotation, it may be necessary to correct the orientation of the image. This can be achieved either by directly determining the orientation of the text using the operator `text_line_orientation`, or by locating another object. Then, the part of the image containing the characters is cropped and aligned using the orientation of the text or the found object.

How to perform alignment using shape-based matching is described in the Solution Guide II-B in section 2.5.3.2 on page 35.

18.2.3 Rectify Image(s)

Similarly to alignment, it may be necessary to rectify the image, e.g., to remove perspective distortions. For slanted characters, the slant angle can be determined using the operator `text_line_slant`.

Detailed information about rectifying images can be found in the Solution Guide III-C in section 3.4 on page 80.

18.2.4 Preprocess Image(s) (Filtering)

Sometimes, the characters may be difficult to extract because of noise, texture, or overlaid structures. Here, operators like `mean_image` or `gauss_filter` can be used to eliminate noise. A fast but slightly less perfect alternative to `gauss_filter` is `binomial_filter`. The operator `median_image` is helpful for suppressing small spots or thin lines. The operator `dots_image` is optimized to emphasize a dot-print while suppressing other

structures in the image. Gray value morphology can be used to eliminate noise structures and to adapt the stroke width of characters.

18.2.5 Extract Segmentation Parameters

When segmenting the characters with blob analysis, instead of using fixed threshold values, the values can be extracted dynamically for each image. For more details, please refer to the description of this step on page 35. Another possibility is the use of the Automatic Text Reader, which is described below.

18.2.6 Segment Image(s)

For the segmentation, various methods can be used. The **Automatic Text Reader** is very easy to use and provides robust results. It combines the two steps segmentation and classification into one call of the operator `find_text`. The **Manual Text Finder** can read engraved text, which cannot be read by the Automatic Text Reader, but it needs a greater effort in setting the parameters. Therefore, the Automatic Text Reader should be used, if possible. Both methods use a text model, which can be specified precisely. Two other common segmentation methods are described under **'General Character Segmentation'**.

Segmentation with the Automatic Text Reader

The Automatic Text Reader segments and classifies text robustly, typically without the need for extensive parameter tuning. `%HALCONEXAMPLES%/solution_guide/basics/simple_reading.hdev` (see page 193) and `%HALCONEXAMPLES%/hdevelop/Applications/OCR/bottle.hdev` (see page 193) provide a good starting point to become familiar with the Automatic Text Reader.

For using the Automatic Text Reader, a model must be created with `create_text_model_reader` with the parameter Mode set to *'auto'*. Here, an OCR classifier must already be passed. Segmentation parameters can then be specified with the operator `set_text_model_param` and can be queried with `get_text_model_param`. After this preparation, the text can be read with the operator `find_text`. This operator selects character candidates based on region and gray-value features and verifies them with the given OCR classifier. The character candidates are then further combined to lines which are subsequently tested if they qualify as a text line.

If the text must match a certain pattern or structure, the parameter `'text_line_structure'` of the operator `set_text_model_param` can be set, which determines the structure, i.e., the number of characters for each character block of the text that shall be detected.

The Automatic Text Reader assumes approximately horizontal text. If the text is not horizontally aligned, the operators `text_line_orientation` and `rotate_image` can be used before the use of `find_text`.

The result of `find_text` is returned in `TextResultID`, which can be queried with `get_text_result` and `get_text_object`, respectively. `get_text_result` returns, e.g., the classification result. `get_text_object` returns the iconic result of the Automatic Text Reader, i.e., the character regions. To delete the result and the text model, use the operators `clear_text_result` and `clear_text_model`, respectively.

Please refer to the Reference Manual of the above mentioned operators for more information about their parameters.

Segmentation with the Manual Text Finder

If engraved text must be segmented or if no suitable OCR classifier can be provided, the Automatic Text Reader cannot be used. Instead, the Manual Text Finder can be used in these cases.

In order to segment images robustly, the parameters of the Manual Text Finder should be set carefully. For a practical introduction to the Manual Text Finder, please refer to the HDevelop example `%HALCONEXAMPLES%/hdevelop/Applications/OCR/find_text_dongle.hdev` and the corresponding example description page 196.

For using the Manual Text Finder, a model must be created with `create_text_model_reader` with the parameter Mode set to *'manual'*. Note that no OCR classifier must be passed in this case. Segmentation parameters should then be specified with the operator `set_text_model_param` and can be queried with `get_text_model_param`. Note that the names of all parameters that can be used with the Manual Text Finder start with `'manual_'`. After this

preparation, the text can be segmented with the operator find_text. This operator selects character candidates based on region and gray-value features. The character candidates are then further combined to lines which are subsequently tested if they qualify as a text line.

If the text must match a certain pattern or structure, the parameter 'manual_text_line_structure' of the operator set_text_model_param can be set, which determines the structure, i.e., the number of characters for each character block of the text that shall be detected.

The Manual Text Finder assumes approximately horizontal text. If the text is not horizontally aligned, the operators text_line_orientation and rotate_image can be used before the use of find_text.

The result of find_text is returned in TextResultID which can be queried with get_text_result and get_text_object, respectively. get_text_result returns, e.g., the number of found text lines. get_text_object returns the iconic result of the Manual Text Finder, i.e., the character regions, which can then be classified with a suitable OCR classifier. To delete the result and the text model, use the operators clear_text_result and clear_text_model, respectively.

Please refer to the Reference Manual of the above mentioned operators for more information about their parameters.

General Character Segmentation

For the common character segmentation you can either use the operator segment_characters to get a region containing all character candidates and then apply select_characters to select those parts of the region that are candidates for the individual characters, or you use blob analysis. There, the most simple method is the operator threshold, with one or more gray value ranges specifying the regions that belong to the foreground objects. Another very common method is dyn_threshold. Here, a second image is passed as a reference. With this approach a local instead of a global threshold is used for each position. Further information can be found in the description of this step for blob analysis on page 34.

18.2.7 Train OCR

Figure 18.3 shows an overview on the generation of the training files: First, the characters from sample images must be extracted using a segmentation method (see above). To each of the single characters a name must be assigned. This can be done either by typing it in, by a programmed input in the case of a well-structured image (having, e.g., multiple samples of each character in different lines), or by reading the character names from file. Then, the regions together with their names are written into training files. The most convenient operator to do this is append_ocr_trainf. Before applying the training, we recommend to check the correctness of the training files. This can, e.g., be achieved by using the operator read_ocr_trainf combined with visualization operators.

Note that you can also train your own system fonts. By altering and distorting the characters of a font, you can increase the number of different training samples for each class and thus also increase the detection rate. For this, the Training File Browser of HDevelop's OCR Assistant can be used. See HDevelop User's Guide, section 6.10 on page 85 for a detailed description of the Training File Browser. Furthermore, also the example program %HALCONEXAMPLES%/hdevelop/Applications/OCR/generate_system_font.hdev shows how to derive training data and an OCR classifier from system fonts.

The actual training is depicted in figure 18.4. First, a new classifier is created. There are four different OCR classifiers available: a neural network (multi-layer perceptron or MLP) classifier, a classifier based on support vector machines (SVM), a classifier based on the k-nearest neighbor approach (k-NN), and the box classifier.

Note that if you want to use the Automatic Text Reader for the segmentation and classification of text, you must provide an MLP-based OCR classifier. Otherwise, an OCR classifier based on MLP, SVM, or k-NN can be used. The k-NN has advantages when only few samples are available, but is outperformed by MLP and SVM in typical OCR applications. Thus, only MLP and SVM are described further in this manual. Please refer to the Solution Guide II-D, section 7.5 on page 86 for more information on how to use the k-NN classifier for OCR applications.

The two recommended classifiers differ as follows: The MLP classifier is fast at classification, but for a large training set slow at training (compared to the classifier based on SVM). If the training can be applied offline and thus is not time critical, MLP is a good choice. The classifier based on SVM leads to slightly better recognition rates than the MLP classifier and is faster at training (especially for large training sets). But, compared to the MLP classifier, the classification needs more time.

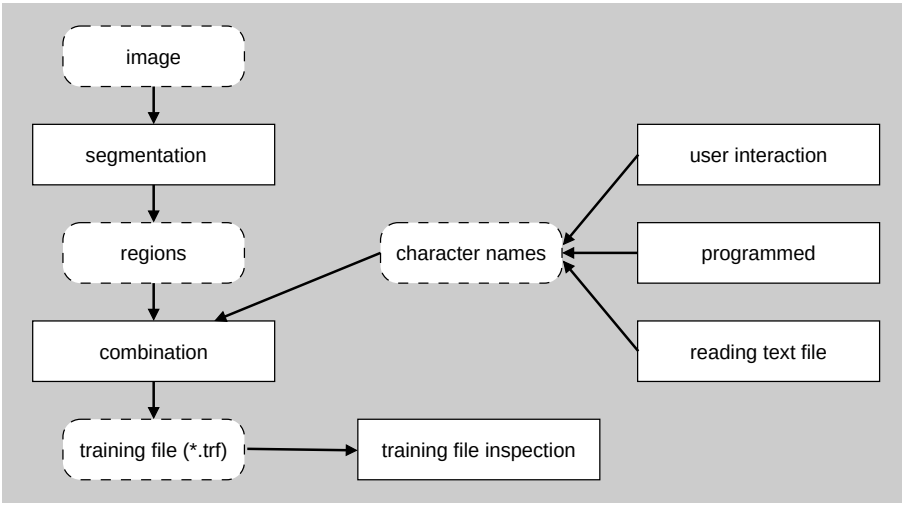


Figure 18.3: Creating training files.

Dependent on the chosen classifier, you create the classifier using `create_ocr_class_mlp` or `create_ocr_class_svm`. Then, the training is applied using `trainf_ocr_class_mlp` or `trainf_ocr_class_svm`. After the training, you typically save the classifier to disk for later use by `write_ocr_class_mlp` or `write_ocr_class_svm`.

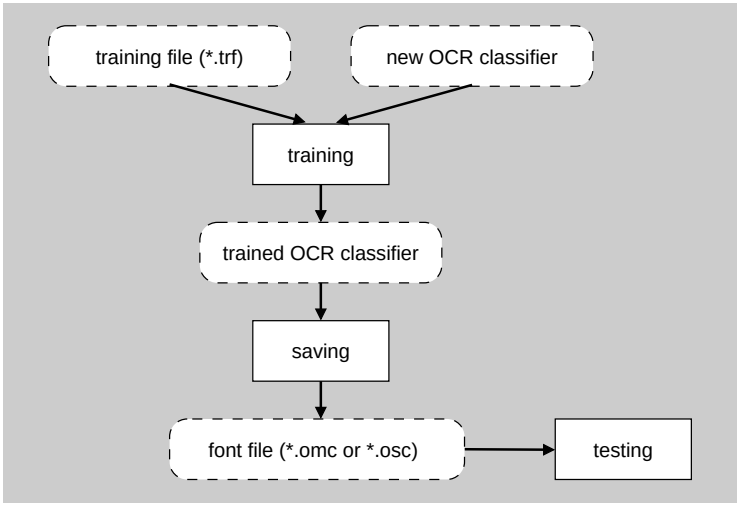


Figure 18.4: Training an OCR classifier.

18.2.8 Read Symbol

Figure 18.5 shows an overview on the reading process. First, the characters must be extracted using an appropriate segmentation method. Here, you must use a method that returns the characters in a form similar to the ones used for training. After reading the classifier (font file) from file (`read_ocr_class_mlp` or `read_ocr_class_svm`), the classifier can be used for reading. The Automatic Text Reader performs the two steps segmentation and classification together in one single step.

For reading multiple operators are provided: In the easiest case, multiple characters are passed to the reading operators (`do_ocr_multi_class_mlp` or `do_ocr_multi_class_svm`). Here, for each region the corresponding name and the confidence are returned. Sometimes, it can be necessary not only to obtain the characters with the highest confidence but also others with lower confidences. A zero, e.g., might easily be mistaken for the character “O”. This information is returned by the operators `do_ocr_single_class_mlp` and `do_ocr_single_class_svm`.

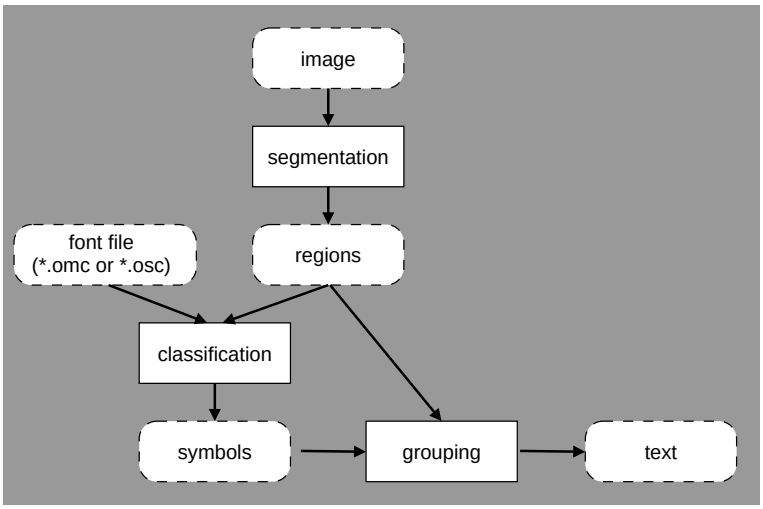


Figure 18.5: Reading characters.

As a final step it might be necessary to group digits to numbers or characters to words. This can be realized with the region processing operators like those described for the method blob analysis on page 36.

Additionally, HALCON provides operators for a syntactic and lexicon-based auto-correction. For example, you can use the operator `do_ocr_word_mlp` instead of `do_ocr_multi_class_mlp` to find sets of characters, i.e., words, that match a regular expression or that are stored in a lexicon, which was created by `create_lexicon` or imported by `import_lexicon`.

18.2.9 Visualize Results

Finally, you might want to display the images, the blob (regions), and the result of the reading process.

For detailed information see the description of this method on page 223.

18.3 Programming Examples

This section gives a brief introduction to using HALCON for OCR. All important steps from training file generation over training to reading are presented.

18.3.1 Generating a Training File

Example: `%HALCONEXAMPLES%/solution_guide/basics/gen_training_file.hdev`

Figure 18.6 shows a training image from which the characters in the third line are used as training samples. For this example image, the segmentation is very simple because the characters are significantly darker than the background. Therefore, `threshold` can be used.

The number of the line of characters that is used for training is specified by the variable `TrainingLine`. To select this line, first the operator `closing_rectangle1` is used to combine characters horizontally into lines. These lines are then converted to their connected components with `connection`. Out of all lines the relevant one is selected using `select_obj`. By using `intersection` with the original segmentation and the selected line as input, the characters for training are returned. These are sorted from left to right, using `sort_region`.

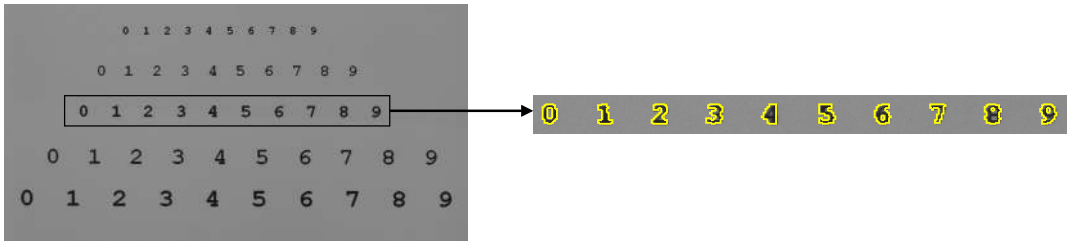


Figure 18.6: Collecting characters for a training file.

```
TrainingLine := 3
threshold (Image, Region, 0, 125)
closing_rectangle1 (Region, RegionClosing, 70, 10)
connection (RegionClosing, Lines)
select_obj (Lines, Training, TrainingLine)
intersection (Training, Region, TrainingChars)
connection (TrainingChars, ConnectedRegions)
sort_region (ConnectedRegions, SortedRegions, 'first_point', 'true', \
             'column')
```

Now, the characters can be stored in the training file. As a preparation step a possibly existing older training file is deleted. Within a loop over all characters the single characters are selected. The variable `Chars` contains the names of the characters as a tuple of strings. With the operator `append_ocr_trainf` the selected regions, together with the gray values and the corresponding name, are added to the training file.

```
Chars := ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
TrainFile := 'numbers.trf'
dev_set_check ('~give_error')
delete_file (TrainFile)
dev_set_check ('give_error')
for i := 1 to 10 by 1
    select_obj (SortedRegions, TrainSingle, i)
    append_ocr_trainf (TrainSingle, Image, Chars[i - 1], TrainFile)
endfor
```

18.3.2 Creating and Training an OCR Classifier

Example: `%HALCONEXAMPLES%/solution_guide/basics/simple_training.hdev`

Having prepared the training file, the creation and training of an OCR classifier is very simple. First, the names of the training file and the final font file are determined. Typically, the same base with different extensions is used. We recommend to use ".trf" for training files. For font files i.e., for OCR classifiers, we recommend to use ".obc" for the box classifier (which is not recommended anymore), ".omc" for the neural network classifier , and ".osc" for the classifier based on support vector machines. If no extension is specified during the reading process, for the box or neural network classification it is also searched for files with the extension ".fnt", which was common for both classifiers in earlier HALCON versions.

To create an OCR classifier, some parameters need to be determined. The most important one is the list of all possible character names. This list can easily be extracted from the training file by using the operator `read_ocr_trainf_names`.

```
TrainFile := 'numbers.trf'
read_ocr_trainf_names (TrainFile, CharacterNames, CharacterCount)
```

Another important parameter is the number of nodes in the hidden layer of the neural network. In this case, it is set to 20. As a rule of thumb, this number should be in the same order as the number of different symbols. Besides these two parameters, here only default values are used for `create_ocr_class_mlp`. The training itself is applied using `trainf_ocr_class_mlp`. We recommend to simply use the default values here as well.

```
NumHidden := 20
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, \
                      NumHidden, 'none', 1, 42, OCRHandle)
trainf_ocr_class_mlp (OCRHandle, TrainFile, 200, 1, 0.01, Error, ErrorLog)
```

Finally, the classifier is stored to disk.

```
FontFile := 'numbers.omc'
write_ocr_class_mlp (OCRHandle, FontFile)
```

Note that for more complex OCR classifiers, i.e., especially, if the training data contains also very noisy and deformed samples, it is recommended to create an MLP-based OCR classifier with regularization of the internal weights (see set_regularization_params_ocr_class_mlp). This enhances the generalization ability of the classifier and prevents an over-fitting to individual degraded training samples.

If an OCR classifier is created for the Automatic Text Reader, it is recommended to additionally define a rejection class with set_rejection_params_ocr_class_mlp, which helps to distinguish characters from background clutter.

18.3.2.1 OCR Classifiers with Regularization and Rejection Class

It is also possible to create and train a classifier with regularized weights as well as rejection class.

Regularized weights can improve the classification:

- If an unregularized MLP makes an error, the confidence of the wrong result will often be very high.
- If a regularized MLP makes an error, it returns an intuitive confidence. This indicates a much better generalization capability.

Parameters for regularization can be set and queried with the operators

- set_regularization_params_ocr_class_mlp and
- get_regularization_params_ocr_class_mlp.

Please refer to the reference documentation of these operators for more information.

How to set the parameters for creating and training of classifiers with regularization is shown in the following HDevelop examples:

- The example %HALCONEXAMPLES%/hdevelop/Applications/OCR/Neural-Nets/regularized_ocr_mlp.hdev creates test samples with heavy distortions that are far outside the range of trained distortions to test the generalization capabilities of the MLP and the regularized MLP.
- The example %HALCONEXAMPLES%/hdevelop/Classification/Neural-Nets/mlp_regularization.hdev shows the effect of regularizing an MLP using two-dimensional data.

A rejection class may be useful since it returns symbols in an image that could not be successfully read either because they are not symbols but, e.g., noise, or because there is a problem with the classification.

Parameters for the rejection class can be set and queried with the operators

- set_rejection_params_ocr_class_mlp and
- get_rejection_params_ocr_class_mlp.

Please refer to the reference documentation of these operators for more information.

The example program %HALCONEXAMPLES%/hdevelop/Applications/Classification/Neural-Nets/set_rejection_params_class_mlp.hdev shows how to use a rejection class for an MLP for classifying two-dimensional data.

18.3.3 Reading Numbers

Example: %HALCONEXAMPLES%/solution_guide/basics/simple_reading.hdev

This example program demonstrates how to read simple text with the Automatic Text Reader (see section 18.2.6 on page 187) using a pretrained OCR font. It reads the numbers in the image numbers_scale depicted in figure 18.2. Instead of a manual segmentation of the numbers with the operators threshold and connection followed by a classification of the segmented regions, the Automatic Text Reader is used to read the numbers in one single step without the need for any parameter tuning, simply with the operator find_text.

```
create_text_model_reader ('auto', 'Document_0-9_NoRej', TextModel)
find_text (Image, TextModel, TextResultID)
get_text_result (TextResultID, 'class', Classes)
```

18.3.4 "Best Before" Date

Example: %HALCONEXAMPLES%/hdevelop/Applications/OCR/bottle.hdev

The task of this example is to inspect the "best before" date on the bottle depicted in figure 18.7.

a) b)

Figure 18.7: (a) Original image; (b) read date.

Again, this task is solved with the Automatic Text Reader (see section 18.2.6 on page 187). Because a lot of text is visible in the image, it is necessary to set some parameters of the text model to restrict the reading result appropriately.

First, the Automatic Text Reader is created with a pretrained OCR font:

```
create_text_model_reader ('auto', FontName, TextModel)
```

The minimum stroke width is increased to suppress all the text that is visible in the surroundings of the "best before" date:

```
set_text_model_param (TextModel, 'min_stroke_width', 6)
```

The known structure of the "best before" date is set to ensure that only text is read that match this structure:

```
set_text_model_param (TextModel, 'text_line_structure', '2 2 2')
```

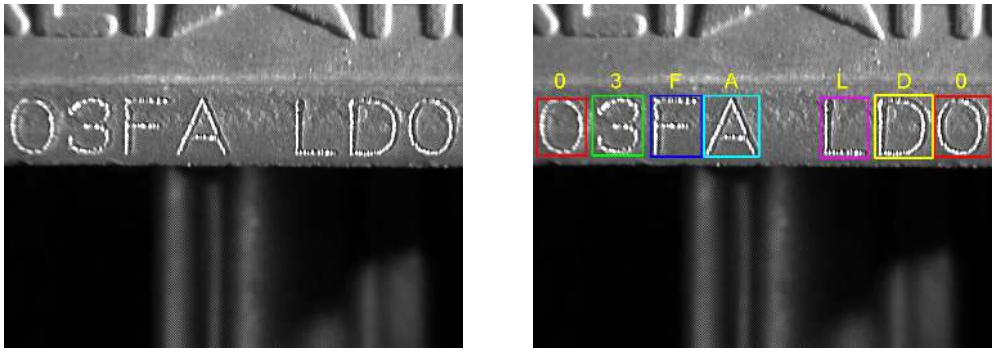
And finally, the text is segmented and read:

```
find_text (Bottle, TextModel, TextResultID)
```

18.3.5 Reading Engraved Text

Example: %HALCONEXAMPLES%/hdevelop/Applications/OCR/engraved.hdev

The task of this example is to read the engraved text on the metal surface depicted in figure 18.8.



a) b)

Figure 18.8: (a) Original image; (b) read characters.

The segmentation is solved by using advanced blob analysis: The characters cannot simply be extracted by selecting dark or light pixels. Instead, a simple segmentation would yield only fractions of the characters together with noise objects. Preprocessing the image using gray value morphology allows to segment the real characters.

```
gray_range_rect (Image, ImageResult, 7, 7)
invert_image (ImageResult, ImageInvert)
threshold (ImageResult, Region, 128, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 1000, 99999)
sort_region (SelectedRegions, SortedRegions, 'first_point', 'true', \
             'column')
```

Finally, the actual reading is performed.

```
read_ocr_class_mlp (FontName, OCRHandle)
for I := 1 to Number by 1
    select_obj (SortedRegions, ObjectSelected, I)
    do_ocr_single_class_mlp (ObjectSelected, ImageInvert, OCRHandle, 1, \
                             Class, Confidence)
endfor
```

18.3.6 Reading Forms

Example: %HALCONEXAMPLES%/hdevelop/Applications/OCR/ocrcolor.hdev

The task of this example is to extract the symbols in the form. A typical problem is that the symbols are not printed in the correct place, as depicted in figure 18.9.

To solve the problem of numbers printed on lines, color is used here: The hue value of the characters differs from the hue of the form. The color classification method is a very simple way to save execution time: In contrast to more difficult color processing problems, here it is sufficient to consider the difference of the red and green channel combined with the intensity.

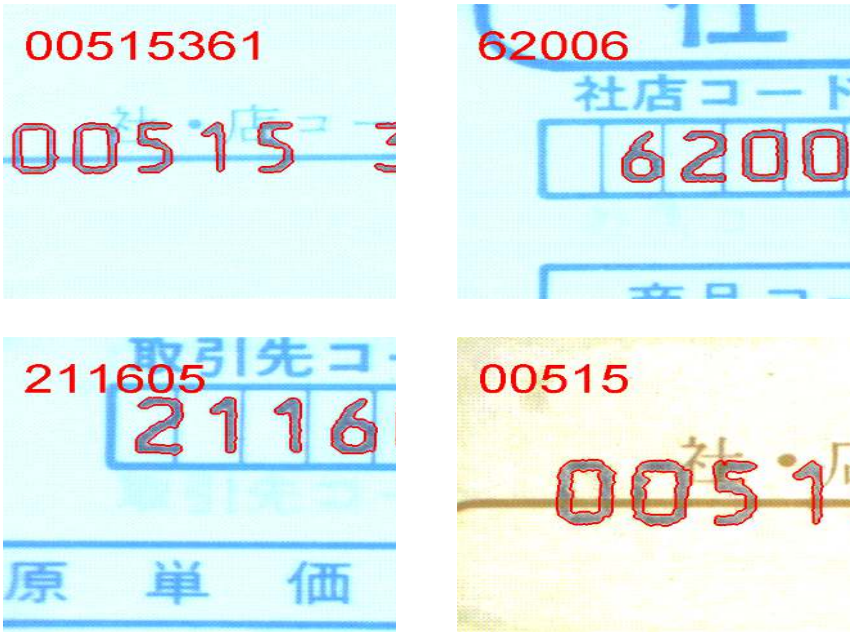


Figure 18.9: Example images for OCR.

```
threshold (Green, ForegroundRaw, 0, 220)
sub_image (RedReduced, GreenReduced, ImageSub, 2, 128)
mean_image (ImageSub, ImageMean, 3, 3)
binary_threshold (ImageMean, Cluster1, 'smooth_histo', 'dark', \
                  UsedThreshold)
difference (Foreground, Cluster1, Cluster2)
concat_obj (Cluster1, Cluster2, Cluster)
opening_circle (Cluster, Opening, 2.5)
```

The selected pixels are grouped and post-processed with morphological operators.

```
closing_rectangle1 (NumberRegion, NumberCand, 1, 20)
difference (Image, NumberCand, NoNumbers)
connection (NumberRegion, NumberParts)
intensity (NumberParts, Green, MeanIntensity, Deviation)
expand_gray_ref (NumberParts, Green, NoNumbers, Numbers, 20, 'image', \
                 MeanIntensity, 48)
union1 (Numbers, NumberRegion)
connection (NumberRegion, Numbers)
```

For reading, it is important not to use the gray values of the background because of the changes in color. To solve this, only region features are used for the font, i.e., the regions are used to create an artificial image that is printed dark on light.

```
paint_region (NoNumbers, Green, ImageOCRRaw, 255, 'fill')
paint_region (NumberRegion, ImageOCRRaw, ImageOCR, 0, 'fill')
```

The actual character classification is performed in the artificial image.

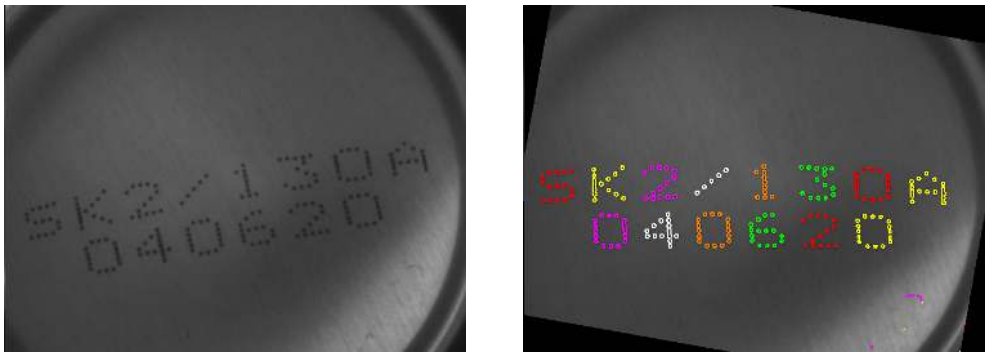
```
read_ocr_class_mlp ('Industrial_0-9_NoRej', OCRHandle)
    do_ocr_multi_class_mlp (FinalNumbers, ImageOCR, OCRHandle, RecChar, \
                            Confidence)
```

18.3.7 Segment and Select Characters

18.3.7.1 Segment Rotated Characters

Example: %HALCONEXAMPLES%/hdevelop/OCR/Segmentation/select_characters.hdev

This example shows how to easily segment the characters of a rotated dot print using the segmentation operators that are provided especially for OCR (see figure 18.10).



a) b)

Figure 18.10: (a) Original image; b) selected characters.

First, the image is read from file. As the print is rotated, the orientation of the text line is determined via text_line_orientation. The obtained angle is used to rotate the image so that the print becomes horizontal.

```
read_image (Image, 'dot_print_rotated/dot_print_rotated_' + J$'02d')
text_line_orientation (Image, Image, 50, rad(-30), rad(30), \
                       OrientationAngle)
rotate_image (Image, ImageRotate, -OrientationAngle / rad(180) * 180, \
              'constant')
```

Then, the operators segment_characters and select_characters are applied to first segment the region of the complete print and then select those parts of the region that are candidates for individual characters. In contrast to a classical segmentation using blob analysis, here the regions of the individual characters are found although they still consist of components that are not connected.

```
segment_characters (ImageRotate, ImageRotate, ImageForeground, \
                    RegionForeground, 'local_auto_shape', 'false', 'true', \
                    'medium', 25, 25, 0, 10, UsedThreshold)
select_characters (RegionForeground, RegionCharacters, 'true', \
                   'ultra_light', 60, 60, 'false', 'false', 'none', 'true', \
                   'wide', 'true', 0, 'completion')
```

The extracted regions can now be used for an OCR application like those described above.

18.3.7.2 Segment Characters with the Manual Text Finder

Example: %HALCONEXAMPLES%/hdevelop/Applications/OCR/find_text_dongle.hdev

This example demonstrates how to segment characters printed in dot print on a dongle with the operator find_text before performing OCR. For more background information, please refer to the section about the segmentation with the Smart Text Finder in section 18.2.6 on page 187.

First, image and classifier are read.

```
read_image (Image, 'ocr/dongle_01')
read_ocr_class_mlp ('DotPrint_NoRej', OCRHandle)
```

Then a text model (TextModel) is created with the operator create_text_model_reader and the text properties are specified with the operator set_text_model_param and TextModel.

```
create_text_model_reader ('manual', [], TextModel)
*
set_text_model_param (TextModel, 'manual_char_width', 24)
set_text_model_param (TextModel, 'manual_char_height', 33)
set_text_model_param (TextModel, 'manual_is_dotprint', 'true')
set_text_model_param (TextModel, 'manual_max_line_num', 2)
set_text_model_param (TextModel, 'manual_return_punctuation', 'false')
set_text_model_param (TextModel, 'manual_return_separators', 'false')
set_text_model_param (TextModel, 'manual_stroke_width', 4)
set_text_model_param (TextModel, 'manual_eliminate_horizontal_lines', \
                      'true')
```

The structures of the text lines are defined with the parameter 'manual_text_line_structure'. E.g., 'manual_text_line_structure' '6 1 8' means that the text has three blocks consisting of 6, 1 and 8 character(s). In order to define more than one structure, an index number can be added to the parameter name like, e.g., 'manual_text_line_structure_2'.

Note that for the second line two structures are defined, because sometimes the '/' is classified as separator and sometimes as character. Furthermore, to increase the robustness of the character recognition, a regular expression is defined, which will later be used by do_ocr_word_mlp.

```
set_text_model_param (TextModel, 'manual_text_line_structure_0', '6 1 8')
set_text_model_param (TextModel, 'manual_text_line_structure_1', '8 10')
set_text_model_param (TextModel, 'manual_text_line_structure_2', '19')
TextPattern1 := '(FLEXID[0-9][A-Z][0-9]{3}[A-F0-9]{4})'
TextPattern2 := '([A-Z]{3}[0-9]{5}.?[A-Z][0-9]{4}[A-Z][0-9]{4})'
Expression := TextPattern1 + '|' + TextPattern2
```

For preprocessing the domain is reduced to the dark area where the text is assumed to be found.

```
binary_threshold (Image, Region, 'max_separability', 'dark', UsedThreshold)
opening_rectangle1 (Region, RegionOpening, 400, 50)
erosion_rectangle1 (RegionOpening, RegionOpening, 11, 11)
connection (RegionOpening, ConnectedRegions)
select_shape_std (ConnectedRegions, SelectedRegion, 'max_area', 70)
reduce_domain (Image, SelectedRegion, ImageReduced)
```

The contrast is then improved with scale_image_max and the image is horizontally aligned.

```
scale_image_max (ImageReduced, ImageScaleMax)
text_line_orientation (SelectedRegion, ImageScaleMax, 35, rad(-30), rad(30), \
                       OrientationAngle)
rotate_image (ImageScaleMax, ImageRotate, deg(-OrientationAngle), \
              'constant')
```

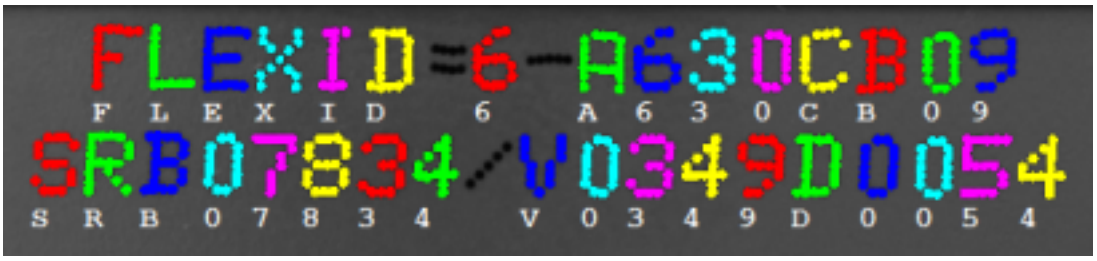


Figure 18.11: Reading characters on a dongle with the Smart Text Finder.

The text is found and the results are displayed for every segmented region. The OCR uses regular expressions to read the text more robustly.

get_text_result returns the number of lines with 'manual_num_lines'. It can also be used to query 'manual_thresholds' if the parameter 'manual_persistance' of the operator set_text_model_param was activated. get_text_object can be used to return 'manual_all_lines' or in this case with 'manual_line', it queries specific lines. The operator also returns 'manual_compensated_image' if 'manual_persistance' was activated.

```
get_text_result (TextResult, 'manual_num_lines', NumLines)
dev_display (ImageRotate)
for J := 0 to NumLines - 1 by 1
    get_text_object (Line, TextResult, ['manual_line',J])
    do_ocr_word_mlp (Line, ImageRotate, OCRHandle, Expression, 3, 5, Class, \
                     Confidence, Word, Score)
```

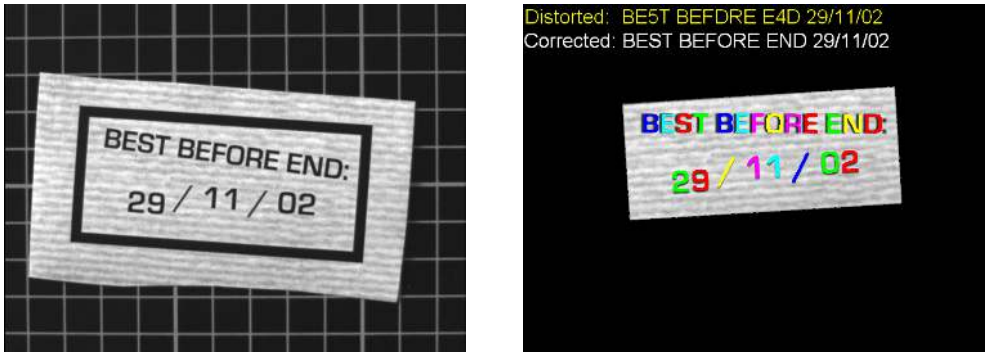
The results are then displayed.

```
smallest_rectangle1 (Line, Row1, Column1, Row2, Column2)
count_obj (Line, NumberOfCharacters)
dev_set_colored (6)
dev_display (Line)
dev_set_color ('white')
for K := 1 to NumberOfCharacters by 1
    select_obj (Line, Character, K)
    set_tposition (WindowHandle, Row2[0] + 4, Column1[K - 1])
endfor
```

18.3.8 Syntactic and Lexicon-Based Auto-Correction of OCR Results

Example: %HALCONEXAMPLES%/hdevelop/OCR/Neural-Nets/label_word_process_mlp.hdev

This example reads the "best before" date depicted in figure 18.12. To correct an incorrect OCR result for the upper text line, a lexicon-based auto-correction is used. Errors can occur, e.g., because of the similarity of characters, e.g., between the character O and the number 0. For the second text line, regular expressions are used to ensure that the result has the correct format.



a) b)

Figure 18.12: (a) Original image; (b) text is corrected using syntactic and lexicon-based auto-correction.

First, the pretrained font Industrial is read as preparation for the actual OCR reading. For the upper line of the text, the three expected words are stored in a lexicon that is created with create_lexicon and will be used later. Then, the images are read, an ROI for the print is generated and aligned, and the regions for the characters are extracted and stored in the variable SortedWords using blob analysis.

```
read_ocr_class_mlp ('Industrial_NoRej', OCRHandle)
create_lexicon ('label', ['BEST', 'BEFORE', 'END'], LexiconHandle)
for i := 1 to 9 by 1
    read_image (Image, 'label/label_0' + i + '.png')
  ... get ROI, align print, and extract regions ...
```

Now, the text line is read twice with the operator do_ocr_word_mlp. The first time it is read without syntactic or lexicon-based auto-correction, and the second time the result is corrected by matching it to the lexicon entries.

```
sort_region (CharactersWords, SortedWords, 'character', 'true', 'row')
gen_empty_obj (Word)
Text := ''
OriginalText := ''
for j := 1 to |Column| - 1 by 1
    select_obj (SortedWords, Character, j)
    concat_obj (Word, Character, Word)
    if (j == |Column| or (Column[j] - Column[j - 1]) > 30)
        do_ocr_word_mlp (Word, ImageOCR, OCRHandle, '.*', 1, 5, Class, \
                         Confidence, WordText, WordScore)
        OriginalText := OriginalText + ' ' + WordText
        do_ocr_word_mlp (Word, ImageOCR, OCRHandle, '<label>', 1, 5, Class, \
                         Confidence, WordText, WordScore)
        Text := Text + ' ' + WordText
        gen_empty_obj (Word)
    endif
endfor
```

The second text line, i.e., the actual date, is read using a regular expression that ensures the correct format for the date. This is done to suppress structures that may be extracted as candidates for characters but syntactically do not fit into the searched string.

```
sort_region (CharactersDate, SortedDate, 'character', 'true', 'row')
do_ocr_word_mlp (SortedDate, ImageOCR, OCRHandle, '.*', 5, 5, Class, \
                 Confidence, OriginalDateText, DateScore)
do_ocr_word_mlp (SortedDate, ImageOCR, OCRHandle, \
                 '^([0-2][0-9]|30|31)/(0[1-9]|10|11|12)/0[0-5]$', 10, 5, \
                 Class, Confidence, DateText, DateScore)
```

18.4 Relation to Other Methods

18.4.1 Alternatives to OCR

Matching (see description on page 89)
As an alternative to classical OCR, matching can be used to read single characters or more complex symbols. In this case, one model for each character must be generated. The advantage of matching is the invariance to rotation. Furthermore, it is not necessary to segment the characters prior to the classification. Therefore, the matching approach should be considered when there is no robust way to separate the characters from the background.

Classification
You can consider the OCR tool as a convenient way of using a classifier. The OCR automatically derives invariant features and passes them to the underlying classifier. If the features offered by the OCR do not fulfill the needs of your application, you can create an "extended" OCR classifier by calculating the features using normal HALCON feature extraction operators and then using them with one of the classifiers that HALCON offers (see the chapters "Regions ▷ Features" and "Classification" in the Reference Manual).

18.5 Tips & Tricks

18.5.1 Composed Symbols

Some characters and symbols are composed of multiple sub-symbols, like an "i", "%", or "!". For the OCR, these sub-symbols must be combined into a single region. If you use the operators `segment_characters` and `select_characters` for the segmentation of the characters, sub-symbols are automatically combined. Otherwise, you can combine them by calling `closing_rectangle1` after thresholding, typically using a small width but a larger height. After calling `connection` to separate the characters, you use the operator `intersection` to get the original segmentation (input parameter 2), while preserving the correct connected components from `connection` (input parameter 1).

18.6 Advanced Topics

18.6.1 Line Scan Cameras

In general, line scan cameras are treated like normal area sensors. But in some cases, not single images but an "infinite" sequence of images showing objects, e.g., on a conveyor belt, must be processed. In this case, the end of one image is the beginning of the next one. This means that text or numbers, which partially lie in both images, must be combined into one object. For this purpose, HALCON provides the operator `merge_regions_line_scan`. This operator is called after the segmentation of one image, and combines the current objects with those of previous images. For more information see the Solution Guide II-A.

18.6.2 Circular Prints

In some cases the symbols are not printed as straight lines but along arcs, e.g., on a CD. To read these, the (virtual) center and radius of the corresponding circle are extracted. Using the operator `polar_trans_image_ext`, the image is then unwrapped. To project a region obtained in the unwrapped image back into the original image, you can use the operator `polar_trans_region_inv`.

18.6.3 OCR Features

HALCON offers many different features for the OCR. Most of these are for advanced use only. In most cases it is recommended to use the feature combination `'default'`. This combination is based on the gray values within the surrounding rectangle of the character. In case that the background of the characters cannot be used, e.g., if it varies because of texture, the features `'pixel_binary'`, `'ratio'`, and `'anisometry'` are good combinations. Here, only the region is used, the underlying gray values are ignored.

18.7 Pretrained OCR Fonts

The following sections shortly introduce you to the pretrained OCR fonts provided by HALCON. You can access them in the subdirectory `ocr` of the directory where you installed HALCON. Note that the pretrained fonts were trained with symbols that are printed dark on light. If you want to read light on dark symbols with one of the provided fonts, you can either invert the image with the operator `invert_image`, or, if this does not lead to a satisfying result, preprocess the image by applying first the operator `gen_image_proto` with a light gray value and then `overpaint_region` with the gray value set to `0`.

Note that the pretrained fonts were trained with the character encoding Windows-1252. Therefore, the appearance of character symbols with an ASCII code above 127 (these are '€', '£', '¥') may differ from the expected appearance depending on the character encoding of your system. In such cases, the classified characters should be checked based on their ASCII code, i.e. 128 for '€', 163 for '£', and 165 for '¥'.

The pretrained fonts are based on an MLP classifier, except for the `'Universal'` font, which is based on CNN.

18.7.1 Pretrained Fonts with Regularized Weights and Rejection Class

All pretrained OCR fonts are available in two versions. Font names ending with `_NoRej` have regularization weights but no rejection class, font names ending with `_Rej` have regularization weights as well as a rejection class. Because of the regularization, the pretrained OCR fonts provide more meaningful confidences. With fonts that provide a rejection class, it is possible to distinguish characters from background clutter. Fonts with a rejection class return the ASCII Code 26, which is SUB (substitute) if no letter was found.

18.7.2 Nomenclature for the Ready-to-Use OCR Fonts

There are several groups of OCR fonts available. The members of each group differ as they contain different symbol sets. The content of an OCR font is described by its name. For the names of the pretrained OCR fonts the following nomenclature is applied:

The name starts with the group name, e.g., `Document` or `DotPrint`, followed by indicators for the set of symbols contained in the OCR font. The meaning of the indicators is the following:

- `0-9`: The OCR font contains the digits `0` to `9`.
- `A-Z`: The OCR font contains the uppercase characters `A` to `Z`.
- `+` : The OCR font contains special characters. The list of special characters varies slightly over the individual OCR fonts. It is given below for each OCR font separately.
- `_NoRej`: The OCR font has no rejection class.
- `_Rej`: The OCR font has a rejection class.

If the name of the OCR font does not contain any of the above indicators or is only followed by the indicators `_NoRej` or `_Rej`, typically, the OCR font contains the digits `0` to `9`, the uppercase characters `A` to `Z`, the lowercase characters `a` to `z`, and special characters. Some of the OCR fonts do not contain lowercase characters (e.g., `DotPrint`). This is explicitly mentioned in the description of the respective OCR fonts.

18.7.3 Ready-to-Use OCR Font `'Document'`

The OCR font `Document` can be used to read characters printed in fonts like Arial, Courier, or Times New Roman. These are typical fonts for printing documents or letters.

Note that the characters `I` and `l` of the font Arial cannot be distinguished. That means that an `l` may be mistaken for an `I` and vice versa.

Available special characters: - = + < > . # $ % & ( ) @ * € £ ¥

The following OCR fonts with different symbol sets are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, a-z, special characters	`Document_Rej`	`Document_NoRej`
A-Z, special characters	`Document_A-Z+_Rej`	`Document_A-Z+_NoRej`
0-9	`Document_0-9_Rej`	`Document_0-9_NoRej`
0-9, A-Z	`Document_0-9A-Z_Rej`	`Document_0-9A-Z_NoRej`

18.7.4 Ready-to-Use OCR Font `'DotPrint'`

The OCR font `DotPrint` can be used to read characters printed with dot printers (see figure 18.13).

It contains no lowercase characters.

Available special characters: - / . * :

The following OCR fonts with different symbol sets are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, special characters	`DotPrint_Rej`	`DotPrint_NoRej`
A-Z, special characters	`DotPrint_A-Z+_Rej`	`DotPrint_A-Z+_NoRej`
0-9	`DotPrint_0-9_Rej`	`DotPrint_0-9_NoRej`
0-9, special characters	`DotPrint_0-9+_Rej`	`DotPrint_0-9+_NoRej`
0-9, A-Z	`DotPrint_0-9A-Z_Rej`	`DotPrint_0-9A-Z_NoRej`

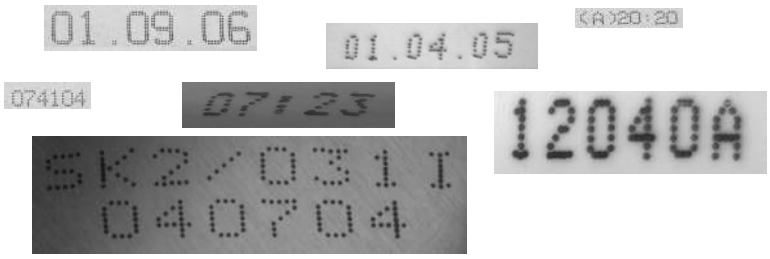


Figure 18.13: Examples for dot prints.

18.7.5 Ready-to-Use OCR Font `'HandWritten_0-9'`

The OCR font `HandWritten_0-9` can be used to read handwritten numbers (see figure 18.14).

It contains the digits 0-9.

Available special characters: none

The following OCR fonts are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9	`HandWritten_0-9_Rej`	`HandWritten_0-9_NoRej`

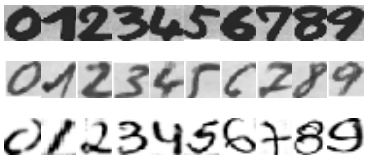


Figure 18.14: Examples for handwritten numbers.

18.7.6 Ready-to-Use OCR Font 'Industrial'

The OCR font `Industrial` can be used to read characters printed in fonts like Arial, OCR-B, or other sans-serif fonts (see figure 18.15). These fonts are typically used to print, e.g., labels.

Available special characters: - / + . $ % * € £ ¥

The following OCR fonts with different symbol sets are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, a-z, special characters	`Industrial_Rej`	`Industrial_NoRej`
A-Z, special characters	`Industrial_A-Z+_Rej`	`Industrial_A-Z+_NoRej`
0-9	`Industrial_0-9_Rej`	`Industrial_0-9_NoRej`
0-9, special characters	`Industrial_0-9+_Rej`	`Industrial_0-9+_NoRej`
0-9, A-Z	`Industrial_0-9A-Z_Rej`	`Industrial_0-9A-Z_NoRej`

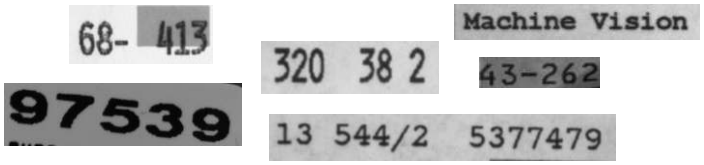


Figure 18.15: Examples for industrial prints.

18.7.7 Ready-to-Use OCR Font 'OCR-A'

The OCR font `OCR-A` can be used to read characters printed in the font OCR-A (see figure 18.16).

Available special characters: - ? ! / \{ } = + < > . # $ % & ( ) @ * € £ ¥

The following OCR fonts with different symbol sets are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, a-z, special characters	`OCRA_Rej`	`OCRA_NoRej`
A-Z, special characters	`OCRA_A-Z+_Rej`	`OCRA_A-Z+_NoRej`
0-9	`OCRA_0-9_Rej`	`OCRA_0-9_NoRej`
0-9, A-Z	`OCRA_0-9A-Z_Rej`	`OCRA_0-9A-Z_NoRej`

Figure 18.16: Selected characters of the OCR-A font.

18.7.8 Ready-to-Use OCR Font 'OCR-B'

The OCR font `OCR-B` can be used to read characters printed in the font OCR-B (see figure 18.17).

Available special characters: - ? ! / \{ } = + < > . # $ % & ( ) @ * € £ ¥

The following OCR fonts with different symbol sets are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, a-z, special characters	OCRB_Rej	OCRB_NoRej
A-Z, special characters	OCRB_A-Z+_Rej	OCRB_A-Z+_NoRej
0-9	OCRB_0-9_Rej	OCRB_0-9_NoRej
0-9, A-Z	OCRB_0-9A-Z_Rej	OCRB_0-9A-Z_NoRej
0-9, A-Z, + and <	OCRB_passport_Rej	OCRB_passport_NoRej

Figure 18.17: Selected characters of the OCR-B font.

18.7.9 Ready-to-Use OCR Font 'Pharma'

The OCR font `Pharma` can be used to read characters printed in fonts like Arial, OCR-B, and other fonts that are typically used in the pharmaceutical industry (see figure 18.18).

This OCR font contains no lowercase characters.

Available special characters: - / . ( ) :

The following OCR fonts with different symbol sets are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, special characters	`Pharma_Rej`	`Pharma_NoRej`
0-9	`Pharma_0-9_Rej`	`Pharma_0-9_NoRej`
0-9, special characters	`Pharma_0-9+_Rej`	`Pharma_0-9+_NoRej`
0-9, A-Z	`Pharma_0-9A-Z_Rej`	`Pharma_0-9A-Z_NoRej`

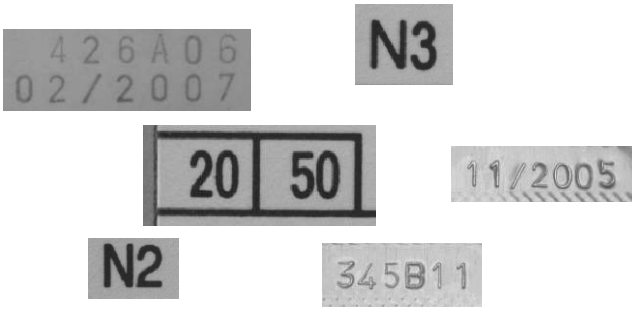


Figure 18.18: Examples for pharmaceutic labels.

18.7.10 Ready-to-Use OCR Font 'SEMI'

The OCR font SEMI can be used to read characters printed in the SEMI font which consists of characters which are designed to be easily distinguished from each other. It has a limited set of characters which can be viewed in figure 18.19.

This OCR font contains no lowercase characters.

Available special characters: - .

The following OCR fonts are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, special characters	SEMI_Rej	SEMI_NoRej

Figure 18.19: Character set of SEMI font.

Figure 18.20: Examples for SEMI font. Note that these images were inverted before the training was applied! That is, this font is nevertheless pretrained only for dark on light symbols.

18.7.11 Ready-to-Use OCR Font 'Universal'

The OCR font Universal can be used to read a wide range of different characters (see figure 18.21). The training of this CNN-based font is based on the characters used for 'Document', 'DotPrint', 'SEMI', and 'Industrial'.

Available special characters: - / = + : < > . # $ % & ( ) @ * € £ ¥

The following OCR fonts are available:

Symbol set	Name of OCR font with rejection class	Name of OCR font without rejection class
0-9, A-Z, a-z, special characters	Universal_Rej	Universal_NoRej
A-Z, special characters	Universal_A-Z+_Rej	Universal_A-Z+_NoRej
0-9	Universal_0-9_Rej	Universal_0-9_NoRej
0-9, special characters	Universal_0-9+_Rej	Universal_0-9+_NoRej
0-9, A-Z	Universal_0-9A-Z_Rej	Universal_0-9A-Z_NoRej
0-9, A-Z, special characters	Universal_0-9A-Z+_Rej	Universal_0-9A-Z+_NoRej

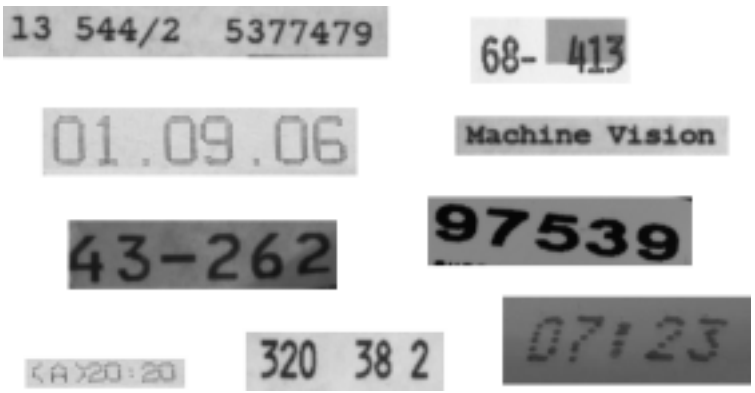


Figure 18.21: Examples for Universal font.

Chapter 19

OCR (Deep OCR)

This chapter describes Optical Character Recognition (OCR) using Deep OCR, a deep-learning-based method designed for this task. The approach differs from the ones introduced in Chapter OCR on page 183 as not only single characters are read but connected characters, to which we will refer as words.

19.1 Basic Concept

Deep OCR is a deep-learning-based method with its own operator set allowing a customized interface. Its models usually consist of two components:

`detection_model`: Detects words in the image.

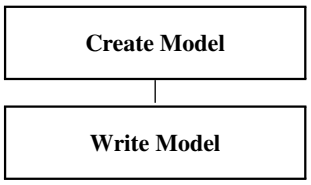
`recognition_model`: Recognizes the word in the detected image part.

This said, the model can detect the words in the image (without the need of determining special regions of interest) and recognize them. Thereby, the words may even be rotated, see e.g., figure 19.1. The provided model can only detect words and recognize characters from the set it has learned and only read fonts similar to the ones it has seen during training. If this model is not sufficient for your specific task, both components of the model can be retrained with custom data (see section 19.2).
For more information on the general concept of deep learning, see "Deep Learning".

It is also possible to create a model consisting only of one of the two components mentioned above. In this case, the model is smaller and faster, but accomplishes only a single task.

19.1.1 Offline Phase

In case the provided Deep OCR model is used as it is, the offline part consists only of the model creation and writing it in a file.



19.1.1.1 Create Model

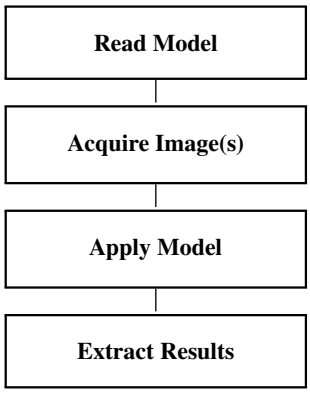
The Deep OCR model needs to be created first. For this the operator `create_deep_ocr` has to be used.

19.1.1.2 Write Model

Write the model in a file using `write_deep_ocr`. This allows to read the model in the online phase instead of creating a new one every time.

19.1.2 Online Phase

The online part consists of reading the images and reading the words on it. In the following its basic steps are listed.



19.1.2.1 Read Model

As a first step, the Deep OCR model you created prior to this step is read using `read_deep_ocr`. Depending on your setting and application you may want to set specific parameters using `set_deep_ocr_param`

19.1.2.2 Acquire Image(s)

Images must be acquired.

For detailed information see the description of this method on page 21.

Depending on the model and the image, preprocessing might be necessary.

19.1.2.3 Apply Model

Apply the Deep OCR model on the read image using `apply_deep_ocr`.

19.1.2.4 Extract Results

Extract the results from the returned dictionary. Possible entries are described in the reference manual entry of `apply_deep_ocr`.

19.2 Retrain Model (Recognition & Detection Component)

The provided model can be retrained in order to adjust it to a specific task. This is done by providing task-specific data for training and evaluation of the model. The requirements and workflow are described in the chapter reference "OCR ▷ Deep OCR".

For a successful training the training data needs to cover the full spectrum of words and character range that might occur in the actual application. After training and evaluation the retrained model can be used instead of the pretrained model in the Deep OCR workflow as described in Basic Concept on page 209.

For each model component an example program using this approach is described in section 19.3.2 (recognition) and section 19.3.3 on page 212 (detection), respectively.

19.3 Programming Examples

This section shows examples using Deep OCR in HALCON.

19.3.1 Locate and Recognize Text

Example: `%HALCONEXAMPLES%/hdevelop/OCR/Deep-OCR/deep_ocr_workflow.hdev`

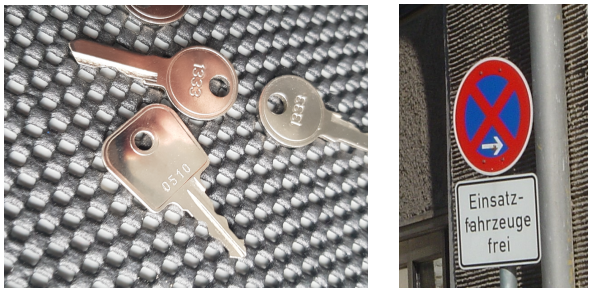


Figure 19.1: Different images with words to be detected and read.

Figure 19.1 shows images with connected characters, thus words. The task is to detect the words and read them within one setup, although the input images are differing significantly. Note how the words on the left image are not all horizontally aligned.

The example program solves the task closely to the shown Basic Concept on page 209. There are two exceptions: As a first, there is no model to be read, so it has to be created. As a second, results are visualized as shown in figure 19.2.

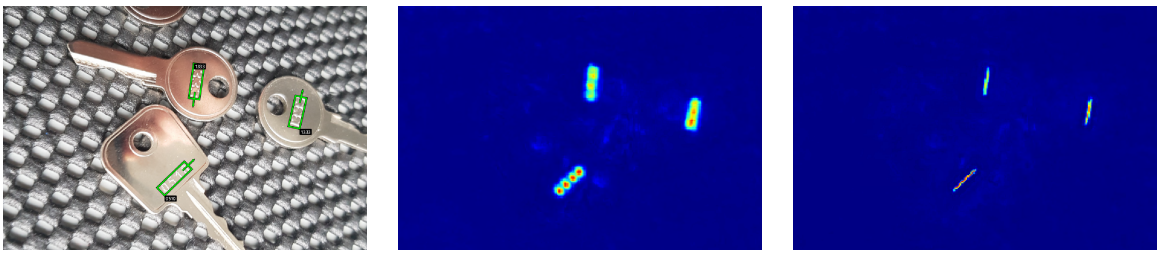


Figure 19.2: Left: The image shows three keys with punched numbers of several digits. These numbers were found and recognized.
Middle: Visualized scores for the detected character centers.
Right: Visualized scores for the connection of detected character centers.

The example continues and demonstrates also how a single component can be applied and how to deal with images too large for the detection component (see section 19.4).

19.3.2 Retrain Recognition Component

Example:
`%HALCONEXAMPLES%/hdevelop/OCR/Deep-OCR/deep_ocr_recognition_training_workflow.hdev`

This example shows how retraining the Deep OCR recognition component can improve your results.

The text in the example images contains characters ('{' and '}') and a font that the pretrained Deep OCR model has not seen during its training. Therefore, a retraining with custom data that contains the font and the characters that will occur during inference is performed. The result of such training is shown in figure 19.3.

Figure 19.3: The retrained model learned to identify the font and characters (bottom) it could not handle correctly before (top).

19.3.3 Retrain Detection Component

Example:
`%HALCONEXAMPLES%/hdevelop/OCR/Deep-OCR/deep_ocr_detection_training_workflow.hdev`

This example shows how retraining the Deep OCR detection component can improve your results.

The numbers in the example images contain contrast and reflections that the pretrained Deep OCR model has not seen during its training. Therefore, a retraining with custom data that contains the contrast and reflections that will occur during inference is performed. The result of such training is shown in figure 19.4.

Figure 19.4: The retrained model improved the localization of numbers (right) compared to the pretrained model (left).

19.4 Large images

If the input `Image` is very large, zooming it to the model input dimensions can lead to degraded detection performance. This is because the text in the model input will become very small. In these cases an automatic tiling helps. Thus, setting *'detection_tiling'* using `set_deep_ocr_param` internally the input image is tiled, meaning split into smaller, overlapping parts. These parts are processed separately and the results pieced together. An illustration of such a tiling is shown in figure 19.5. As these tiles are much smaller than the original image, the text in them stays readable for the model.

This is also shown in the example: %HALCONEXAMPLES%/hdevelop/OCR/Deep-OCR/deep_ocr_workflow.hdev

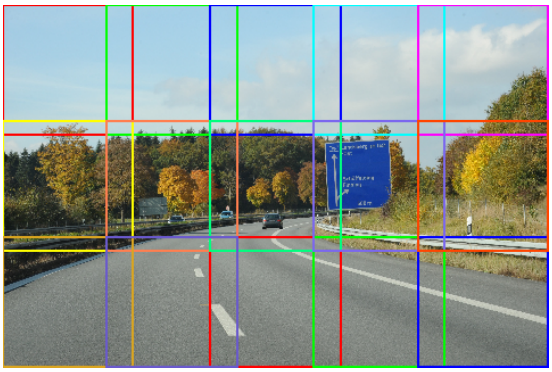


Figure 19.5: Example of a tiled input image.

19.5 Relation to Other Methods

The approach differs from the ones introduced in Chapter OCR on page 183 in various ways. As a first, the model does not try to recognize only single characters but connected characters, thus words. There is not a specific, single font for a Deep OCR model. The model has been trained with a variety of font types and which it can recognize. Further the model is able to detect words of different size and orientation.
But the drawback is that the model can not be enhanced by training an additional character or font type yet. If a specific character has not been learned often enough during training, the model will most probably fail to recognize it. Words written in a font the model has not learned will most probably not be read correctly, some times not even detected.

Chapter 20

Stereo Vision

The basic principle of stereo vision is that 3D coordinates of object points are determined from two or more images that are acquired simultaneously from different points of view. HALCON provides two stereo methods: binocular stereo and multi-view stereo.

Binocular stereo uses exactly two cameras and returns disparity images, distance images, or 3D coordinates. Figure 20.1, e.g., shows a stereo image pair of a board and the resulting height map of the board's components.

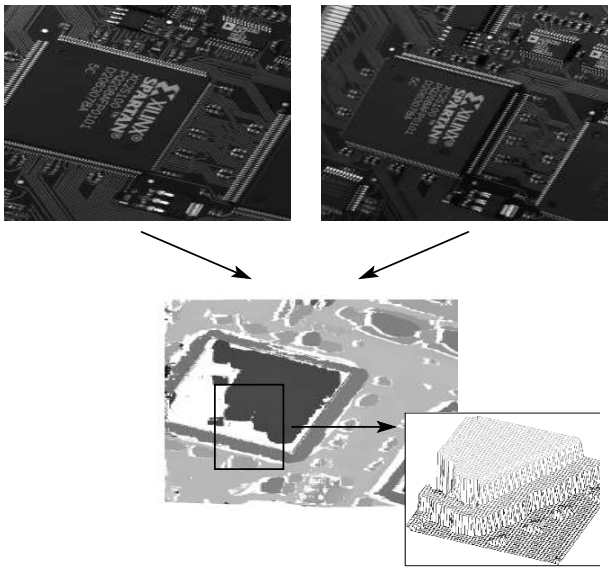


Figure 20.1: Basic principle of binocular stereo. Top: stereo image pair; Bottom: height map.

Multi-view stereo can also use more than two cameras. It can be used to either reconstruct surfaces that are returned as 3D object models or to reconstruct individual points. Figure 20.2, e.g., shows the images of a multi-view stereo camera system that is used to reconstruct the surface of pipe joints and the resulting 3D object model.

The advantage of stereo vision is that 3D information of the surface of arbitrarily shaped objects can be determined from images. Stereo vision can also be combined with other vision methods, e.g., as a preprocessing step for blob analysis, which can be used to extract specific objects or object parts from the depth image, or surface-based 3D matching, which locates objects that are provided as 3D models in the reconstructed surface.

For detailed information about stereo vision, please refer to the Solution Guide III-C, chapter 5 on page 117.

20.1 Basic Concept

The derivation of 3D information with a stereo camera system consists of four main steps:

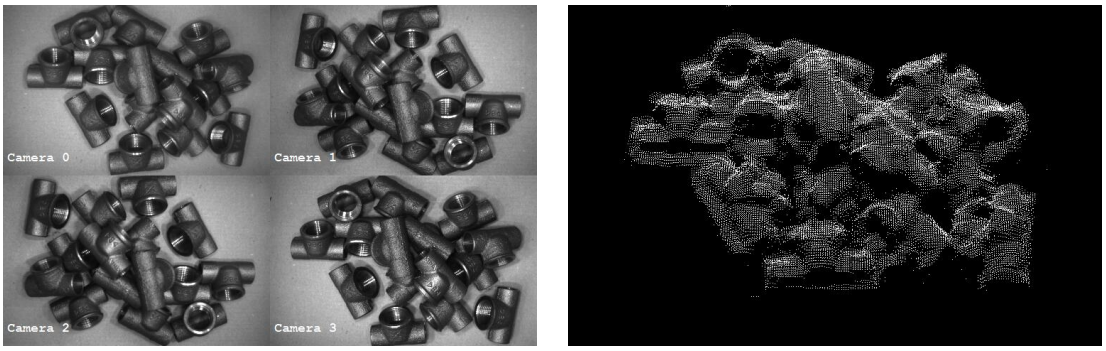
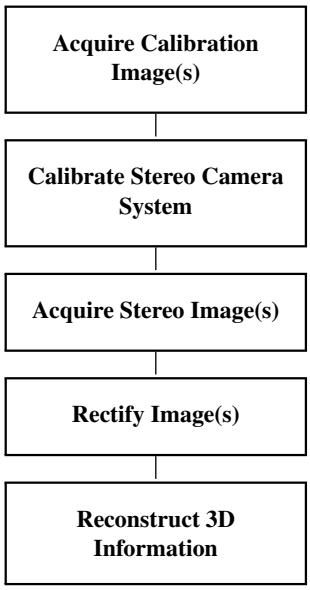


Figure 20.2: Left: images of a multi-view stereo camera system; right: reconstructed surface of pipe joints.



20.1.1 Acquire Calibration Image(s)

A number of stereo calibration images is acquired. Each image shows the HALCON calibration plate in a different position and orientation.

20.1.2 Calibrate Stereo Camera System

Using the previously acquired calibration images, the stereo camera system is calibrated. For this, the image coordinates of the calibration marks must be extracted from the calibration images. Then, the parameters of the stereo setup are determined.

For **binocular stereo**, the parameters are then used to create the rectification maps for the rectification of the stereo images.

For **multi-view stereo**, a so-called stereo model is created, which contains the camera parameters and further information.

The calibration process is described in detail in the Solution Guide III-C in section 5.2 on page 122.

20.1.3 Acquire Stereo Image(s)

Stereo images are simultaneously acquired with the calibrated stereo camera system. They show the object for which the 3D information should be reconstructed.

20.1.4 Rectify Image(s)

For **binocular stereo**, the stereo images must be rectified such that corresponding points (conjugate points) lie on the same row in both rectified images. For this, the rectification map that has been determined above must be used.

20.1.5 Reconstruct 3D Information

Now, 3D information can be reconstructed.

With **binocular stereo**, for each point of the first rectified image the conjugate point in the second rectified image is determined (stereo matching). For these points, either the disparity or the distance to the stereo camera system can be calculated and returned as a gray value image. The reference plane to which these distances are related can be changed (see the Solution Guide III-C, section 5.3.5.4 on page 134). It is also possible to derive the distance or the 3D coordinates for selected points for which the disparity is known. What is more, 3D coordinates can be determined from the image coordinates of each pair of conjugate points directly. See the Solution Guide III-C, section 5.3.5 on page 130, for details.

With **multi-view stereo**, you can reconstruct complete surfaces or 3D coordinates for selected points. When reconstructing surfaces, you first set parameters for the reconstruction with `set_stereo_model_param`, then you apply `set_stereo_model_image_pairs` to define which cameras build pairs, and finally you reconstruct the surface with `reconstruct_surface_stereo`. When reconstructing selected points of an object, you first extract the corresponding points from the stereo images, then you accumulate the correspondence information for all cameras that image the object, and finally you reconstruct the points with `reconstruct_points_stereo`. See the Solution Guide III-C, section 5.4.2 on page 141, for details.

20.2 Extended Concept

In many cases, the derivation of 3D information with a binocular stereo system involves more steps than described above. Reasons for this are, e.g., the need to restrict the stereo reconstruction to an ROI. Furthermore, postprocessing like the transformation of the 3D coordinates into another coordinate system or the visualization of results is often required.

20.2.1 Use Region Of Interest

A region of interest can be created to reduce the image domain for which the stereo matching will be performed. This will reduce the processing time.

For detailed information see the description of this method on page 25.

20.2.2 Transform Results Into World Coordinates

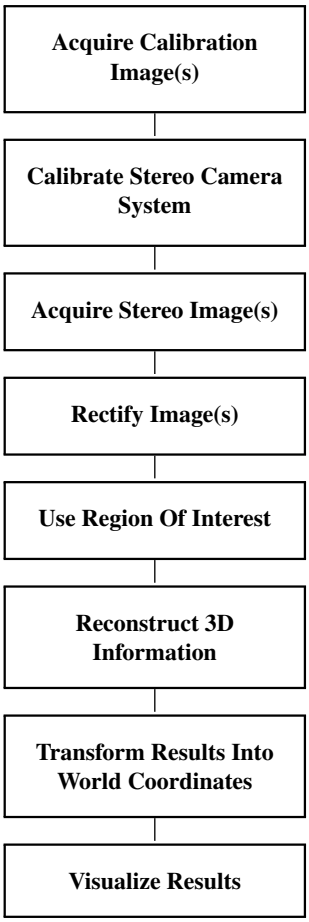
In some applications, the 3D coordinates must be transformed into a given world coordinate system.

For **binocular stereo**, this step is performed after the reconstruction: Beforehand, the relation between the given world coordinate system and the stereo camera system must be determined. Then, the 3D coordinates can be transformed as requested. How to transform results into world coordinates is described in detail in the Solution Guide III-C in section 3.3 on page 76.

For **multi-view stereo**, the world coordinate system can be specified already when creating the stereo model (see the Solution Guide III-C, section 5.4.1.2 on page 140 for details).

20.2.3 Visualize Results

Finally, you might want to visualize the disparity or distance images or the reconstructed surface (3D object model). We recommend to have a look at the provided examples.



20.3 Programming Examples

This section gives a brief introduction to using HALCON for stereo vision.

20.3.1 Segment the Components of a Board With Binocular Stereo

Example: `%HALCONEXAMPLES%/hdevelop/Applications/Object-Recognition-2D/board_components.hdev`

Figure 20.3 shows a stereo image pair of a board together with the result of a segmentation of raised objects. The segmentation has been carried out based on the distance image that was derived with binocular stereo.

First, a calibration data model is created and initialized:

```
create_calib_data ('calibration_object', 2, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 'all', [], StartCamPar)
set_calib_data_calib_object (CalibDataID, 0, CalDescrFile)
```

Then, a number of calibration images must be acquired. Here, a calibration plate with rectangularly arranged marks is used. A subset of these calibration images is shown in figure 20.4. For details regarding the calibration process, including how to take a suitable set of calibration images, refer to the chapter reference "Calibration".

In the individual calibration images, `find_calib_object` locates the calibration plate, extracts the image coordinates of the calibration marks, and stores them in the calibration data model.

```
for I := 1 to Number by 1
    find_calib_object (ImageL, CalibDataID, 0, 0, I, [], [])
    find_calib_object (ImageR, CalibDataID, 1, 0, I, [], [])
endfor
```

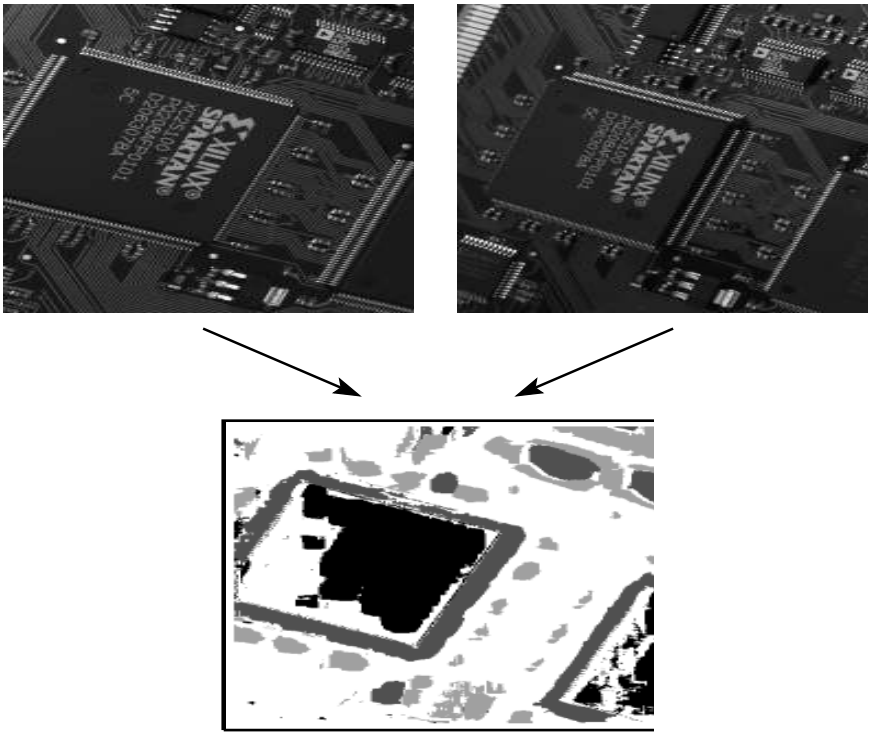


Figure 20.3: Segment board components based on their height.

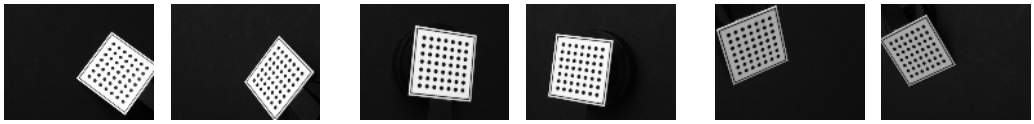


Figure 20.4: A subset of the calibration images that are used for the calibration of the stereo camera system.

With this, the actual calibration of the stereo camera system is performed and the results are accessed.

```
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParamL)
get_calib_data (CalibDataID, 'camera', 1, 'params', CamParamR)
get_calib_data (CalibDataID, 'camera', 1, 'pose', cLPcR)
```

Now, the rectification maps for the rectification of the stereo image pair can be generated.

```
gen_binocular_rectification_map (MapL, MapR, CamParamL, CamParamR, cLPcR, 1, \
                                 'viewing_direction', 'bilinear', \
                                 RectCamParL, RectCamParR, CamPoseRectL, \
                                 CamPoseRectR, RectLPosRectR)
```

Then, each stereo image pair acquired with the calibrated stereo camera system can be rectified. This has the effect that conjugate points have the same row coordinate in both images. The rectified images are displayed in figure 20.5

```
map_image (ImageL, MapL, ImageRectifiedL)
map_image (ImageR, MapR, ImageRectifiedR)
```

From the rectified images, a distance image can be derived in which the gray values represent the distance of the respective object point to the stereo camera system. This step is the core of the stereo approach. Here, the stereo matching, i.e., the determination of the conjugate points takes place.

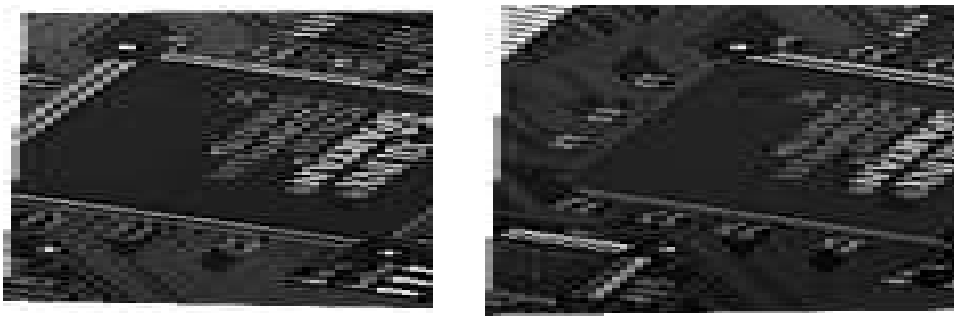


Figure 20.5: Rectified images.

```
binocular_distance (ImageRectifiedL, ImageRectifiedR, DistanceImage, \
                    ScoreImageDistance, RectCamParL, RectCamParR, \
                    RectLPosRectR, 'ncc', MaskWidth, MaskHeight, \
                    TextureThresh, MinDisparity, MaxDisparity, NumLevels, \
                    ScoreThresh, 'left_right_check', 'interpolation')
```

Finally, the distance image can, e.g., be corrected such that a given object plane receives a specified distance value, e.g., zero. Objects that deviate from the given object plane can thus be segmented very easily with a threshold operation.

```
threshold (HeightAboveReferencePlaneReduced, Range1, Height1_Min, \
           Height1_Max)
threshold (HeightAboveReferencePlaneReduced, Range2, Height2_Min, \
           Height2_Max)
threshold (HeightAboveReferencePlaneReduced, Range3, Height3_Min, \
           Height3_Max)
```

20.3.2 Reconstruct the Surface of Pipe Joints With Multi-View Stereo

Example: %HALCONEXAMPLES%/hdevelop/Applications/Robot-Vision/locate_pipe_joints_stereo.hdev

Figure 20.2 shows images from a 4-camera system. Using multi-view stereo, the surface of the pipe joints is reconstructed.

Here, the calibration is performed in an offline step, the known camera parameters are used.

```
init_camera_setup (CameraSetupModelID)
```

Using the camera setup model, a stereo model for surface reconstruction is created.

```
create_stereo_model (CameraSetupModelID, 'surface_pairwise', [], [], \
                     StereoModelID)
```

Now, the image pairs are specified. First, some parameters for the rectification maps are set. Then, it is specified between which cameras the disparity is computed.

```
set_stereo_model_param (StereoModelID, 'rectif_interpolation', 'bilinear')
set_stereo_model_param (StereoModelID, 'rectif_sub_sampling', 1.2)
set_stereo_model_image_pairs (StereoModelID, [0, 2], [1, 3])
```

Furthermore, the stereo model is configured by setting several parameters. First, the bounding box is set that restricts the reconstruction to a specific part of the 3D space. Additionally, the internal call of binocular_disparity is adjusted.

```
set_stereo_model_param (StereoModelID, 'bounding_box', [-0.2, -0.07, -0.075, \
                        0.2, 0.07, -0.004])
set_stereo_model_param (StereoModelID, 'sub_sampling_step', 3)
set_stereo_model_param (StereoModelID, 'binocular_filter', \
                        'left_right_check')
```

Finally, the reconstruction operator is called. It returns the reconstructed surface in form of a 3D object model.

```
reconstruct_surface_stereo (Images, StereoModelID, PipeJointPileOM3DID)
```

In section 11.3.2 on page 108, the reconstructed surface is then used as input for surface-based 3D matching, which recognizes and locates individual pipe joints.

20.4 Relation to Other Methods

20.4.1 Methods that are Using Stereo Vision

Blob Analysis (see description on page 33)
The results of **binocular stereo** can be used as input for blob analysis. This may be useful if locally high structures must be extracted that cannot be reliably detected from the original image. By applying blob analysis to the distance image, such structures may be extracted very easily.

3D Matching (Surface-Based) (see description on page 101)
The 3D object model that is returned by a surface reconstruction with **multi-view stereo** can be used as input for surface-based 3D matching.

3D Primitives Fitting (see Solution Guide III-C, section 4.5 on page 111)
The 3D object model that is returned by a surface reconstruction with **multi-view stereo** can be used as input for 3D primitives fitting if the contained surface is meshed. To get a meshed surface, the parameter `'point_meshing'` must have been set for the stereo model with `set_stereo_model_param` before reconstructing the surface.

20.5 Tips & Tricks

20.5.1 Speed Up

Many online applications require maximum speed. Although the stereo matching is a very complex task, you can speed up this process by using regions of interest, which are the standard method to increase the speed by processing only those areas where objects must be inspected. This can be done by using pre-defined regions but also by an online generation of the region of interest that depends on other objects found in the image.

Additionally, for the surface extraction with multi-view stereo, a significant speed up can be obtained when selecting the bounding box that restricts the reconstruction to a part of the 3D space as small as possible. The bounding box must be set with `set_stereo_model_param` before applying the reconstruction.

20.6 Advanced Topics

20.6.1 High Accuracy

Sometimes very high accuracy is required. To achieve a high distance resolution, i.e., a high accuracy with which the distance of the object surface from the stereo camera system can be determined, special care should be taken of the configuration of the stereo camera setup. The setup should be chosen such that the distance between the cameras as well as the focal length are large and the stereo camera system is placed as close as possible to the object. For more information, please refer to the Solution Guide III-C, section 5.1.2 on page 120.

Chapter 21

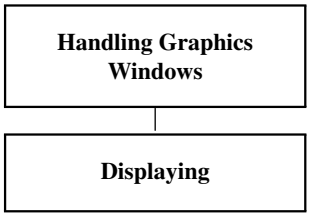
Visualization

Displaying data in HALCON is quite easy: In the graphics windows provided by HALCON, all supported data types can be visualized directly using specific display operators. Both the creation of these windows and the displaying requires only little programming effort because the functionality is optimized for the use in machine vision.

Making use of the HALCON visualization provides several advantages. First of all, it saves a lot of time during the development because all important visualization methods are already predefined. Furthermore, the functionality is independent of the operating system: Writing a program under Windows using the visualization of HALCON can be ported easily to Linux because the visualization operators behave identically and therefore only user code making use of operating system functions needs to be rewritten.

21.1 Basic Concept

For visualization there are two important aspects to consider: The graphics windows and the data that must be visualized.



21.1.1 Handling Graphics Windows

HALCON provides an easy-to-use operator to create a graphics window for visualization: `open_window`. This operator takes all necessary parameters to specify the dimensions, the mode, and the relation to a potential parent window. As a result, a `WindowHandle` is returned with which you refer to the window when displaying into it or when visualization parameters are changed. Note that the dimension of the window is not limited by the size of the virtual display. Thus, you can work also on systems with multiple screens. The most important operators for controlling a window are `clear_window` to reset it to its background color, `set_part` to specify the display coordinate system and `close_window` when the window is no longer needed. The operator `set_window_param` allows to set different parameters of an open window.

21.1.2 Displaying

For each HALCON data type, specific operators for displaying are provided. The most convenient operator for iconic data (images, regions, and XLD) is `disp_obj`. This operator automatically handles gray or color images,

regions, and XLDs. To control the way how data is presented in the window, operators with the prefix `set_` (or `dev_set_` for the visualization in HDevelop) are used. They allow to control the color, the draw mode, the line width, and many other parameters.

21.1.3 A First Example

An example for this basic concept is the following program, which shows how to visualize an image overlaid with a segmentation result. Here, the visualization operators provided in HDevelop are used.

Figure 21.1: Visualizing the segmented clips.

After reading the image from file the dark clips are selected with `binary_threshold`, which automatically selects the threshold value. After determining the connected components and selecting regions with the appropriate size, the result is visualized. First, the image is displayed. After this, the parameters for regions are set to multi-colors (12) and margin mode. Finally, the regions are displayed with these settings.

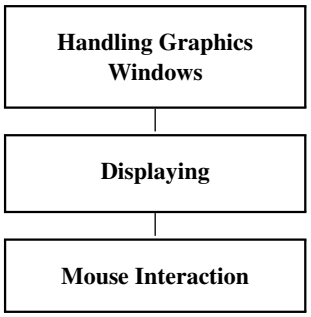
```
read_image (Image, 'clip')
binary_threshold (Image, Dark, 'max_separability', 'dark', UsedThreshold)
connection (Dark, Single)
select_shape (Single, Selected, 'area', 'and', 5000, 10000)
dev_display (Image)
dev_set_colored (12)
dev_set_draw ('margin')
dev_display (Selected)
```

21.2 Extended Concept

In advanced applications it is required to gain complete control over the visualization process. This can include using graphics windows in programs developed with Microsoft Visual Basic or Microsoft C++, changing the behavior of the graphics windows, making use of buffered output, or even using external programs for the visualization. HALCON provides full control over all these topics to provide advanced flexibility, in addition to the ease of use.

21.2.1 Handling Graphics Windows

In this section we consider the concept of graphics windows in more detail.



- The graphics windows are designed such that each one stores all the corresponding parameters in a kind of graphics context. Whenever an operator like `set_draw` is called, this context is modified to be used for this window until the value is overwritten. All operators that modify the graphics context start with the prefix `set_`. The current value of the context can be requested using the corresponding operator with the prefix `get_`, e.g., `get_draw`.
- Besides display parameters, each graphics window has a coordinate system that can be defined with `set_part`. The upper left corner of an image is (0,0), the lower right corner is (height-1,width-1). The size of an image can be requested using `get_image_size`. Please note that unlike in HDevelop this coordinate system must be specified by calling `set_part`. Otherwise, the part of the image that is visualized is undefined.
- The operator `open_window` has a parameter called `FatherWindow`. By default, the value 0 is used, which means that the window has no parent and floats as an independent instance. You can construct hierarchies of windows by passing the handle of one window as parent to the other. Besides this, it is also possible to use the parent mechanism to embed the graphics windows into other forms (see section 21.5 on page 230). To set different parameters of an open window, use `set_window_param`.

21.2.2 Displaying

After having opened a graphics window, the returned window handle is used to communicate with it. Typically, first the visualization parameters are specified before data is displayed. To control images only few operators are needed: `set_paint` (for profiles and 3D plots), `set_lut` (for look-up-tables), and `set_part_style` (for the quality of the zooming interpolation). To specify the output for regions, many parameters are available. The most important ones are `set_draw` (for filled or margin mode), `set_color` (for the pen color), `set_line_width` (for the pen width), and `set_colored` (for multi-color modes). To display XLD data, the same parameters (except `set_draw`) are used.

With the display mode '3d_plot' you can create an interactive display of a height field in 3D. The operator `update_window_pose` allows to manipulate the pose of such a 3D plot in an intuitive way and the operator `unproject_coordinates` calculates the image coordinates for a point in a 3D plot window. The example `%HALCONEXAMPLES%/hdevelop/Graphics/Parameters/set_paint_3d_plot.hdev` shows how to use these operators. Note that when working with HDevelop, you can also switch directly to the interactive display of the height field in the 3d_plot mode using the button that is described in the HDevelop User's Guide (see section 6.8.4 on page 80).

The visualization itself is performed with operators like `disp_image`, `disp_color`, `disp_region`, or `disp_xld`. The most convenient way is to use `disp_obj`, which automatically uses the correct method.

For text output, you first specify the font with `set_font`. The desired position in the graphics window is determined with `set_tposition`. Writing text into the window is performed with `write_string`

For a flicker-free display see section 21.5 on page 230.

21.2.3 Mouse Interaction

The most important thing to know is that HALCON does *not* use an event-driven approach for mouse handling. Each operator is designed such that the mouse interaction starts when the operator is called and finishes when

the operator returns. If an event-driven mode is needed one has to use the standard mechanisms provided by the operating system.

Interacting with the mouse mainly involves two tasks:

- The first task is to request the position of the mouse. This can be achieved using get_mposition if a pixel-precise result is sufficient, or using get_mposition_sub_pix for a subpixel-precise output. These operators return immediately and have the position and the mouse button as result. An alternative are get_mbutton and get_mbutton_sub_pix for subpixel-precise positions. These operators only return when a mouse button has been clicked.
- The second important action is drawing shapes with the mouse. This is done with special operators whose names start with draw_. Operators for many different shapes (like circles or rectangles) are provided. Furthermore, different data types like regions or XLD contours can be used for the result.

21.3 Programming Examples

This section gives a brief introduction to using the visualization operators provided by HALCON.

21.3.1 Displaying HALCON data structures

Example: %HALCONEXAMPLES%/solution_guide/basics/display_operators.hdev

The example program is designed to show the major features of displaying images, regions, XLD, and text. It consists of a small main program that calls procedures handling the four different data types. The program is written for HDevelop and uses its specific display operators. This is done in a way that naturally ports (e.g., with the automatic export) to other language interfaces like C++, C#, or Visual Basic.

The main procedure contains five procedures: open_graphics_window, display_image, display_regions, display_xld, and display_text. To switch between the programs of the individual procedures you can use the combo box Procedures in the program window. Following, selected parts of each procedure are explained.

```
read_image (Image, 'fabrik')
open_graphics_window (Image, WindowHandle)
display_image (Image, WindowHandle)
regiongrowing (Image, Regions, 1, 1, 3, 100)
display_regions (Image, Regions, WindowHandle)
edges_sub_pix (Image, Edges, 'lanser2', 0.5, 10, 30)
display_xld (Image, Edges, WindowHandle)
display_text (Image, Regions, WindowHandle)
```

open_graphics_window is a support procedure to open a graphics window that has the same size as the image. This is done by calling get_image_size to access the image dimensions. Before opening the new window, the existing window is closed. To adapt the coordinate system accordingly, dev_set_part is called. This would be done automatically in HDevelop, but for the other programming environments this step is necessary. Finally, the default behavior of HDevelop of displaying each result automatically is switched off. This has the effect that only programmed output will be visible.

```
get_image_size (Image, Width, Height)
dev_close_window ()
dev_open_window (0, 0, Width, Height, 'white', WindowHandle)
dev_set_part (0, 0, Height - 1, Width - 1)
dev_update_window ('off')
```

Then, the display procedure for images is called: display_image. It has the Image and the WindowHandle as input parameters. First, the window is activated, which again is not needed for HDevelop, but is important for other programming environments. Now, the image is displayed in the graphics window.

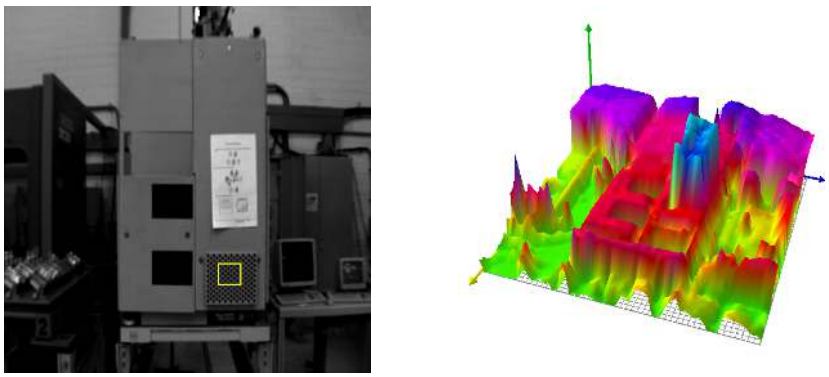
```
dev_set_window (WindowHandle)
dev_display (Image)
```

To change the look-up-table (LUT), dev_set_lut is called and the effect will become visible after calling dev_display once again.

```
dev_set_lut ('temperature')
dev_display (Image)
```

Next, a part of the image is displayed using a so-called 3D plot (see figure 21.2). Here, the gray values are treated as height information. For this mode another LUT is used.

```
gen_rectangle1 (Rectangle, 358, 298, 387, 329)
dev_set_draw ('margin')
dev_set_color ('yellow')
dev_display (Rectangle)
dev_set_part (358, 298, 387, 329)
dev_set_lut ('twenty_four')
dev_set_paint ('3d_plot')
dev_display (Image)
```



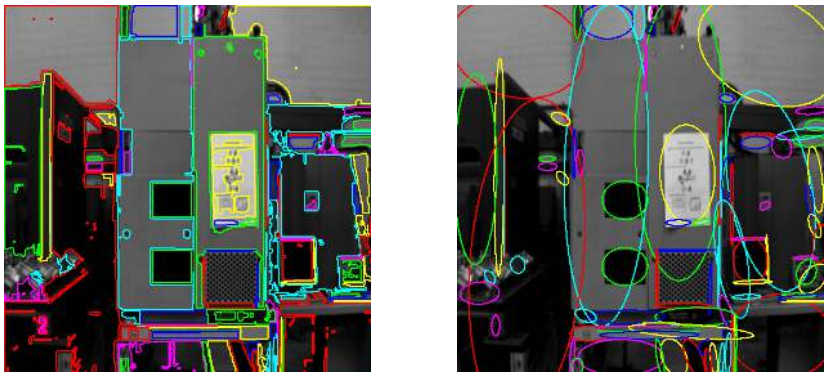
a) b)

Figure 21.2: (a) Original image with ROI; (b) 3D plot of image within the ROI.

The procedure to display regions is called display_regions. It first displays the image as background and then sets the display parameters for the regions. dev_set_draw specifies that only the region border is visualized. dev_set_colored activates the multi-color mode, where each region is displayed with different colors (which change cyclically). As an alternative to simply showing the original shape of the regions, HALCON enables you to modify the shape using dev_set_shape. In the given example the equivalent ellipses are chosen. The result is depicted in figure 21.3.

```
dev_display (Image)
dev_set_draw ('margin')
dev_set_colored (6)
dev_display (Regions)
dev_display (Image)
dev_set_shape ('ellipse')
dev_display (Regions)
```

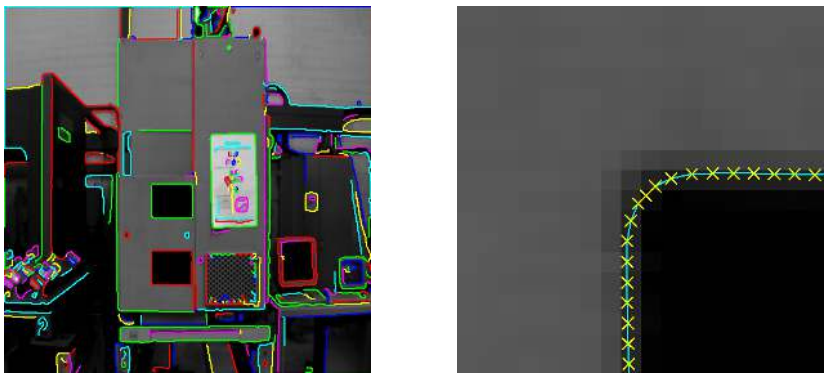
The procedure display_xld first shows all contours overlaid on the image using the multi-color mode. Then, a zoom is defined using dev_set_part. This zoom mode allows to inspect the subpixel accurate contours easily. To give additional information, contours with a given size are selected and for each of these the control points are extracted using get_contour_xld. The coordinates are here returned as tuples of real values. For each of these control points, a cross is generated using gen_cross_contour_xld, which is then overlaid onto the contour.



a) b)

Figure 21.3: (a) Regions and (b) their equivalent ellipses.

```
dev_display (Contours)
gen_rectangle1 (Rectangle, 239, 197, 239 + 17, 197 + 17)
dev_set_part (239, 197, 239 + 17, 197 + 17)
select_shape_xld (Contours, SelectedXLD, 'area', 'and', 2000, 3000)
count_obj (SelectedXLD, Number)
for k := 1 to Number by 1
    select_obj (SelectedXLD, SingleContour, k)
    get_contour_xld (SingleContour, Row, Col)
    for i := 0 to |Row| - 1 by 1
        gen_cross_contour_xld (Cross, Row[i], Col[i], 0.8, rad(45))
        dev_display (Cross)
    endfor
endfor
```



a) b)

Figure 21.4: Subpixel accurate contour control points: (a) original image; (b) zoomed image part.

The last part of the program is a procedure called `display_text`. It shows how to handle the mouse and text output. The task is to click with the mouse into the graphics window to select a specific region and to calculate features, which are then displayed in the graphics window. First, the font is selected. This is the only part of the visualization that is not portable between Linux and Windows because of the incompatible font name handling. Therefore, the convenience procedure `set_display_font` is called to hide the internals and unify the font selection. This procedure is part of the standard procedure path of HDevelop, and can be called from any program.

```
set_display_font (WindowHandle, 16, 'mono', 'true', 'false')
```

The rest of the program consists of a `while`-loop, which terminates as soon as the right mouse button is pressed. The mouse operator `get_mbutton` waits until the user clicks with the mouse into the graphics window and then

returns the coordinate and the button value. This coordinate is used to select the region that contains this point using select_region_point. For this region, the size and the center of gravity are calculated with area_center. First, the text cursor is positioned with set_tposition and the values are displayed using write_string. Here, it can be seen how conveniently strings can be composed using the "+" operator.

```
Button := 0
while (Button != 4)
    get_mbutton (WindowHandle, Row, Column, Button)
    select_region_point (Regions, DestRegions, Row, Column)
    area_center (DestRegions, Area, RowCenter, ColumnCenter)
    if (|Area| > 0)
        set_tposition (WindowHandle, Row, Column)
        dev_set_color ('yellow')
        write_string (WindowHandle, '(' + RowCenter + ', \
                    ' + ColumnCenter + ') = ' + Area)
    endif
endwhile
```

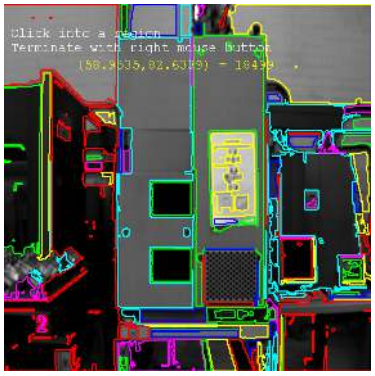


Figure 21.5: Center of gravity and area of selected region.

21.4 Tips & Tricks

21.4.1 Saving Window Content

HALCON provides an easy way to save the content of a graphics window to a file. This can, e.g., be useful for documentation purposes. The corresponding operator is called dump_window. It takes the window handle and the file name as input. The parameter Device allows to select amongst different file formats.

21.4.2 Execution Time

Generally, visualization takes time. To reduce time, it is recommended to visualize only when it is really needed for the specific task. When using HDevEngine (see the Programmer's Guide in part VI on page 137), by default the execution of all dev_* operators is suppressed. Thus, you can use as much display operators as you need while developing with HDevelop but save time when executing the final application from a programming language.

Further influences on the run time concern the graphics card and the bit depth. Choosing a bit depth of 16 instead of 32 in many cases speeds up the program execution significantly. So, if speed is important for you, we recommend to simply try different bit depths, perform typical display operators like disp_image, disp_color, disp_region, or disp_xld, and measure the execution time with count_seconds. Additionally, although it has generally only a small influence on the run time, cases exist where you can speed up the visualization also by choosing a lower screen resolution.

21.5 Advanced Topics

21.5.1 Programming Environments

The handling of graphics windows differs in the different programming environments. Especially HDevelop has a specific way of working with the visualization.

- Because HDevelop is an interactive environment, the handling of windows must be as easy as possible. In particular, it is important to display data without any need of programming. Therefore, the concept of window handles is used only if needed. Normally, the window where the output has to go to is not specified explicitly. Instead, HDevelop makes use of the activation status of the graphics windows. As a consequence, the display operators of HDevelop do not have a parameter for a window handle. Visualization operators in HDevelop look identical to their counterparts of HALCON except that their name starts with `dev_` and that the parameter for the window handle is suppressed. When exporting the code, this missing handle will automatically be inserted.

  The second difference is that the windows in HDevelop have a history to automatically redisplay the data when the window has been resized. This is not the case with standard HALCON graphics windows.

  The last difference is that HDevelop automatically sets the coordinate system according to the current image, whereas you must do this explicitly with programmed code when using HALCON. To make an export to, e.g., C++, C#, or Visual Basic transparent, we recommend to use `dev_set_window` when working with multiple graphics windows and to call `dev_set_part` to specify the coordinate system.

- When using HALCON windows in MFC, the usual way is to use the windows as a subwindow of a parent form. This can easily be achieved by using the window handle of the current form as the father. The handle must be converted to a `long` value that can then be passed to `open_window`. Note that `set_check` is needed to change to exception handling of HALCON in this context.

```
set_window_attr("border_width",0);
set_check("~father");
long lWWindowID = (long)m_hWnd;
open_window(0,0,640,480,lWWindowID,"visible","",&m_lWindowID);
set_check("father");
```

- Opening a HALCON window in Visual Basic is similar to the approach used for MFC. Here, you use the memberhWnd of the form or of another subwindow like a picture box. As an alternative, the `HWindowXCtrl` can be used.

```
Call sys.SetCheck("~father")
Call op.OpenWindow(8, 8, 320, 240, form.hWnd, "", "", WindowHandle)
Call sys.SetCheck("father")
```

21.5.2 Flicker-Free Visualization

For a flicker-free visualization, a sequential call of display operators is not suitable, because after each call the new data will immediately be flushed on the visible screen, which may cause flickering results.

When using `set_window_param` with `'flush'` set to `'false'`, the window is no longer updated after each display of an object. Instead, you have to call `flush_buffer` explicitly to update the contents of the graphics window. An example using this approach is provided under `%HALCONEXAMPLES%/hdevelop/Inspection/Bead-Inspection/apply_bead_inspection_model.hdev`.

21.5.3 Visualization Quality for Regions when Zooming

For performance reasons, the visualization of regions when zooming in or out of the graphics window is not very exact. If you need a better visualization and the speed of the visualization is not critical for your application, you can set the parameter `'region_quality'` of the operator `set_window_param` to `'good'` and select a weighted interpolation for the zooming using the operator `set_part_style`.

21.5.4 Remote Visualization

In some applications, the computer used for processing differs from the computer used for visualization. Such applications can easily be created with HALCON using the socket communication. Operators like `send_image` or `receive_tuple` allow a transparent transfer of the relevant data to the control computer to apply visualization there.

21.5.5 Programmed Visualization

Sometimes it might be necessary not to apply the standard HALCON visualization operators, but to use a self-programmed version. This can be achieved by using the access functions provided for all data types. Examples for these are `get_image_pointer1`, `get_region_runs`, or `get_contour_xld`. Operators like these allow full access to all internal data types. Furthermore, they provide the data in various forms (e.g., runlength encoding, points, or contours) to make further processing easier. Based on this data, a self programmed visualization can be developed easily.

As an alternative, with `set_window_type` the window type `'pixmap'` can be chosen. In this case, all displayed data is painted into an internal buffer that can be accessed with `get_window_pointer3`. The returned pointers reference the three color channels of the buffer. This buffer can then easily be transferred (e.g., to another system) and/or transformed into the desired format. One example for a conversion is to call `gen_image3` to create a HALCON color image.

Chapter 22

Compute Devices

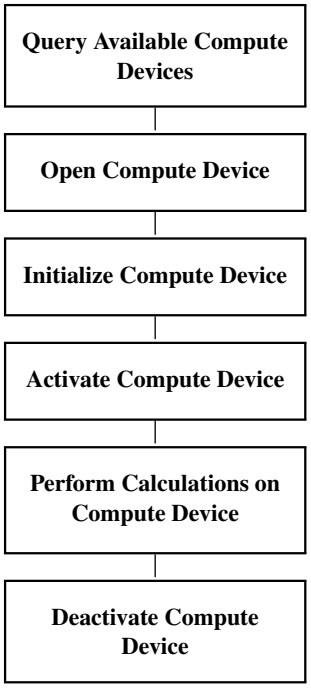
By using compute devices, calculations from the CPU can be transferred to a compute device which leads to a significant speedup of some operators. Using compute devices in HALCON is quite easy. They can simply be activated and deactivated. The use of compute devices is based on OpenCL, which means that all OpenCL compatible graphics cards (e.g., NVIDIA, AMD and Intel cards) are supported. To find out whether operators support compute devices, please refer to the 'Execution Information' paragraph in the operator reference. A list of operators supporting compute devices can also be found at the end of this chapter (see page 245).

Compute devices are especially useful for filter and transformation operations, as well as for color processing, but they are also available for more complex operators like subpixel precise edge extraction. The usage of compute devices is very transparent. As soon as a compute device has been opened and activated, the operators take over and guide the memory transfer and the synchronization.

However, not all applications will benefit from the use of compute devices. This chapter provides an introduction into using compute devices and will also assist you with the decision whether compute devices may be an option to speed up your application or not.

22.1 Basic Concept

Using compute devices for an application basically comprises four obligatory steps.



22.1.1 Query Available Compute Devices

The operator query_available_compute_devices returns a list of available compute devices.

22.1.2 Open Compute Device

The operator open_compute_device opens a compute device for the current thread.

22.1.3 Initialize Compute Device

The operator init_compute_device initializes the compute device and prepares selected operators. If init_compute_device is not called, the initialization is performed on demand.

22.1.4 Activate Compute Device

The operator activate_compute_device enables calculations on the compute device for the current thread.

22.1.5 Perform Calculations on Compute Device

In this step calculations can be performed on a compute device.

22.1.6 Deactivate Compute Device

The operator deactivate_compute_device disables the calculations on a compute device for the current thread.

22.1.7 A First Example

The following program lines show those basic steps in the context of a simple program, demonstrating how to use compute devices with HALCON.

First, available compute devices have to be queried.

```
query_available_compute_devices (DeviceIdentifier)
```

After that, a compute device is opened. It is referenced by DeviceIdentifier and returns the handle DeviceHandle.

```
open_compute_device (DeviceIdentifier, DeviceHandle)
```

Then an image is read.

```
read_image (Image, 'rings_and_nuts')
```

Now, the operator that is supposed to run on the compute device, in this case derivate_gauss, is initialized.

```
init_compute_device(DeviceHandle, 'derivate_gauss')
```

The compute device can then be activated for the current HALCON thread to start the calculations.

```
activate_compute_device (DeviceHandle)
```

Now the calculation can be performed on the compute device.

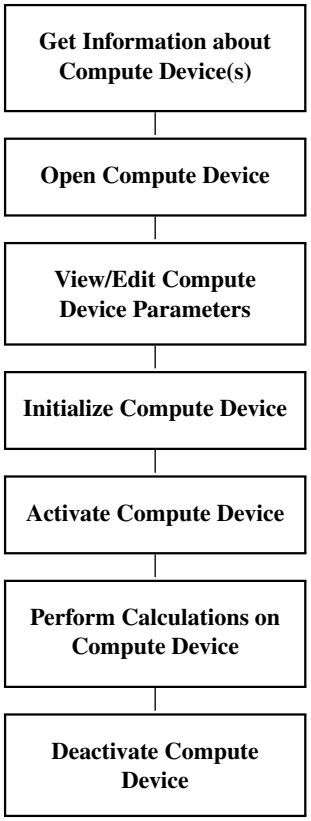
```
derivate_gauss (Image, DerivGauss, 5, 'none')
```

When the compute device `DeviceHandle` is not needed anymore, it can be deactivated.

```
deactivate_compute_device (DeviceHandle)
```

22.2 Extended Concept

In advanced applications, it is required to gain complete control over the compute device process.



22.2.1 Get Information about Compute Device(s)

There are two operators that allow you to obtain general information as well as very specific information about compute devices.

- To find out, which compute devices are available, call the operator `query_available_compute_devices`. This operator returns the list of available compute devices.
- In order to get information details on a certain compute device, call the operator `get_compute_device_info`.

22.2.2 Open Compute Device

To open a compute device for preparation, `open_compute_device` has to be called.

22.2.3 View/Edit Compute Device Parameters

There are two operators that allow you to view and edit parameters for compute devices. The operator `get_compute_device_param` lets you query the set parameters for compute devices and the operator `set_compute_device_param` allows you to change the settings for compute devices.

The following settings can be viewed or edited:

- For image and buffer cache as well as for pinned memory the cache capacity can be set.
- For image and buffer cache as well as for pinned memory the used cache can be queried.
- Furthermore pinned memory, i.e., memory with a fixed address that cannot be moved, can be allocated. Note that the transfer rate for this kind of memory is usually faster, however it is essential to increase the cache for this memory if more may be required.
- Asynchronous execution can be turned on and off.

22.2.4 Initialize Compute Device

To prepare the execution of operators on a compute device, call `init_compute_device`. It is important to call this operator before using a compute device, since it prepares the operators that are going to be used for execution on the compute device by compiling the respective kernel. If `init_compute_device` is not called, the initialization is performed 'on demand' which means that the first time the compute device is used for an operator, it will be significantly slower.

22.2.5 Activate Compute Device

To activate the execution of operators on the compute device, call `activate_compute_device`. With this operator, calculations on the compute device are activated. Note that only one compute device can be active for one thread. The compute device handle is thread-specific.

22.2.6 Perform Calculations on Compute Device

In this step, calculations can be performed on a compute device. See section 22.6 on page 245 for a list of operators that can be calculated on a compute device. **Note that not all parameter values may be available for using an operator on a compute device. Please refer to the operator reference for more information on supported parameters.**

22.2.7 Deactivate Compute Device

To deactivate the execution of operators on the compute device call `deactivate_compute_device`.

22.2.8 Release Compute Device

Note that for most applications, it is `not` necessary to release compute devices with the operator `release_compute_device`. This operator should only be called for freeing device related resources before unloading the HALCON library, if the library was loaded using a mechanism like LoadLibrary or dlopen (to learn more about this operator, please refer to the reference documentation of `release_compute_device`).

22.3 Programming Example

This section gives a brief introduction to using the compute device operators provided by HALCON.

22.3.1 How to Use Compute Devices With HALCON

Example: %HALCONEXAMPLES%/hdevelop/System/Compute-Devices/compute_devices.hdev

The example program is designed to show how to use the different compute device operators in HALCON as well as how to evaluate whether compute devices will speed up an application, in this case an affine transformation.

First, available compute devices are detected and information about the vendor is displayed.

```
query_available_compute_devices (DeviceIdentifier)
disp_message (WindowHandle, \
              'Found ' + |DeviceIdentifier| + ' Compute Device(s):', \
              'window', 12, 12, 'black', 'true')
for Index := 0 to |DeviceIdentifier| - 1 by 1
    get_compute_device_info (DeviceIdentifier[Index], 'name', DeviceName)
    get_compute_device_info (DeviceIdentifier[Index], 'vendor', \
                             DeviceVendor)
    Message[Index] := 'Device #' + Index + ': ' + DeviceVendor + ' ' +  \
                      DeviceName
endfor
disp_message (WindowHandle, Message, 'window', 42, 12, 'white', 'false')
```

To determine whether the application runs faster with or without the use of compute devices, a benchmark can be performed. To prepare the time measurement, the compute device is opened , a previously defined DeviceIndex is used in case several compute devices are found, and the asynchronous execution is deactivated to get comparable results.

```
open_compute_device (DeviceIdentifier[DeviceIndex], DeviceHandle)
set_compute_device_param (DeviceHandle, 'asynchronous_execution', 'false')
```

First the compute device has to be initialized, otherwise the initialization is performed with the first call of the operator which would influence the results of the comparison. Then the compute device is activated.

```
init_compute_device (DeviceHandle, 'affine_trans_image')
activate_compute_device (DeviceHandle)
```

Then input data for the benchmark has to be provided.

```
read_image (Image, 'rings_and_nuts')
```

Then the number of benchmark loops is set.

```
Loops := 200
```

The operator affine_trans_image is then called once to fill the caches. Then the loops are executed and the time it takes on the GPU is calculated.

```
affine_trans_image (Image, ImageAffineTrans, HomMat2D, 'constant', 'false')
count_seconds (Before)
for Index := 1 to Loops by 1
    affine_trans_image (Image, ImageAffineTrans, HomMat2D, 'constant', \
                        'false')
endfor
count_seconds (After)
TimeGPU := (After - Before) * 1000.0 / Loops
```

Since the transfer time is also relevant for the decision whether compute devices will improve speed, a modified version of this benchmark is performed that includes the transfer from CPU to GPU and back. The operator set_grayval ensures that the image must be transferred to the GPU. The operator get_image_pointer1 ensures that ImageAffineTrans is transferred back to the CPU.

```
count_seconds (Before)
for Index := 1 to Loops by 1
    set_grayval (Image, 0, 0, Grayval)
    affine_trans_image (Image, ImageAffineTrans, HomMat2D, 'constant', \
                        'false')
    get_image_pointer1 (ImageAffineTrans, Pointer, Type, Width, Height)
endfor
count_seconds (After)
TimeGPUinclTransfer := (After - Before) * 1000.0 / Loops
```

Then the compute device can be deactivated.

```
deactivate_compute_device (DeviceHandle)
```

Now the same benchmark is performed on the CPU to compare the results.

```
affine_trans_image (Image, ImageAffineTrans, HomMat2D, 'constant', 'false')
count_seconds (Before)
for Index := 1 to Loops by 1
    affine_trans_image (Image, ImageAffineTrans, HomMat2D, 'constant', \
                        'false')
endfor
count_seconds (After)
TimeCPU := (After - Before) * 1000.0 / Loops
SpeedUp := TimeCPU / TimeGPU
```

Finally the runtimes of the CPU and GPU are compared. A result may look like the following example (figure 22.1).

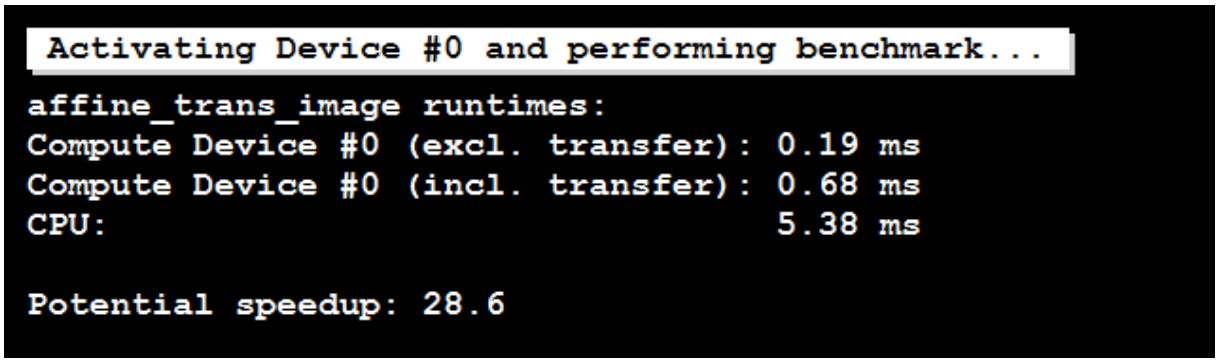


Figure 22.1: A benchmark helps to decide whether compute devices improve the speed of an application. In this case the CPU is a Intel core 2 Duo 3GHz and the compute device is a NVIDIA GeForce GTX 460 graphics card.

22.4 Tips and Tricks

This section is supposed to inform you how to optimize speed while using a compute device but also to help you with the decision whether compute devices may be useful for a certain application or if it may be just as fast to process the whole application on the CPU.

To write an own operator that fits into the compute device framework see the Extension Package example `extension_package/useropencl`.

22.4.1 Speedup

Since many aspects influence whether using compute devices can improve speed, the only way to know for certain which is faster, processing on CPU or on CPU and compute device, is to compare the execution times (for an

example how to compare execution times, see section 22.3 on page 236). However, there are quite a few aspects that may help to improve the speedup or influence the decision whether or not an application benefits from the use of compute devices. These aspects will be presented in the following sections as well as an example of how execution times can be compared quickly.

Requirements for Improving Speed with Compute Devices

The highest speedup can be achieved by

- using a high-end graphics card,
- using hardware with high memory bandwidth, and
- creating a well structured program (group compute device operations to minimize transfer between CPU and compute device).

Memory transfer can be measured with the hbench option (`-memory`).

Note that HALCON's operators are automatically parallelized. This means that the available CPU cores are recognized and the threads are then automatically divided without the need for the programmer to interfere. This way, the speed increases proportionally to the number of processors since operators with AOP are significantly faster. Therefore it is very important to compare CPU and compute device performance on the same hardware that will be used in the subsequent application.

Aspects to be Considered

Note that the transfer of data between CPU and compute device is a bottleneck. Using compute devices may only be an efficient solution if the whole application is faster, i.e., execution time plus transfer time.

Note also that the energy consumption of a graphics card is very high. Graphics cards may also not be designed for use in an industrial environment. Therefore problems concerning the graphics cards may be:

- They might not be robust against heat or vibrations.
- They may not correct memory errors (ECC).
- They may only perform single-precision floating point operations.

Other aspects that should be considered concerning the transfer are:

- Transfer rate is usually larger for large block sizes.
- Transfer is usually faster when using page locked (pinned) memory (i.e. memory with a fixed address that can be set via `set_compute_device_param`).

Using Compute Devices does not Always Improve Speed

It is unlikely that transferring calculations to the compute device will improve speed if

- the CPU is already very fast,
- fast operators are used,
- the transfer times are slow because of
  - memory bandwidth or
  - large images.

The Influence of Image Size on Speed

Improving the speed of calculations works most efficiently for large images. Benchmarks have shown that images with a width that can be divided by four can be calculated faster. Therefore, the performance can be optimized if an image is either cropped or enlarged such that the width is a multiple of 4 byte, or even better 16 byte, before it is processed on the compute device.

Note, however, that for some operators performing geometric transformations, the image will be calculated on the CPU instead of being transfered to the compute device if it is too large. The operator `get_compute_device_info` can be used to query image width and height that is supported by the compute device.

Operators

Operators that return sub-pixel precise XLD contours perform in two steps, filtering and extracting the XLD contours.

If calculations are executed on the compute device, the filtering is performed directly on the compute device. The filtered images are transfered back to the CPU for XLD extraction.

This transfer back to the CPU is relevant for the total execution time. For this reason, it is important to have a PC with high transfer rate between CPU- and compute device memory.

Allocating Cache for Maximum Performance

Two kinds of caches are used on the compute device, buffer (for filters like `mean_image` or `sobel_amp`) and images (for geometric transformations). For maximum performance, we recommend to allocate those two kinds of cache before executing an operator. Otherwise, the runtime will be higher since memory is allocated dynamically. The default of those caches is one third of the available memory each (to change the maximum size of the compute device image cache, use `set_compute_device_param` (`DeviceHandle, 'buffer_cache_capacity'`)).

22.4.2 Measuring Execution Times

When using compute devices, processes on the compute device and on the CPU are performed asynchronously by default. A visualization of asynchronous processing can be found in figure 22.2.

Asynchronous processing has many advantages:

- Operators that are processed on the CPU do not have to wait for the compute device operators.
- CPU and compute device process in parallel.

However, due to the asynchronous processing even though the general execution time may decrease, the execution time of single operator may not be measured correctly because an operator might wait for another one. Therefore, to get the exact execution times, asynchronous processing has to be turned off.

22.4.2.1 Using Synchronous Processing for Test Purposes

If synchronous processing is required, e.g., if execution time of a single operator should be measured with `set_compute_device_param`, the generic parameter `DeviceHandle 'asynchronous_execution'` should be set to `'false'`. Note however that for measuring the execution time of an application, the default setting `'true'` should be used to achieve the best performance.

The example `%HALCONEXAMPLES%/hdevelop/System/Compute-Devices/compute_devices.hdev` shows how to perform a benchmark (see also section 22.3.1 on page 237) and how to make sure that the image is really transfered to the compute device and therefore if the transfer time is included in the time measurements.

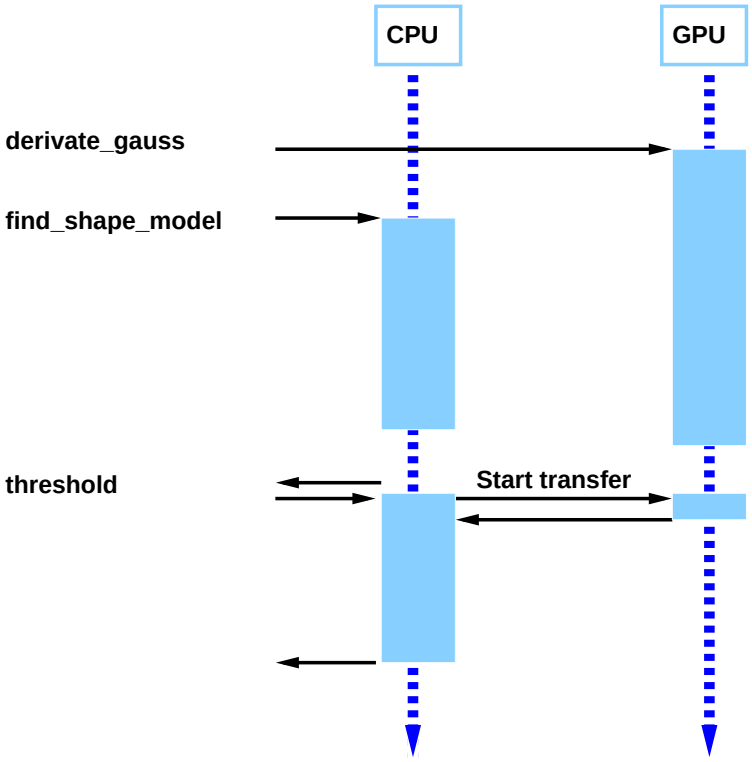


Figure 22.2: As default, operators are processed asynchronously on CPU and compute device.

22.4.3 Exchanging or Simulating Operators that do not support Compute Devices

Some operators that do not support compute devices can be simulated by other operators.

Use the operator `convol_image` to replace linear filters like

- `laplace`,
- `bandpass_image`,
- or any other arbitrary filter mask.

22.4.3.1 Domain and Region Processing

All compute devices that are addressed via OpenCL are optimized for the processing of images not regions. Therefore the domain cannot be used to reduce the area to which the operator is applied. The main reason for this is that the whole image needs to be transferred to the compute device.

These operators are

- `binomial_filter`,
- `convol_image`,
- `derivate_gauss`,
- `median_image`,
- `sobel_amp`,
- `sobel_dir`,
- `lines_gauss`,

- edges_image,
- edges_sub_pix

To reduce transfer and execution times of images with reduced domains as far as possible, crop the image, e.g., with the operator crop_domain before they are transferred to the compute device. Note that, depending on the filter size of the operator to be executed, the domain of the image should be suitably enlarged before the call of crop_domain.

22.4.3.2 Simulating Region Processing

In this paragraph, the simulation of region processing is shown for the example of color segmentation. Color segmentation can be sped up by using compute devices instead of just the CPU. In order to calculate the complete color segmentation on a compute device and therefore reduce transfer time, region processing has to be simulated. This is done by creating binary images, i.e., images with exactly two gray values. In the following example, these gray values are 0 and 255. Pixels with a gray value of 0 are defined as outside of the region whereas pixels with a value of 255 are recognized as within the region. Therefore regular morphology operators (like, e.g. dilation_rectangle1 can be replaced by gray-value morphology operators (like, e.g., gray_dilation_rect).

In the example below, the blue plastic part of a fuse should be detected. This example first shows which operator calls solve this problem on the CPU and then which operators are used for the same solution on a compute device. figure 22.3 shows the results of each step of the region processing on the CPU compared to the results of simulated region processing on a compute device.

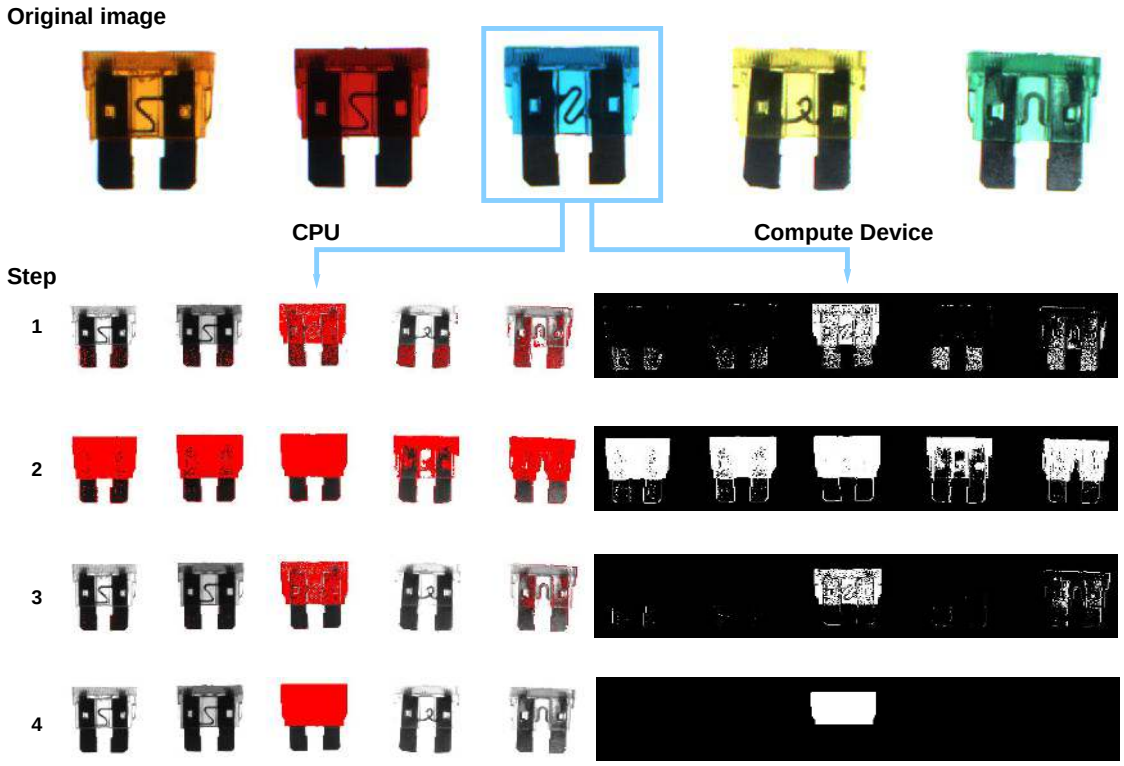


Figure 22.3: Region processing for color images on the CPU and simulated region processing on a compute device. Steps (CPU:GPU) 1) threshold : lut_trans (Hue) 2) threshold : lut_trans (Saturation), 3) intersection : min_image 4) dilation_rectangle1 and erosion_rectangle1 : gray_dilation_rect and gray_erosion_rect.

.

First, an image is read. For further processing, the three-channel image is converted into three images.

```
read_image (ColorFuses00, 'color/color_fuses_00.png')
decompose3 (ColorFuses00, Image1, Image2, Image3)
```

The image is subsequently transformed into the color space 'hsv' with the operator trans_from_rgb. Then the region is processed on the CPU. For this purpose, regions with a certain hue and saturation are selected (see figure 22.3, steps 1 and 2).

```
trans_from_rgb (Image1, Image2, Image3, Hue, Saturation, ImageResult3, \
                'hsv')
threshold (Hue, Region, 125, 145)
threshold (Saturation, Region1, 100, 255)
```

Now to combine the two regions, the intersection of the two resulting regions of the thresholds is calculated (see figure 22.3 on page 242, step 3). To reduce this region to the blue plastic part of the fuse, the morphology operators dilation_rectangle1 and erosion_rectangle1 are used (see figure 22.3 on page 242, step 4). Now the upper part of the blue fuse can be determined.

```
intersection (Region, Region1, RegionIntersection)
dilation_rectangle1 (RegionIntersection, RegionDilation, 11, 11)
erosion_rectangle1 (RegionDilation, RegionErosion, 91, 31)
dilation_rectangle1 (RegionErosion, RegionDilation1, 81, 21)
```

The same process can be simulated on a compute device. A fast solution to start processing color images on a compute device is to transfer the Bayer image to the compute devices where the operator cfa_to_rgb can be executed to transform it into a color image.

Only a subset of the color space transformations can be processed on the compute device with trans_from_rgb (as shown in this example) and trans_to_rgb.

```
trans_from_rgb (Image1, Image2, Image3, Hue, Saturation, ImageResult3, \
                'hsv')
```

All other color space transformations can be simulated with linear_trans_color. If the color in an image changes, this effect can be corrected by modifying TransMat as explained in the reference documentation of linear_trans_color.

Instead of using a threshold like on the CPU, the image is scaled such that all pixels belonging to the object are mapped to the gray value 255 and all that are outside of the object are mapped to 0. To speed up this mapping, a LUT transformation is applied. Two look-up tables are used for further processing.

The first LUT maps three gray-value ranges. The values

- 0 to 124 are mapped to 0,
- 125 to 145 are mapped to 255, and
- 146 to 255 are mapped to 0.

```
LUT_125_145 := [gen_tuple_const(125,0),gen_tuple_const(21,255), \
                gen_tuple_const(110,0)]
```

The second LUT maps two gray-value ranges. The values

- 0 to 99 are mapped to 0 and
- 100 to 255 are mapped to 255.

```
LUT_100_255 := [gen_tuple_const(100,0),gen_tuple_const(156,255)]
```

The result of using the defined LUTs with the operators lut_trans for hue and saturation are two byte images with only two different gray values: 0 and 255 (see figure 22.3 on page 242, steps 1 and 2).

```
lut_trans (Hue, HueThresh, LUT_125_145)
lut_trans (Saturation, SaturationThresh, LUT_100_255)
```

By calculating the minimum of both images and using the two gray-value morphology operators gray_dilation_rect and gray_erosion_rect, the blue plastic part of the fuse can be determined (see figure 22.3 on page 242, steps 3 to 4).

```
min_image (HueThresh, SaturationThresh, ImageIntersection)
gray_dilation_rect (ImageIntersection, ImageMax, 11, 11)
gray_erosion_rect (ImageMax, ImageMin, 31, 91)
gray_dilation_rect (ImageMin, ImageMax1, 21, 81)
```

Note that it must be considered that the runtime of gray-value morphology operators increases with the size of the filter masks whereas operators like opening_rectangle1 that perform region morphology have a constant runtime that is independent from the size of the structuring element.

Operators using regions like union, intersection, and difference are simulated on the compute device with the operators max_image, min_image, and sub_image if the regions are represented as binary images.

22.4.4 Limitations

For other operators, only certain parameters may be supported for being processed on a compute device.

A few examples are listed here:

- median_image can only be used with filter mask sizes 3x3 and 5x5
- derivate_gauss can only be used with a limited number of filters and maximum smoothing with sigma 20.7 (depending on the filter this value can even be lower),
- trans_from_rgb can only be used for color spaces 'cielab', 'hsv' and 'hsi'.

These are just some examples for limitations. **Remember to always check which parameter settings are available for an operator that is processed on a compute device.** If only a certain subset of values is available for a parameter, this information is listed just beneath the regular values under `List of values (for compute devices):`.

For more information on limitations, please refer to the Installation Guide (in section 1.5.1 on page 12).

22.4.5 Multithreading

22.4.5.1 Using One Compute Device

To use one compute device in different threads, the compute device needs to be opened in each thread separately. Thus, one compute device is represented by different compute device handles. Note that each compute device handle is treated as a separate device and cannot access data stored on the other 'devices' without copying to the CPU first.

We do not advice to use one compute device in multiple threads, because you need a lot of synchronization logic to make sure that the compute device is not used simultaneously by another thread, and that the results are shared correctly between threads. Thus, you may end up with a complicated code.

If you look at the operator reference for e.g., activate_compute_device under the heading 'Execution Information', you will see that the multithreading scope is local. This means that compute device operators must only be called from the thread that created the compute device handle, and compute device handles must not be shared across threads.

Note, however, if you try to access the same image object from different threads using the same compute device, an error will be raised.

22.4.5.2 Using More Than One Compute Device

If planning to use more than one compute device, it should be checked whether the PC hardware is able to handle this. It is important to always install the latest graphics driver. If the compute device is accessed via a Linux system, the user must be member of the 'video' group.

By default only one compute device is used if the compute device is activated with `activate_compute_device`. If another compute device has been activated previously, this compute device is automatically deactivated for the current thread.

It is, however, possible to use more than one compute device. Then, one thread per device must be used. Each thread connects to its compute device and starts processing. Since this does not cost much execution time, all threads can run on the same CPU.

22.5 Technical Details

Please refer to the websites of NVIDIA, ATI/AMD and Intel for a list of compatible cards. To check the performance of a card, it is useful to measure the runtimes and transfer times which can be done via `hbench -compute_device [-ref]` or as described in the example section 22.3 on page 236.

The following technical characteristics should be taken into account when considering to use compute devices:

- **Power/Performance Ratio** In some applications the power consumption and/or the emitted heat is of importance. It can therefore be useful to compare different cards with respect to their performance and consumed power.
- **Robustness** It is useful to check whether a compute device was designed for industrial use.

22.6 Operators Supporting Compute Devices

If an operator supports compute devices, this is stated in the section "Execution Information" of its entry in the operator reference manual. This information can also be queried with the operator `get_operator_info`. The following HDevelop program generates a list of all operators that support compute devices:

```
get_operator_name ('', Operators)
GPUOperators := []
SupportedDevices := []
for I := 0 to |Operators|-1 by 1
    get_operator_info (Operators[I], 'compute_device', Information)
    if (Information != 'none')
        GPUOperators := [GPUOperators, Operators[I]]
        SupportedDevices := [SupportedDevices, Information]
    endif
endfor
```

Chapter 23

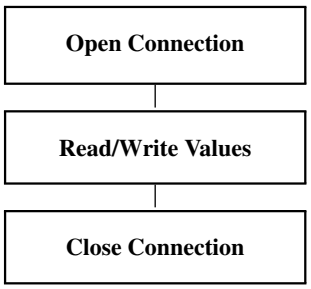
I/O Devices

HALCON supports I/O devices to perform data-acquisition, e.g., to read sensor values or write control data. To let you concentrate on the actual machine vision problem, HALCON provides you with interfaces performing this interaction for a large number of I/O devices (see http://www.mvtec.com/products/interfaces).

Within your HALCON application, communication with I/O devices is thus reduced to a few lines of code, i.e., a few operator calls. What's more, this simplicity is not achieved at the cost of limiting the available functionality.

23.1 Basic Concept

Communicating with I/O devices basically consists of three steps.



23.1.1 Open Connection

If you want to read or write values on an I/O device, the first step is to connect to this device and open a transmission channel. HALCON relieves you of all device-specific details; all you need to do is to call the operator open_io_device, specifying the name of the corresponding I/O device interface. Once the connection is established, the available transmission channels can be queried with query_io_device. A transmission channel is opened using the operator open_io_channel. Afterwards, the transmission channel is used to read and write actual values.

23.1.2 Read/Write Values

Having opened a transmission channel, you read and write values using the operators read_io_channel and write_io_channel, respectively.

23.1.3 Close Image Acquisition Device

At the end of the application, you close the connection to the I/O transmission channel and the device to free its resources by first calling the operator close_io_channel, followed by the operator close_io_device.

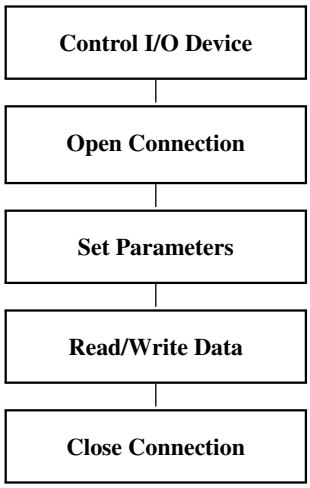
23.1.4 A First Example

The following code reads values from an I/O device of name IoInterfaceName:

```
open_io_device (IOInterfaceName, DeviceName[0], [], [], IoDeviceHandle)
query_io_device (IoDeviceHandle, [], 'io_channel_names.digital_input', \
                 ChannelsInput)
open_io_channel (IoDeviceHandle, ChannelsInput[Channel0], [], [], IoHandle0)
read_io_channel (IoHandle0, Value0, Status0)
close_io_channel ([IoHandle0,IoHandle1])
close_io_device (IoDeviceHandle)
```

23.2 Extended Concept

Many applications involving I/O devices are straightforward, but in some cases additional configuration work is required. Therefore, HALCON allows to further parameterize the data acquisition process.



23.2.1 Control I/O Device Interface

Depending on the used I/O device interface, certain actions may be performed on the interface before a connection to an actual device is being established. As an example, the OPC UA interface uses this mechanism to handle certificates for encrypted data acquisition sessions. These I/O interface actions are performed using the operator `control_io_interface`. See the documentation of your I/O device interface for detailed information.

23.2.2 Open Connection

You can query the available devices and other information of your I/O device interface using the operator `query_io_interface`. When connecting to your date acquisition device with `open_io_device`, the main parameters are the name of the corresponding HALCON I/O device interface and the name of the device itself. As a result, you obtain a so-called handle, with which you can access the device later, e.g., to open a transmission channel. Using the obtained handle of the transmission channel, you can acquire date with `read_io_channel` or `write_io_channel`.

With the generic parameters of `open_io_device` you can describe the configuration of your I/O device, which is necessary when using more complex configurations.

Detailed information about the parameters of `open_io_device` can be found in the documentation of your I/O device interface. More information is available for download from MVTec's web server under `http://www.mvtec.com/products/interfaces`.

23.2.3 Set Parameters

As described above, you can already set parameters when connecting to the I/O device with open_io_device. These parameters are the so-called *generic parameters*, because they are specific to the corresponding I/O interfaces. However, data acquisition devices differ widely regarding the provided functionality, leading to many more *special parameters*. These parameters can be customized with the operator set_io_device_param.

With the operator get_io_device_param you can query the current values of the parameters.

23.3 Programming Examples

Example programs for all provided I/O interfaces can be downloaded via the "MVTec Software Manager" (SOM) or at http://www.mvtec.com/products/interfaces.

23.4 Tips & Tricks

23.4.1 Unsupported I/O Devices

If you want to use an I/O device that is currently not supported by HALCON, i.e., for which no HALCON I/O device interface exists, you can create your own interface. A description how to create and integrate an I/O device interface as well as a template source code that can be used as the basis of an integration can be downloaded from MVTec's web server under http://www.mvtec.com/products/interfaces.

Index