HALCON

a product of MVTec

Solution Guide III-C

3D Vision

HALCON 24.11 *Progress-Steady*

MVTec Software GmbH

Building Vision for Business

Machine vision in 3D world coordinates, Version 24.11.1.0

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

About This Manual

Measurements in 3D become more and more important. HALCON provides many methods to perform 3D measurements. This Solution Guide gives you an overview over these methods, and it assists you with the selection and the correct application of the appropriate method.

A short characterization of the various methods is given in chapter 1 on page 9. Principles of 3D transformations and poses as well as the description of the camera model can be found in chapter 2 on page 13. Afterwards, the methods to perform 3D measurements are described in detail.

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

Symbols

The following symbol is used within the manual:

This symbol indicates an information you should **pay attention** to.

Contents

Chapter 1

Introduction

With HALCON you can perform 3D vision in various ways. The main applications comprise the 3D position recognition and the 3D inspection, which both consist of several different approaches with different characteristics, so that for a wide range of 3D vision tasks a proper solution can be provided. This Solution Guide provides you with detailed information on the available approaches, including also some auxiliary methods that are needed only in specific cases.

An introduction to 3D vision and many other topics is available in interactive online courses at our MVTec Academy.

What Basic Knowledge Do You Need for 3D Vision?

Typically, you have to calibrate your camera(s) before applying a 3D vision task. Especially, if you want to achieve accurate results, the **camera calibration** is essential, because it is of no use to extract edges with an accuracy of 1/40 pixel if the lens distortion of the uncalibrated camera accounts for a couple of pixels. This also applies if you use cameras with telecentric lenses. But don't be afraid of the calibration process: In HALCON, this can be done with just a few lines of code. To prepare you for the camera calibration, chapter 2 on page 13 introduces you to the details on the camera model and parameters. The actual camera calibration is then described in chapter 3 on page 59.

Using a camera calibration, you can transform image processing results into arbitrary 3D coordinate systems and thus derive metrical information from images, regardless of the position and orientation of the camera with respect to the object. In other words, you can perform inspection tasks in 3D coordinates in specified object planes, which can be oriented arbitrarily with respect to the camera. This is, e.g., useful if the camera cannot be mounted such that it looks perpendicular to the object surface. Thus, besides the pure camera calibration, chapter 3 shows how to apply a general **3D vision task with a single camera in a specified plane**. Additionally, it shows how to **rectify the images** such that they appear as if they were acquired from a camera that has no lens distortions and that looks exactly perpendicular onto the object surface. This is useful for tasks like OCR or the recognition and localization of objects, which rely on images that are not distorted too much with respect to the training images.

Before you develop your application, we recommend to read chapter 2 and chapter 3 and then, depending on the task at hand, to step into the section that describes the 3D vision approach you selected for your specific application.

How Can You Obtain an Object's 3D Position and Orientation?

The position and orientation of 3D objects with respect to a given 3D coordinate system, which is needed, e.g., for pick-and-place applications (3D alignment), can be determined by one of the methods described in chapter 4 on page 91:

- The pose estimation of a known 3D object from **corresponding points** (section 4.1 on page 92) is a rather general approach that includes a camera calibration and the extraction of at least three significant points for which the 3D object coordinates are known. The approach is also known as "mono 3D".
- HALCON's **3D matching** locates known 3D objects based on a 3D model of the object. In particular, it automatically searches objects that correspond to a 3D model in the search data and determines their 3D poses. The model must be provided, e.g., as a Computer Aided Design (CAD) model. Available approaches

are the *shape-based 3D matching* (section 4.2 on page 95) that searches the model in 2D images and the *surface-based 3D matching* (section 4.3 on page 104) that searches the model in a 3D scene, i.e., in a set of 3D points that is available as 3D object model, which can be obtained by a 3D reconstruction approach like stereo or sheet of light. Note that the surface-based matching is also known as "volume matching", although it only relies on points on the object's surface.

- HALCON's **3D primitives fitting** (section 4.5 on page 111) fits a primitive 3D shape like a cylinder, sphere, or plane into a 3D scene, i.e., into a set of 3D points that is available as a 3D object model, which can be obtained by a 3D reconstruction approach like stereo or sheet of light followed by a 3D segmentation.
- The calibrated **perspective matching** locates perspectively distorted planar objects in images based on a 2D model. In particular, it automatically searches objects that correspond to a 2D model in the search images and determines their 3D poses. The model typically is obtained from a representative model image. Available approaches are the *calibrated perspective deformable matching* (section 4.6 on page 114) that describes the model by its contours and the *calibrated descriptor-based matching* (section 4.7 on page 114) that describes the model by a set of distinctive points that are called "interest points".
- The **circle pose estimation** (section 4.8 on page 115) and **rectangle pose estimation** (section 4.9 on page 116) use the perspective distortions of circles and rectangles to determine the pose of planar objects that contain circles and/or rectangles in a rather convenient way.

How Can You Inspect a 3D Object?

The inspection of 3D objects can be applied by different means. If the inspection in a specified plane is sufficient, you can **use a camera calibration together with a 2D inspection** as is described in chapter 3 on page 59.

If the surface of the 3D object is needed and/or the inspection can not be reduced to a single specified plane, you can **use a 3D reconstruction together with a 3D inspection**. That is, you use the point, surface, or height information returned for a 3D object by a 3D reconstruction and inspect the object, e.g., by comparing it to a reference point, surface, or height.

Figure 1.1 provides you with an overview on the methods that are available for 3D position recognition and 3D inspection. For an introduction to 3D object models, please refer to section 2.3 on page 38.

How Can You Reconstruct 3D Objects?

To determine points on the surface of arbitrary objects, the following approaches are available:

- HALCON's **stereo vision** functionality (chapter 5 on page 117) allows to determine the 3D coordinates of any point on the object surface based on two (binocular stereo) or more (multi-view stereo) images that are acquired suitably from different points of view (typically by separate cameras). Using multi-view stereo, you can reconstruct a 3D object in full 3D, in particular, you can reconstruct it from different sides.
- A laser triangulation with **sheet of light** (chapter 6 on page 147) allows to get a height profile of the object. Note that besides a single camera, additional hardware, in particular a laser line projector and a unit that moves the object relative to the camera and the laser, is needed.
- With **depth from focus** (DFF) (chapter 7 on page 163) a height profile can be obtained using images that are acquired by a single telecentric camera but at different focus positions. In order to vary the focus position additional hardware like a translation stage or linear piezo stage is required. Note that depending on the direction in which the focus position is modified, the result corresponds either to a height image or to a distance image. A height image contains the distances between a specific object or measure plane and the object points, whereas the distance image typically contains the distances between the camera and the object points. Both can be called also depth image or "Z image".
- With **photometric stereo** (Reference Manual, chapter "3D Reconstruction ▷ Photometric Stereo") a height image can be obtained using images that are acquired by a single telecentric camera but with at least three different telecentric illumination sources for which the spatial relations to the camera must be known. Note that the height image reflects only relative heights, i.e., with photometric stereo no calibrated 3D reconstruction is possible.
- With **structured light** (Solution Guide I, chapter 8 on page 69) a 3D object model is derived from projected patterns on a diffuse object surface. The setup consists of calibrated camera and projector.

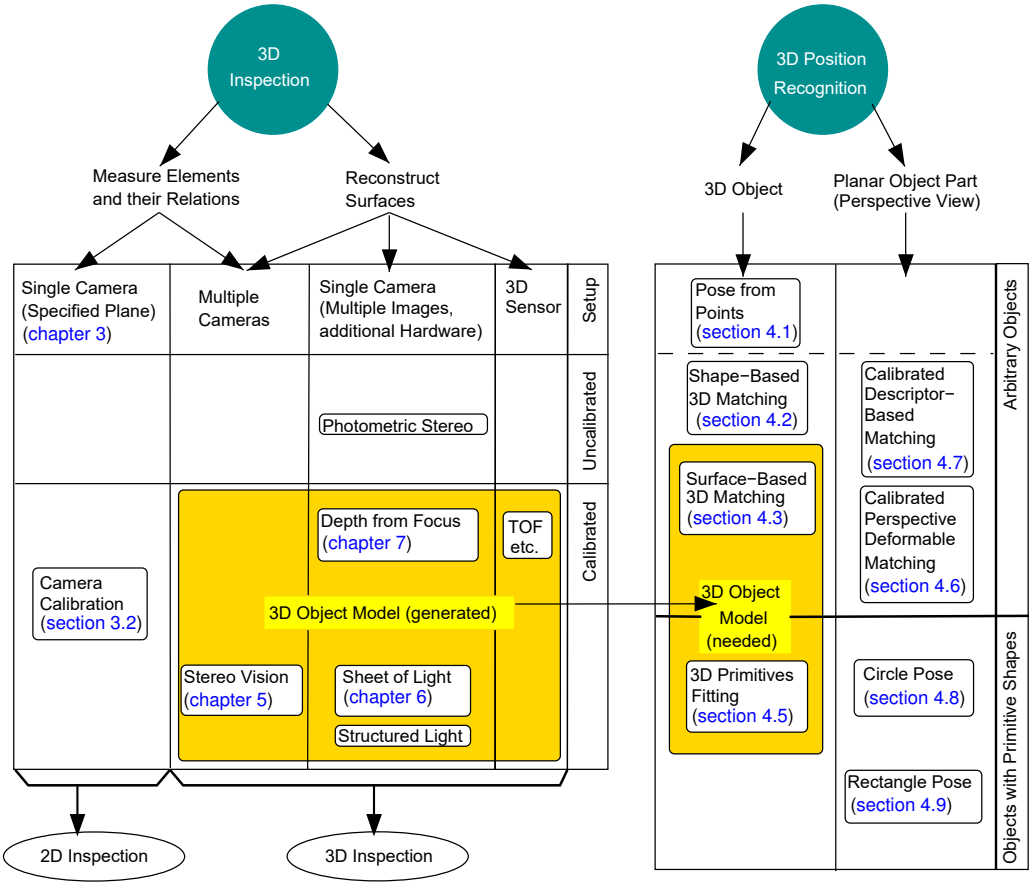


Figure 1.1: Overview to the main methods used for 3D Vision.

- Besides the 3D reconstruction approaches provided by HALCON, you can obtain 3D information also by **specific 3D sensors** like time of flight (TOF) cameras. These cameras typically are calibrated and return X, Y, and Z images.

Figure 1.2 allows to compare some important features of the different 3D reconstruction approaches like the approach-specific result types.

How Can You Extend 3D Vision to Robot Vision?

A typical application area for 3D vision is **robot vision**, i.e., using the results of machine vision to command a robot. In such applications you must perform an additional calibration: the so-called hand-eye calibration, which determines the relation between camera and robot coordinates (chapter 8 on page 175). Again, this calibration must be performed only once (offline). Its results allow you to quickly transform machine vision results from camera into robot coordinates.

What Tasks May be Needed Additionally?

If the object that you want to inspect is too large to be covered by one image with the desired resolution, multiple images, each covering only a part of the object, can be combined into one larger **mosaic image**. This can be done either based on a calibrated camera setup with very high precision (chapter 9 on page 191) or highly automated for arbitrary and even varying image configurations (chapter 10 on page 205).

If an image shows distortions that are different to the common perspective distortions or lens distortions, caused, e.g., by a non-flat object surface, the so-called **grid rectification** can be applied to rectify the image (chapter 11 on page 219).

3D Reconstruction Approach	Hardware Requirements	Object Size	Possible Results
Multi-View Stereo	multiple cameras, calibration object	approx. > 10 cm	3D object model or X, Y, Z coordinates
Binocular Stereo	two cameras, calibration object	approx. > 10 cm	X, Y, Z coordinates, approach-specific disparity image, or Z image
Sheet of Light	camera, laser line projector, unit to move the object, and calibration object	object must fit onto the moving unit	3D object model, X, Y, Z images, or approach-specific disparity image
Depth from Focus	telecentric camera, hardware to variate the focus position	approx. < 2cm	Z image
Photometric Stereo	telecentric camera, at least three telecentric illumination sources	restricted by field of view of telecentric lens	Z image
Structured Light	camera, projector	few mm^2 to several m^2	3D object model
3D Sensors	special camera like calibrated TOF	approx. 30cm-5m	X, Y, Z images

Figure 1.2: 3D reconstruction: a coarse comparison.

Chapter 2

Basics

2.1 3D Transformations and Poses

Before we start explaining how to perform 3D vision with HALCON, we take a closer look at some basic questions regarding the use of 3D coordinates:

- How to describe the transformation (translation and rotation) of points and coordinate systems,
- how to describe the position and orientation of one coordinate system relative to another, and
- how to determine the coordinates of a point in different coordinate systems, i.e., how to transform coordinates between coordinate systems.

In fact, all these tasks can be solved using one and the same means: homogeneous transformation matrices and their more compact equivalent, 3D poses.

2.1.1 3D Coordinates

The position of a 3D point P is described by its three coordinates (x_p, y_p, z_p). The coordinates can also be interpreted as a 3D vector (indicated by a bold-face lower-case letter). The coordinate system in which the point coordinates are given is indicated to the upper right of a vector or coordinate. For example, the coordinates of the point P in the camera coordinate system (denoted by the letter c) and in the world coordinate system (denoted by the letter w) would be written as:

$$\mathbf{p}^c = \begin{pmatrix} x_p^c \\ y_p^c \\ z_p^c \end{pmatrix} \qquad \mathbf{p}^w = \begin{pmatrix} x_p^w \\ y_p^w \\ z_p^w \end{pmatrix}$$

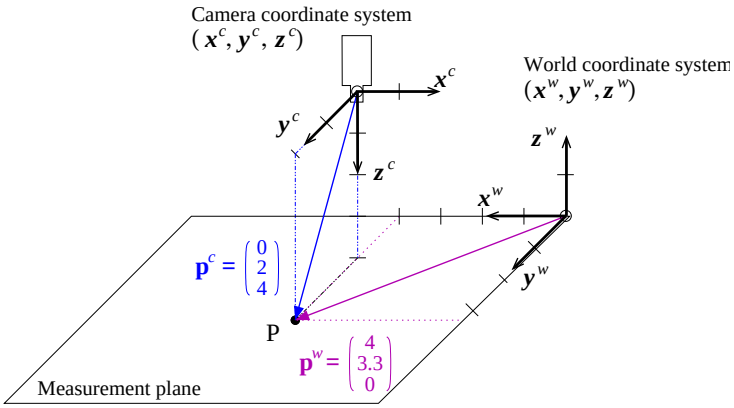


Figure 2.1: Coordinates of a point in two different coordinate systems.

Figure 2.1 depicts an example point lying in a plane where measurements are to be performed and its coordinates in the camera and world coordinate system, respectively.

2.1.2 Transformations using 3D Transformation Matrices

2.1.2.1 Translation

Translation of Points

In figure 2.2, our example point has been translated along the x-axis of the camera coordinate system.

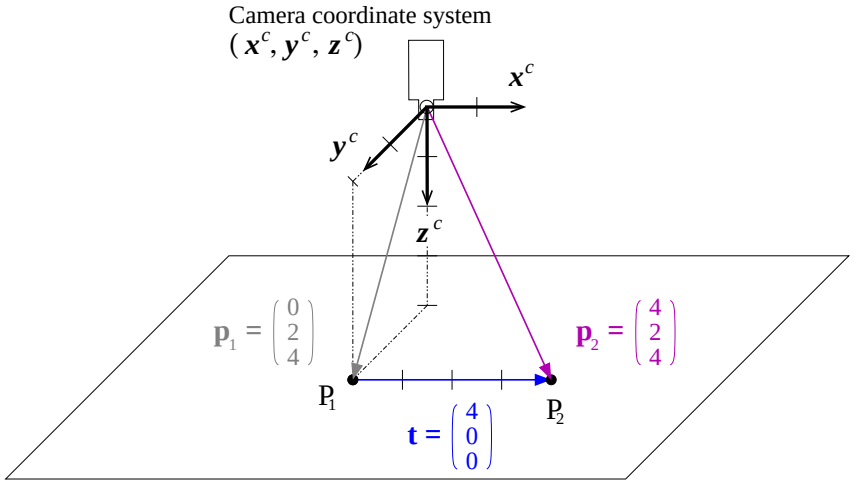


Figure 2.2: Translating a point.

The coordinates of the resulting point P_2 can be calculated by adding two vectors, the coordinate vector $\mathbf{p}_1$ of the point and the translation vector $\mathbf{t}$:

$$\mathbf{p}_2 = \mathbf{p}_1 + \mathbf{t} = \begin{pmatrix} x_{p_1} + x_t \\ y_{p_1} + y_t \\ z_{p_1} + z_t \end{pmatrix} \tag{2.1}$$

Multiple translations are described by adding the translation vectors. This operation is *commutative*, i.e., the sequence of the translations has no influence on the result.

Translation of Coordinate Systems

Coordinate systems can be translated just like points. In the example in figure 2.3, the coordinate system c_1 is translated to form a second coordinate system, c_2. Then, the position of c_2 in c_1, i.e., the coordinate vector of its origin relative to c_1 ($\mathbf{o}_{c_2}^{c_1}$), is identical to the translation vector:

$$\mathbf{t}^{c_1} = \mathbf{o}_{c_2}^{c_1} \tag{2.2}$$

Coordinate Transformations

Let's turn to the question how to transform point coordinates between (translated) coordinate systems. In fact, the translation of a point can also be thought of as translating it together with its local coordinate system. This is depicted in figure 2.3: The coordinate system c_1, together with the point Q_1, is translated by the vector $\mathbf{t}$, resulting in the coordinate system c_2 and the point Q_2. The points Q_1 and Q_2 then have the same coordinates relative to their local coordinate system, i.e., $\mathbf{q}_1^{c_1} = \mathbf{q}_2^{c_2}$.

If coordinate systems are only translated relative to each other, coordinates can be transformed very easily between them by adding the translation vector:

$$\mathbf{q}_2^{c_1} = \mathbf{q}_2^{c_2} + \mathbf{t}^{c_1} = \mathbf{q}_2^{c_2} + \mathbf{o}_{c_2}^{c_1} \tag{2.3}$$

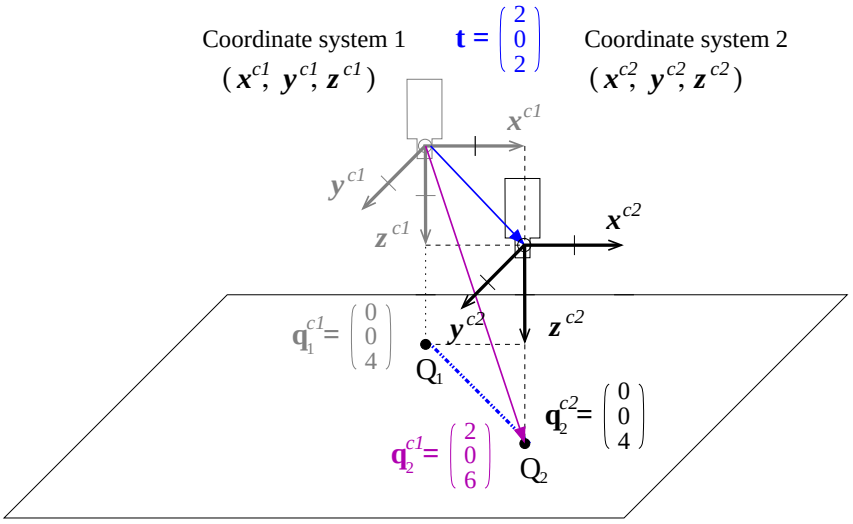


Figure 2.3: Translating a coordinate system (and point).

In fact, figure 2.3 visualizes this equation: $\mathbf{q}_2^{c_1}$, i.e., the coordinate vector of Q_2 in the coordinate system c_1, is composed by adding the translation vector $\mathbf{t}$ and the coordinate vector of Q_2 in the coordinate system c_2 ($\mathbf{q}_2^{c_2}$).

The downside of this graphical notation is that, at first glance, the direction of the translation vector appears to be contrary to the direction of the coordinate transformation: The vector points from the coordinate system c_1 to c_2, but transforms coordinates from the coordinate system c_2 to c_1. According to this, the coordinates of Q_1 in the coordinate system c_2, i.e., the inverse transformation, can be obtained by subtracting the translation vector from the coordinates of $Q1$ in the coordinate system c_1:

$$\mathbf{q}_1^{c_2} = \mathbf{q}_1^{c_1} - \mathbf{t}^{c_1} = \mathbf{q}_1^{c_1} - \mathbf{o}_{c_2}^{c_1} \tag{2.4}$$

Summary

- Points are translated by adding the translation vector to their coordinate vector. Analogously, coordinate systems are translated by adding the translation vector to the position (coordinate vector) of their origin.
- To transform point coordinates from a translated coordinate system c_2 into the original coordinate system c_1, you apply the same transformation to the points that was applied to the coordinate system, i.e., you add the translation vector used to translate the coordinate system c_1 into c_2.
- Multiple translations are described by adding all translation vectors; the sequence of the translations does not affect the result.

2.1.2.2 Rotation

Rotation of Points

In figure 2.4a, the point $\mathbf{p}_1$ is rotated by $-90°$ around the z-axis of the camera coordinate system.

Rotating a point is expressed by multiplying its coordinate vector with a 3×3 rotation matrix $\mathbf{R}$. A rotation around the z-axis looks as follows:

$$\mathbf{p}_3 = \mathbf{R}_z(\gamma) \cdot \mathbf{p}_1 = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x_{p_1} \\ y_{p_1} \\ z_{p_1} \end{pmatrix} = \begin{pmatrix} \cos\gamma \cdot x_{p_1} - \sin\gamma \cdot y_{p_1} \\ \sin\gamma \cdot x_{p_1} + \cos\gamma \cdot y_{p_1} \\ z_{p_1} \end{pmatrix} \tag{2.5}$$

Rotations around the x- and y-axis correspond to the following rotation matrices:

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \qquad \mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{2.6}$$

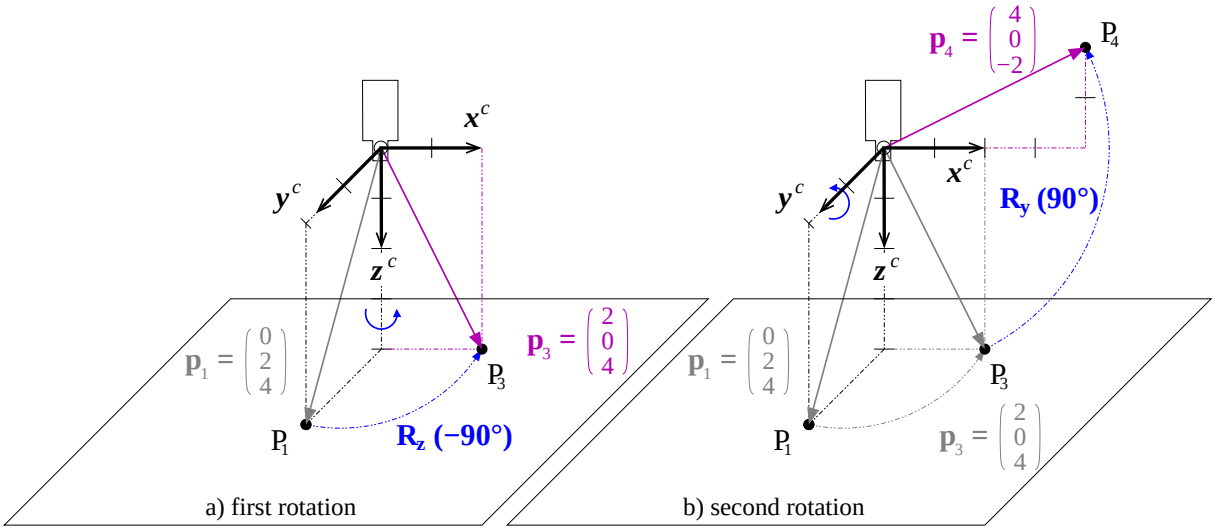


Figure 2.4: Rotate a point: (a) first around the $\mathbf{z}^c$-axis; (b) then around the $\mathbf{y}^c$-axis.

Chain of Rotations

In figure 2.4b, the rotated point is further rotated around the y-axis. Such a chain of rotations can be expressed very elegantly by a chain of rotation matrices:

$$\mathbf{p}_4 = \mathbf{R}_y(\beta) \cdot \mathbf{p}_3 = \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\gamma) \cdot \mathbf{p}_1 \tag{2.7}$$

Note that in contrast to a multiplication of scalars, the multiplication of matrices is not commutative, i.e., if you change the sequence of the rotation matrices, you get a different result.

Rotation of Coordinate Systems

In contrast to points, coordinate systems have an orientation relative to other coordinates systems. This orientation changes when the coordinate system is rotated. For example, in figure 2.5a the coordinate system c_3 has been rotated around the y-axis of the coordinate system c_1, resulting in a different orientation of the camera. Note that in order to rotate a coordinate system in your mind's eye, it may help to image the points of the axis vectors being rotated.

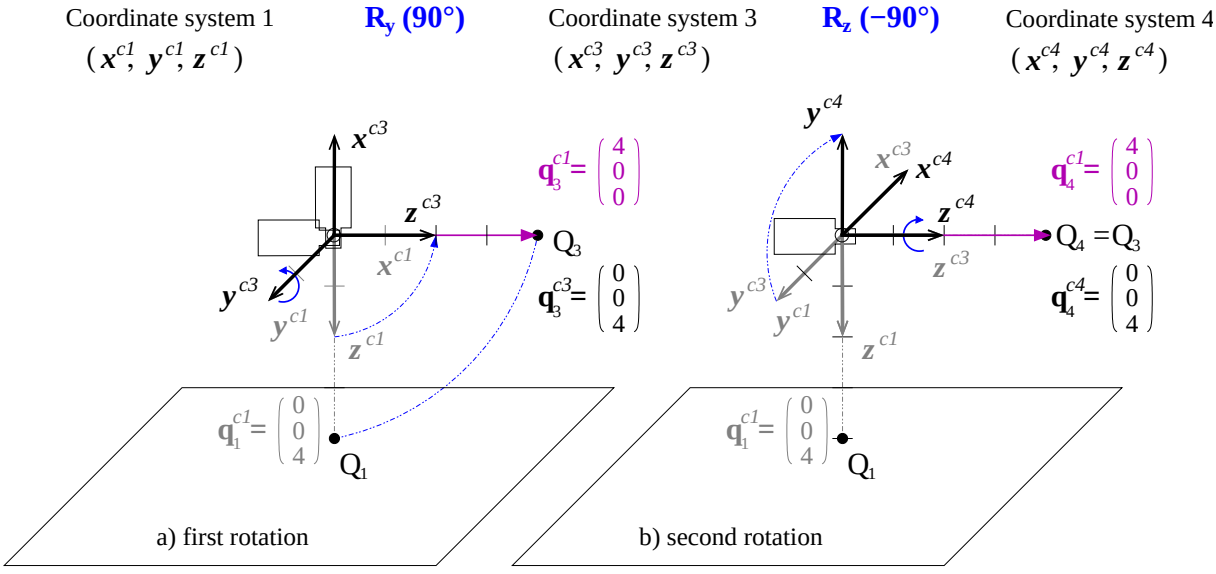


Figure 2.5: Rotate coordinate system: (a) first around the $\mathbf{y}^{c1}$-axis; (b) then around the $\mathbf{z}^{c3}$-axis.

Just like the position of a coordinate system can be expressed directly by the translation vector (see equation 2.2 on page 14), the orientation is contained in the rotation matrix: The columns of the rotation matrix correspond to the axis vectors of the rotated coordinate system in coordinates of the original one:

$$\mathbf{R} = \begin{bmatrix} \mathbf{x}_{c_3}^{c_1} & \mathbf{y}_{c_3}^{c_1} & \mathbf{z}_{c_3}^{c_1} \end{bmatrix} \tag{2.8}$$

For example, the axis vectors of the coordinate system c_3 in figure 2.5a can be determined from the corresponding rotation matrix $\mathbf{R}_y(90°)$ as shown in the following equation; you can easily check the result in the figure.

$$\mathbf{R}_y(90°) = \begin{bmatrix} \cos(90°) & 0 & \sin(90°) \\ 0 & 1 & 0 \\ -\sin(90°) & 0 & \cos(90°) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

$$\Rightarrow \quad \mathbf{x}_{c_3}^{c_1} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \quad \mathbf{y}_{c_3}^{c_1} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \mathbf{z}_{c_3}^{c_1} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Coordinate Transformations

Like in the case of translation, to transform point coordinates from a rotated coordinate system c_3 into the original coordinate system c_1, you apply the same transformation to the points that was applied to the coordinate system c_3, i.e., you multiply the point coordinates with the rotation matrix used to rotate the coordinate system c_1 into c_3:

$$\mathbf{q}_3^{c_1} = {}^{c_1}\mathbf{R}_{c_3} \cdot \mathbf{q}_3^{c_3} \tag{2.9}$$

This is depicted in figure 2.5 also for a chain of rotations, which corresponds to the following equation:

$$\mathbf{q}_4^{c_1} = {}^{c_1}\mathbf{R}_{c_3} \cdot {}^{c_3}\mathbf{R}_{c_4} \cdot \mathbf{q}_4^{c_4} = \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\gamma) \cdot \mathbf{q}_4^{c_4} = {}^{c_1}\mathbf{R}_{c_4} \cdot \mathbf{q}_4^{c_4} \tag{2.10}$$

In Which Sequence and Around Which Axes are Rotations Performed?

If you compare the chains of rotations in figure 2.4 and figure 2.5 and the corresponding equations 2.7 and 2.10, you will note that two different sequences of rotations are described by the same chain of rotation matrices: In figure 2.4, the point was rotated *first* around the *z-axis* and *then* around the *y-axis*, whereas in figure 2.5 the coordinate system is rotated *first* around the *y-axis* and *then* around the *z-axis*. Yet, both are described by the chain $\mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\gamma)$!

The solution to this seemingly paradox situation is that in the two examples the chain of rotation matrices can be "read" in different directions: In figure 2.4 it is read from the right to left, and in figure 2.5 from left to the right.

However, there still must be a difference between the two sequences because, as we already mentioned, the multiplication of rotation matrices is not commutative. This difference lies in the second question in the title, i.e., around which axes the rotations are performed.

Let's start with the second rotation of the coordinate system in figure 2.5b. Here, there are two possible sets of axes to rotate around: those of the "old" coordinate system c_1 and those of the already rotated, "new" coordinate system c_3. In the example, the second rotation is performed around the "new" z-axis.

In contrast, when rotating points as in figure 2.4, there is only one set of axes around which to rotate: those of the "old" coordinate system.

From this, we derive the following rules:

- When reading a chain from the **left to right**, rotations are performed around the **"new"** axes.
- When reading a chain from the **right to left**, rotations are performed around the **"old"** axes.

As already remarked, point rotation chains are always read from right to left. In the case of coordinate systems, you have the choice how to read a rotation chain. In most cases, however, it is more intuitive to read them from left to right.

Figure 2.6 shows that the two reading directions really yield the same result.

Summary

- Points are rotated by multiplying their coordinate vector with a rotation matrix.
- If you rotate a coordinate system, the rotation matrix describes its resulting orientation: The column vectors of the matrix correspond to the axis vectors of the rotated coordinate system in coordinates of the original one.
- To transform point coordinates from a rotated coordinate system c_3 into the original coordinate system c_1, you apply the same transformation to the points that was applied to the coordinate system, i.e., you multiply them with the rotation matrix that was used to rotate the coordinate system c_1 into c_3.
- Multiple rotations are described by a chain of rotation matrices, which can be read in two directions. When read from left to right, rotations are performed around the "new" axes; when read from right to left, the rotations are performed around the "old" axes.

2.1.3 Rigid Transformations using Homogeneous Transformation Matrices

Rigid Transformation of Points

If you combine translation and rotation, you get a so-called rigid transformation. For example, in figure 2.7, the translation and rotation of the point from figures 2.2 and 2.4 are combined. Such a transformation is described as follows:

$$\mathbf{p}_5 = \mathbf{R} \cdot \mathbf{p}_1 + \mathbf{t} \tag{2.11}$$

For multiple transformations, such equations quickly become confusing, as the following example with two transformations shows:

$$\mathbf{p}_6 = \mathbf{R}_a \cdot (\mathbf{R}_b \cdot \mathbf{p}_1 + \mathbf{t}_b) + \mathbf{t}_a = \mathbf{R}_a \cdot \mathbf{R}_b \cdot \mathbf{p}_1 + \mathbf{R}_a \cdot \mathbf{t}_b + \mathbf{t}_a \tag{2.12}$$

An elegant alternative is to use so-called *homogeneous transformation matrices* and the corresponding homogeneous vectors. A homogeneous transformation matrix $\mathbf{H}$ contains both the rotation matrix and the translation vector. For example, the rigid transformation from equation 2.11 can be rewritten as follows:

$$\begin{pmatrix} \mathbf{p}_5 \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\,0\,0 & 1 \end{bmatrix} \cdot \begin{pmatrix} \mathbf{p}_1 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} \cdot \mathbf{p}_1 + \mathbf{t} \\ 1 \end{pmatrix} = \mathbf{H} \cdot \begin{pmatrix} \mathbf{p}_1 \\ 1 \end{pmatrix} \tag{2.13}$$

The usefulness of this notation becomes apparent when dealing with sequences of rigid transformations, which can be expressed as chains of homogeneous transformation matrices, similarly to the rotation chains:

$$\mathbf{H}_1 \cdot \mathbf{H}_2 = \begin{bmatrix} \mathbf{R}_a & \mathbf{t}_a \\ 0\,0\,0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_b & \mathbf{t}_b \\ 0\,0\,0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_a \cdot \mathbf{R}_b & \mathbf{R}_a \cdot \mathbf{t}_b + \mathbf{t}_a \\ 0\,0\,0 & 1 \end{bmatrix} \tag{2.14}$$

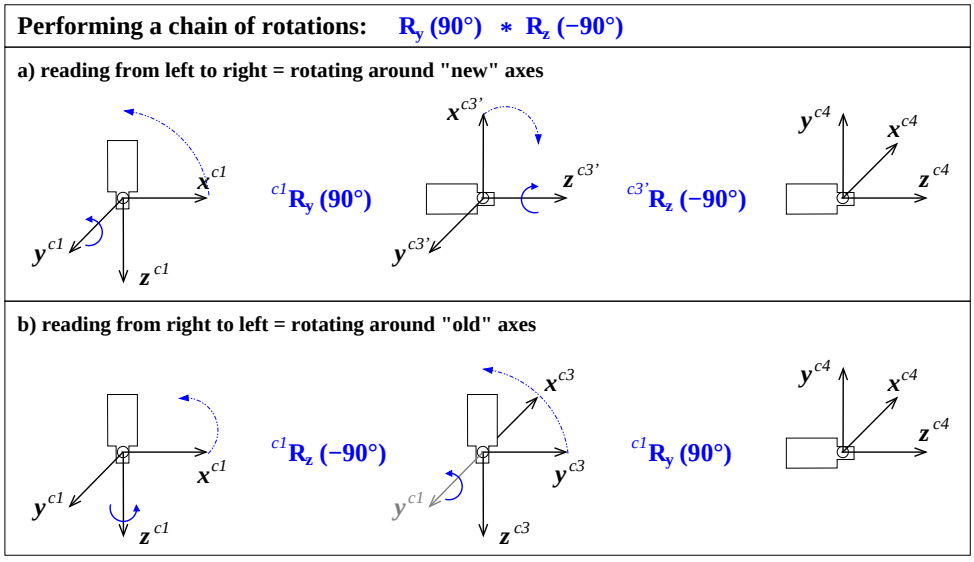


Figure 2.6: Performing a chain of rotations (a) from left to the right, or (b) from right to left.

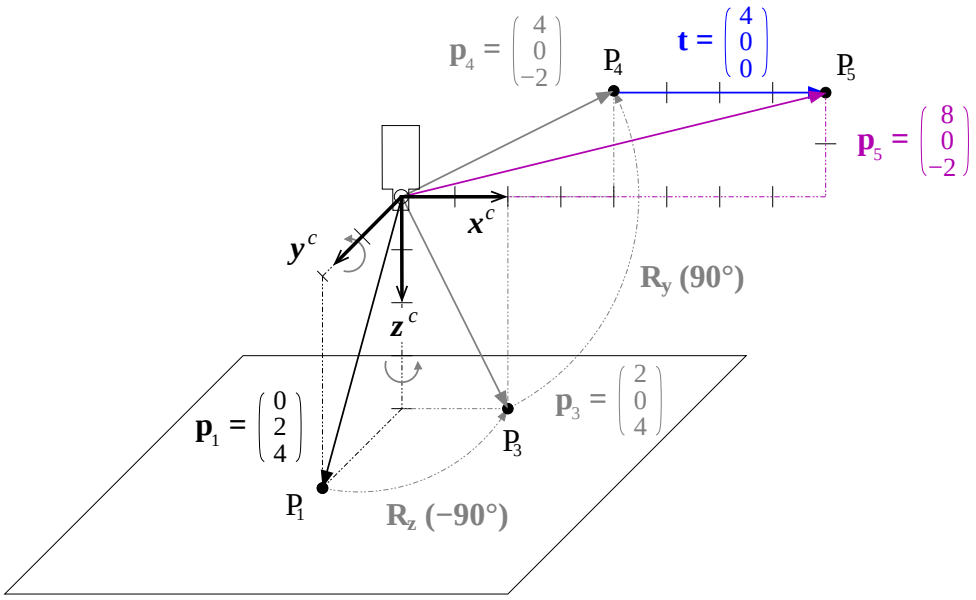


Figure 2.7: Combining the translation from figure 2.2 on page 14 and the rotation of figure 2.4 on page 16 to form a rigid transformation.

As explained for chains of rotations, chains of rigid transformation can be read in two directions. When reading from left to right, the transformations are performed around the "new" axes, when read from right to left around the "old" axes.

In fact, a rigid transformation is already a chain, since it consists of a translation and a rotation:

$$\mathbf{H} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\,0\,0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \\ 0 & 1 & 0 & \mathbf{t} \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R} & & 0 \\ & & & 0 \\ 0\,0\,0 & & & 1 \end{bmatrix} = \mathbf{H}(\mathbf{t}) \cdot \mathbf{H}(\mathbf{R}) \tag{2.15}$$

If the rotation is composed of multiple rotations around axes as in figure 2.7, the individual rotations can also be written as homogeneous transformation matrices:

$$\mathbf{H} = \begin{bmatrix} \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\gamma) & \mathbf{t} \\ 0\,0\,0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \\ 0 & 1 & 0 & \mathbf{t} \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R}_y(\beta) & & 0 \\ & & & 0 \\ 0\,0\,0 & & & 1 \end{bmatrix} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R}_z(\gamma) & & 0 \\ & & & 0 \\ 0\,0\,0 & & & 1 \end{bmatrix}$$

Reading this chain from right to left, you can follow the transformation of the point in figure 2.7: First, it is rotated around the z-axis, then around the ("old") y-axis, and finally it is translated.

Rigid Transformation of Coordinate Systems

Rigid transformations of coordinate systems work along the same lines as described for a separate translation and rotation. This means that the homogeneous transformation matrix ${}^{c_1}\mathbf{H}_{c_5}$ describes the transformation of the coordinate system c_1 into the coordinate system c_5. At the same time, it describes the position and orientation of coordinate system c_5 relative to coordinate system c_1: Its column vectors contain the coordinates of the axis vectors and the origin.

$${}^{c_1}\mathbf{H}_{c_5} = \begin{bmatrix} \mathbf{x}_{c_5}^{c_1} & \mathbf{y}_{c_5}^{c_1} & \mathbf{z}_{c_5}^{c_1} & \mathbf{o}_{c_5}^{c_1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

As already noted for rotations, chains of rigid transformations of coordinate systems are typically read from left to right. Thus, the chain above can be read as first translating the coordinate system, then rotating it around its "new" y-axis, and finally rotating it around its "newest" z-axis.

Coordinate Transformations

As described for the separate translation and the rotation, to transform point coordinates from a rigidly transformed coordinate system c_5 into the original coordinate system c_1, you apply the same transformation to the points that was applied to the coordinate system c_5, i.e., you multiply the point coordinates with the homogeneous transformation matrix:

$$\begin{pmatrix} \mathbf{p}_5^{c_1} \\ 1 \end{pmatrix} = {}^{c_1}\mathbf{H}_{c_5} \cdot \begin{pmatrix} \mathbf{p}_5^{c_5} \\ 1 \end{pmatrix} \tag{2.17}$$

Typically, you leave out the homogeneous vectors if there is no danger of confusion and simply write:

$$\mathbf{p}_5^{c_1} = {}^{c_1}\mathbf{H}_{c_5} \cdot \mathbf{p}_5^{c_5} \tag{2.18}$$

Summary

- Rigid transformations consist of a rotation and a translation. They are described very elegantly by homogeneous transformation matrices, which contain both the rotation matrix and the translation vector.
- Points are transformed by multiplying their coordinate vector with the homogeneous transformation matrix.
- If you transform a coordinate system, the homogeneous transformation matrix describes the coordinate system's resulting position and orientation: The column vectors of the matrix correspond to the axis vectors and the origin of the coordinate system in coordinates of the original one. Thus, you could say that a homogeneous transformation matrix "is" the position and orientation of a coordinate system.
- To transform point coordinates from a rigidly transformed coordinate system c_5 into the original coordinate system c_1, you apply the same transformation to the points that was applied to the coordinate system, i.e., you multiply them with the homogeneous transformation matrix that was used to transform the coordinate system c_1 into c_5.
- Multiple rigid transformations are described by a chain of transformation matrices, which can be read in two directions. When read from left to the right, rotations are performed around the "new" axes; when read from the right to left, the transformations are performed around the "old" axes.

HALCON Operators

As we already anticipated at the beginning of section 2.1 on page 13, homogeneous transformation matrices are the answer to all our questions regarding the use of 3D coordinates. Because of this, they form the basis for HALCON's operators for 3D transformations. Below, you find a brief overview of the relevant operators. For more details follow the links into the Reference Manual.

- `hom_mat3d_identity` creates the identity transformation
- `hom_mat3d_translate` translates along the "old" axes: $\mathbf{H}_2 = \mathbf{H}(\mathbf{t}) \cdot \mathbf{H}_1$
- `hom_mat3d_translate_local` translates along the "new" axes: $\mathbf{H}_2 = \mathbf{H}_1 \cdot \mathbf{H}(\mathbf{t})$
- `hom_mat3d_rotate` rotates around the "old" axes: $\mathbf{H}_2 = \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}_1$
- `hom_mat3d_rotate_local` rotates around the "new" axes: $\mathbf{H}_2 = \mathbf{H}_1 \cdot \mathbf{H}(\mathbf{R})$
- `hom_mat3d_compose` multiplies two transformation matrices: $\mathbf{H}_3 = \mathbf{H}_1 \cdot \mathbf{H}_2$
- `hom_mat3d_invert` inverts a transformation matrix: $\mathbf{H}_2 = \mathbf{H}_1^{-1}$
- `affine_trans_point_3d` transforms a point using a transformation matrix: $\mathbf{p}_2 = \mathbf{H}_0 \cdot \mathbf{p}_1$

2.1.4 Transformations using 3D Poses

Homogeneous transformation matrices are a very elegant means of describing transformations, but their content, i.e., the elements of the matrix, are often difficult to read, especially the rotation part. This problem is alleviated by using so-called *3D poses*.

A 3D pose is nothing more than an easier-to-understand representation of a rigid transformation: Instead of the 12 elements of the homogeneous transformation matrix, a pose describes the rigid transformation with 6 parameters, 3 for the rotation and 3 for the translation: (`TransX`, `TransY`, `TransZ`, `RotX`, `RotY`, `RotZ`). The main principle

behind poses is that even a rotation around an arbitrary axis can always be represented by a sequence of three rotations around the axes of a coordinate system.

In HALCON, you create 3D poses with `create_pose`; to transform between poses and homogeneous matrices you can use `hom_mat3d_to_pose` and `pose_to_hom_mat3d`.

2.1.4.1 Sequence of Rotations

However, there is more than one way to represent an arbitrary rotation by three parameters. This is reflected by the HALCON operator `create_pose`, which lets you choose between different pose types with the parameter `OrderOfRotation`. If you pass the value *'gba'*, the rotation is described by the following chain of rotations:

$$\mathbf{R}_{gba} = \mathbf{R}_x(\texttt{RotX}) \cdot \mathbf{R}_y(\texttt{RotY}) \cdot \mathbf{R}_z(\texttt{RotZ}) \tag{2.19}$$

You may also choose the inverse order by passing the value *'abg'*:

$$\mathbf{R}_{abg} = \mathbf{R}_z(\texttt{RotZ}) \cdot \mathbf{R}_y(\texttt{RotY}) \cdot \mathbf{R}_x(\texttt{RotX}) \tag{2.20}$$

For example, the transformation discussed in the previous sections can be represented by the homogeneous transformation matrix

$$\mathbf{H} = \begin{bmatrix} \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\gamma) & \mathbf{t} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\beta \cdot \cos\gamma & -\cos\beta \cdot \sin\gamma & \sin\beta & x_t \\ \sin\gamma & \cos\gamma & 0 & y_t \\ -\sin\beta \cdot \cos\gamma & \sin\beta \cdot \sin\gamma & \cos\beta & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The corresponding pose with the rotation order *'gba'* is much easier to read:

$$(\texttt{TransX} = x_t, \texttt{TransY} = y_t, \texttt{TransZ} = z_t, \texttt{RotX} = 0, \texttt{RotY} = 90°, \texttt{RotZ} = -90°)$$

If you look closely at figure 2.5 on page 16, you can see that the rotation can also be described by the sequence $\mathbf{R}_z(-90°) \cdot \mathbf{R}_x(-90°)$. Thus, the transformation can also be described by the following pose with the rotation order *'abg'*:

$$(\texttt{TransX} = x_t, \texttt{TransY} = y_t, \texttt{TransZ} = z_t, \texttt{RotX} = -90°, \texttt{RotY} = 0, \texttt{RotZ} = -90°)$$

HALCON Operators

Below, the relevant HALCON operators for dealing with 3D poses are briefly described. For more details follow the links into the Reference Manual.

- `create_pose` creates a pose
- `hom_mat3d_to_pose` converts a homogeneous transformation matrix into a pose
- `pose_to_hom_mat3d` converts a pose into a homogeneous transformation matrix
- `convert_pose_type` changes the pose type
- `write_pose` writes a pose into a file
- `read_pose` reads a pose from a file
- `set_origin_pose` translates a pose along its "new" axes
- `pose_invert` inverts a pose
- `pose_compose` multiplies two poses, i.e., it sequentially applies two transformations (poses)

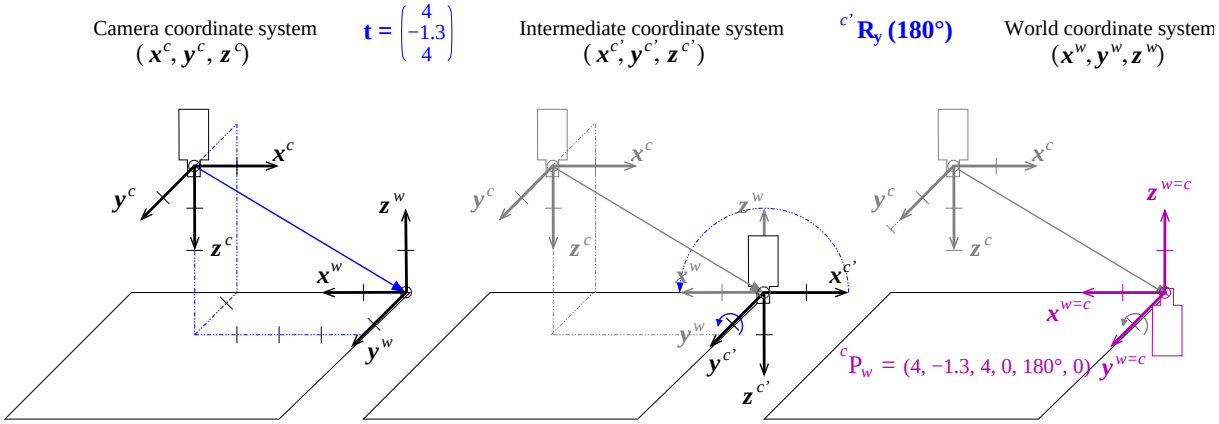


Figure 2.8: Determining the pose of the world coordinate system in camera coordinates.

2.1.4.2 How to Determine the Pose of a Coordinate System

The previous sections showed how to describe known transformations using translation vectors, rotation matrices, homogeneous transformation matrices, or poses. Sometimes, however, there is another task: How to describe the position and orientation of a coordinate system with a pose.

Figure 2.8 shows how to proceed for a rather simple example. The task is to determine the pose of the world coordinate system from figure 2.1 on page 13 relative to the camera coordinate system.

In such a case, we recommend to build up the rigid transformation from individual translations and rotations from left to right. Thus, in figure 2.8 the camera coordinate system is first translated such that its origin coincides with that of the world coordinate system. Now, the y-axes of the two coordinate systems coincide; after rotating the (translated) camera coordinate system around its (new) y-axis by 180°, it has the correct orientation.

2.1.5 Transformations using Dual Quaternions and Plücker Coordinates

2.1.5.1 Dual Quaternions

In contrast to unit quaternions, which are able to represent 3D rotations, a unit dual quaternion is able to represent a full 3D rigid transformation, i.e., a 3D rotation and a 3D translation. Hence, unit dual quaternions are an alternative representation to 3D poses and 3D homogeneous transformation matrices for 3D rigid transformations. In comparison to transformation matrices with 12 elements, dual quaternions with 8 elements are a more compact representation. Similar to transformation matrices, dual quaternions can be combined easily to concatenate multiple transformations. Furthermore, they allow a smooth interpolation between two 3D rigid transformations and an efficient transformation of 3D lines.

A dual quaternion $\hat{q} = q_r + \epsilon * q_d$ consists of the two quaternions q_r and q_d, where q_r is the real part, q_d is the dual part, and ϵ is the dual unit number ($\epsilon^2 = 0$). Each quaternion $q = w + ix + jy + kz$ consists of the scalar part w and the vector part $v = (x, y, z)$, where $(1, i, j, k)$ are the basis elements of the quaternion vector space.

In HALCON, a dual quaternion is represented by a tuple with eight values $[w_r, x_r, y_r, z_r, w_d, x_d, y_d, z_d]$, where w_r and $v_r = (x_r, y_r, z_r)$ are the scalar and the vector part of the real part and w_d and $v_d = (x_d, y_d, z_d)$ are the scalar and the vector part of the dual part.

Each 3D rigid transformation can be represented as a screw (see figure 2.9 and figure 2.10):

The parameters that fully describe the screw are:

- screw angle θ
- screw translation d
- direction $\mathbf{L} = (\mathbf{L}^x, \mathbf{L}^y, \mathbf{L}^z)^T$ of the screw axis with $||L|| = 1$
- moment $\mathbf{M} = (\mathbf{M}^x, \mathbf{M}^y, \mathbf{M}^z)^T$ of the screw axis with $L * M = 0$

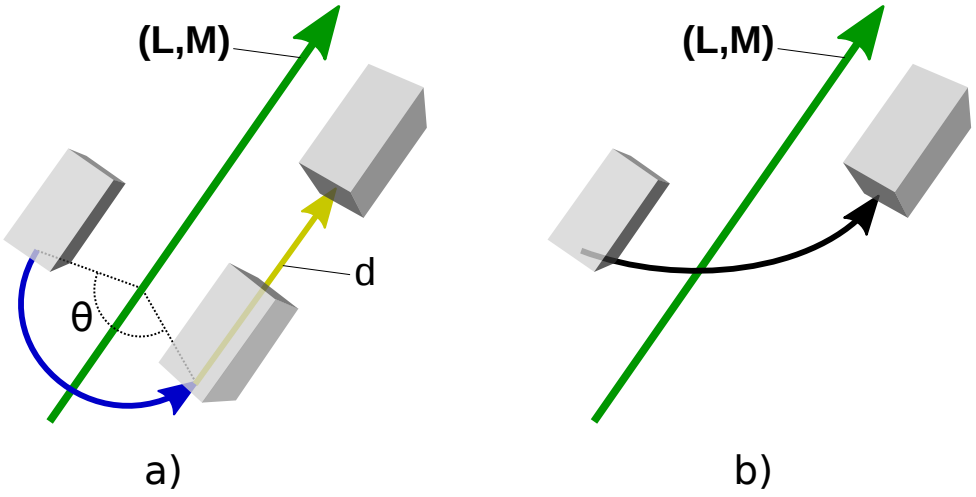


Figure 2.9: a) A 3D rigid transformation defined by a rotation and a translation... b) can be represented as a screw.

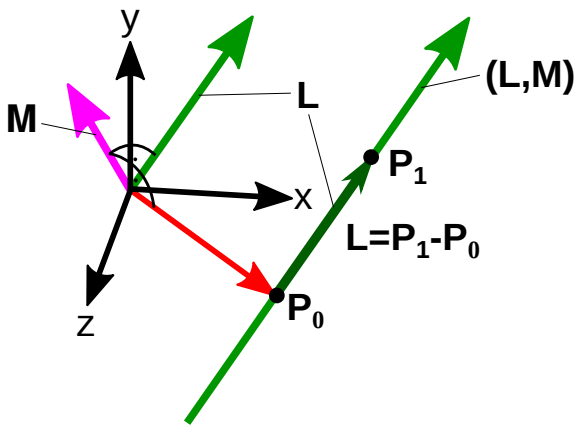


Figure 2.10: Moment M of the screw axis.

A screw is composed of a rotation about the screw axis given by L and M by the angle θ and a translation by d along this axis. The position of the screw axis is defined by its moment with respect to the origin of the coordinate system. M is a vector that is perpendicular to the direction of the screw axis L and perpendicular to a vector from the origin to a point P_0 on the screw axis. It is calculated by the vector product $M = P_0 \times L$.

Hence, M is the normal vector of the plane that is spanned by the screw axis and the origin. Note that $P_0 = L \times M$ is the point on the screw axis (L, M) with the shortest distance to the origin of the coordinate system. The elements of a unit dual quaternion are related to the screw parameters of the 3D rigid transformation as:

$$\hat{q} = \begin{pmatrix} cos\frac{\theta}{2} \\ Lsin\frac{\theta}{2} \end{pmatrix} + \epsilon \begin{pmatrix} \frac{-d}{2} sin\frac{\theta}{2} \\ Msin\frac{\theta}{2} + L\frac{d}{2}cos\frac{\theta}{2} \end{pmatrix} \tag{2.21}$$

Note that $\hat{q}$ and $-\hat{q}$ represent the same 3D rigid transformation. Further note that the inverse of a unit dual quaternion is its conjugate, i.e., $\hat{q}^{-1} = \bar{\hat{q}}$.

The conjugation of a dual quaternion $\hat{q} = q_r + \varepsilon q_d$ is given by $\bar{\hat{q}} = \bar{q}_r + \varepsilon \bar{q}_d$, where $\bar{q}_r$ and $\bar{q}_d$ are the conjugations of the quaternions q_r and q_d.

The conjugation of a quaternion $q = x_0 + x_1 i + x_2 j + x_3 k$ is given by $\bar{q} = x_0 - x_1 i - x_2 j - x_3 k$.

HALCON Operators

- `pose_to_dual_quat` converts a 3D pose to a unit dual quaternion

- dual_quat_to_pose converts a dual quaternion to a 3D pose
- dual_quat_compose multiplies two dual quaternions
- dual_quat_interpolate interpolates between two dual quaternions
- dual_quat_to_screw converts a unit dual quaternion into a screw
- screw_to_dual_quat converts a screw into a dual quaternion
- dual_quat_to_hom_mat3d converts a unit dual quaternion into a homogeneous transformation matrix
- dual_quat_trans_line_3d transforms a 3D line with a unit dual quaternion
- dual_quat_trans_point_3d transforms a 3D point with a unit dual quaternion
- dual_quat_conjugate conjugates a dual quaternion
- dual_quat_normalize normalizes a dual quaternion
- serialize_dual_quat serializes a dual quaternion
- deserialize_dual_quat deserializes a serialized dual quaternion

2.1.5.2 3D lines and Plücker Coordinates

Plücker coordinates are a very useful representation of lines in 3D space.

A line in 3D space, as shown in figure 2.11, can be described by two points $\mathbf{P}_0$ and $\mathbf{P}_1$. However, this usage of arbitrary points comes with the disadvantage that the same line can be described in multiple ways.
Another approach is to take the unit line direction $\mathbf{L}$ and the line moment $\mathbf{M}$. $\mathbf{M}$ is a vector that is perpendicular to the plane spanned by the origin, a point on the line, and the line direction $\mathbf{L}$. $\mathbf{L}$ and $\mathbf{M}$ define the line independent of the arbitrary line points. The six parameters of $\mathbf{L}$ and $\mathbf{M}$ are called the Plücker coordinates of the line.

From its definition, it holds that $\|\mathbf{L}\| = 1$ and $\mathbf{L} \cdot \mathbf{M} = 0$, where $\cdot$ denotes the dot product of two vectors.

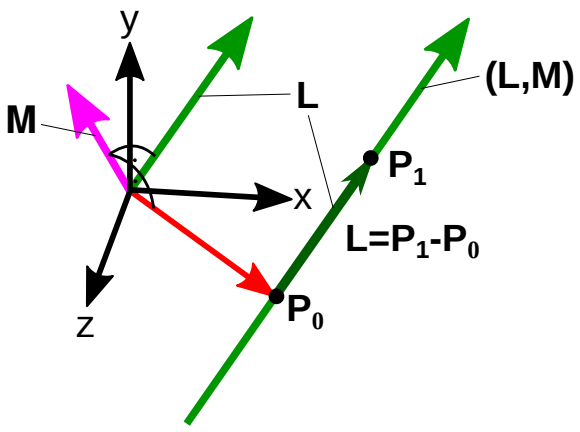


Figure 2.11: A line in 3D space and its components.

Using Plücker coordinates, it is very simple and efficient to compute the distance D of a point $\mathbf{P}$ to a line: $D = \|\mathbf{P} \times \mathbf{L} - \mathbf{M}\|$.

HALCON Operators

- distance_point_pluecker_line Calculate the distance between a 3D point and a 3D line given by Plücker coordinates.
- pluecker_line_to_point_direction Convert a 3D line given by Plücker coordinates to a 3D line given by a point and a direction.

- `pluecker_line_to_points` Convert a 3D line given by Plücker coordinates to a 3D line given by two points.
- `point_direction_to_pluecker_line` Convert a 3D line given by a point and a direction to Plücker coordinates.
- `points_to_pluecker_line` Convert a 3D line given by two points to Plücker coordinates.
- `point_pluecker_line_to_hom_mat3d` Approximate a 3D affine transformation from 3D point-to-line correspondences.

2.1.5.3 Dual Quaternions and Plücker Coordinates

Plücker lines can be transformed efficiently with rigid transformations using dual quaternions.

Lines in 3D can be represented by dual unit vectors. A dual unit vector can be interpreted as a dual quaternion with 0 scalar part. The 3D rigid transformation that is represented by a unit dual quaternion is easily related to the corresponding screw around a screw axis. As described in section 2.1.5.1 on page 22, the screw axis is defined by its direction $\mathbf{L}$ with $\|\mathbf{L}\| = 1$ and its moment $\mathbf{M}$. But $\mathbf{L}$ and $\mathbf{M}$ are exactly the Plücker coordinates introduced in section 2.1.5.2 on page 24.

Consequently, a line $\hat{l}$ can be represented by a dual quaternion with 0 scalar part by

$$\hat{l} = l_r + \varepsilon l_d = \begin{pmatrix} 0 \\ L_x \\ L_y \\ L_z \end{pmatrix} + \varepsilon \begin{pmatrix} 0 \\ M_x \\ M_y \\ M_z \end{pmatrix}.$$

The line $\hat{l}$ can be transformed by the 3D rigid transformation that is represented by the unit dual quaternion $\hat{q}$ very conveniently:

$$\hat{k} = \hat{q}\hat{l}\bar{\hat{q}}$$

The resulting dual quaternion $\hat{k}$ also has 0 scalar part and directly contains the direction and the moment of the transformed line in its vector part.

2.2 Camera Model and Parameters

If you want to derive *accurate* world coordinates from your imagery, you first have to calibrate your camera. To calibrate a camera, a model for the mapping of the 3D points of the world to the 2D image generated by the camera, lens, and frame grabber is necessary.

HALCON supports the calibration of two different kinds of cameras: area scan cameras and line scan cameras. While area scan cameras acquire the image in one step, line scan cameras generate the image line by line (see Solution Guide II-A, section 6.6 on page 39). Therefore, the line scan camera must move relative to the object during the acquisition process.

Two different types of lenses are relevant for machine vision tasks. The first type of lens effects a perspective projection of the world coordinates into the image, just like the human eye does. With this type of lens, objects become smaller in the image the farther they are away from the camera. This combination of camera and lens is called a *pinhole camera model* because the perspective projection can also be achieved if a small hole is drilled in a thin planar object and this plane is held parallel in front of another plane (the image plane).

The second type of lens that is relevant for machine vision is called a telecentric lens. Its major difference is that it effects a parallel projection of the world coordinates onto the image plane (for a certain range of distances of the object from the camera). This means that objects have the same size in the image independent of their distance to the camera. This combination of camera and lens is called a *telecentric camera model*.

The distinct types of camera use different parameters. An overview of the units used in HALCON for the different camera parameters is given in chapter "Calibration ▷ Multi-View".

In the following, after a short overview, first the camera model for area scan cameras is described in detail, then, the camera model for line scan cameras is explained.

2.2.1 Map 3D World Points to Pixel Coordinates

To transform a 3D point $\mathbf{p}^w = (\mathbf{x}^w, \mathbf{y}^w, \mathbf{z}^w)^T$, which is given in world coordinates, into a 2D point $\mathbf{q}^i = (\mathbf{r}, \mathbf{c})^T$, which is given in pixel coordinates, a chain of transformations is needed:

$$\mathbf{p}^w \rightarrow \mathbf{p}^c \rightarrow \mathbf{q}^c \rightarrow \tilde{\mathbf{q}}^c \left[\rightarrow \mathbf{q}^t\right] \rightarrow \mathbf{q}^i \tag{2.22}$$

First, $\mathbf{p}^w$ is transformed into the camera coordinate system into $\mathbf{p}^c$. Then, $\mathbf{p}^c$ is projected into the image plane, i.e., converted to the 2D point $\mathbf{q}^c$, still in metric coordinates. Then, lens distortion is applied to $\mathbf{q}^c$, transforming it into the distorted point $\tilde{\mathbf{q}}^c$. If a tilt lens is used, $\tilde{\mathbf{q}}^c$ only lies on a virtual image plane of a system without tilt. This is corrected by projecting $\tilde{\mathbf{q}}^c$ to the point $\mathbf{q}^t$ on the tilted image plane. Finally, the coordinates of the distorted point $\tilde{\mathbf{q}}^c$ (or $\mathbf{q}^t$) are converted to pixel coordinates, which results in the final point $\mathbf{q}^i$.

2.2.2 Area Scan Cameras

Figure 2.12 displays the perspective projection effected by a pinhole camera graphically. The world point P is projected through the optical center of the lens to the point P' in the image plane, which is located at a distance of f (the focal length) behind the optical center. Actually, the term "focal length" is not quite correct and would be appropriate only for an infinite object distance. To simplify matters, in the following always the term "focal length" is used even if the "image distance" is meant. Note that the focal length and thus the focus must not be changed after applying the camera calibration.

Although the image plane in reality lies behind the optical center of the lens, it is easier to pretend that it lies at a distance of f in front of the optical center, as shown in figure 2.13. This causes the image coordinate system to be aligned with the pixel coordinate system (row coordinates increase downward and column coordinates to the right) and simplifies most calculations.

2.2.2.1 Transformation into Camera Coordinates (External Camera Parameters)

With this, we are now ready to describe the projection of objects in 3D world coordinates to the 2D image plane and the corresponding camera parameters. First, we should note that the points P are given in a world coordinate system (WCS). To make the projection into the image plane possible, they need to be transformed into the camera coordinate system (CCS). The CCS is defined so that its x and y axes are parallel to the column and row axes of the image, respectively, and the z axis is perpendicular to the image plane.

The transformation from the WCS to the CCS is a rigid transformation, which can be expressed by a pose or, equivalently, by the homogeneous transformation matrix ${}^c\mathbf{H}_w$. Therefore, the camera coordinates $\mathbf{p}^c = (\mathbf{x}^c, \mathbf{y}^c, \mathbf{z}^c)^T$ of point P can be calculated from its world coordinates $\mathbf{p}^w = (\mathbf{x}^w, \mathbf{y}^w, \mathbf{z}^w)^T$ simply by

$$\mathbf{p}^c = {}^c\mathbf{H}_w \cdot \mathbf{p}^w \tag{2.23}$$

The six parameters of this transformation (the three translations t_x, t_y, and t_z and the three rotations α, β, and γ) are called the *external camera parameters* because they determine the position of the camera with respect to the world. In HALCON, they are stored as a pose, i.e., together with a code that describes the order of translation and rotations.

2.2.2.2 Projection

The next step is the projection of the 3D point given in the CCS into the image plane coordinate system (IPCS). For the pinhole camera model, the projection is a perspective projection, which is given by

$$\mathbf{q}^c = \begin{pmatrix} u \\ v \end{pmatrix} = \frac{f}{\mathbf{z}^c} \begin{pmatrix} \mathbf{x}^c \\ \mathbf{y}^c \end{pmatrix} \tag{2.24}$$

For cameras with hypercentric lenses, the following equation holds instead:

$$\mathbf{q}^c = \begin{pmatrix} u \\ v \end{pmatrix} = \frac{-f}{\mathbf{z}^c} \begin{pmatrix} \mathbf{x}^c \\ \mathbf{y}^c \end{pmatrix} \tag{2.25}$$

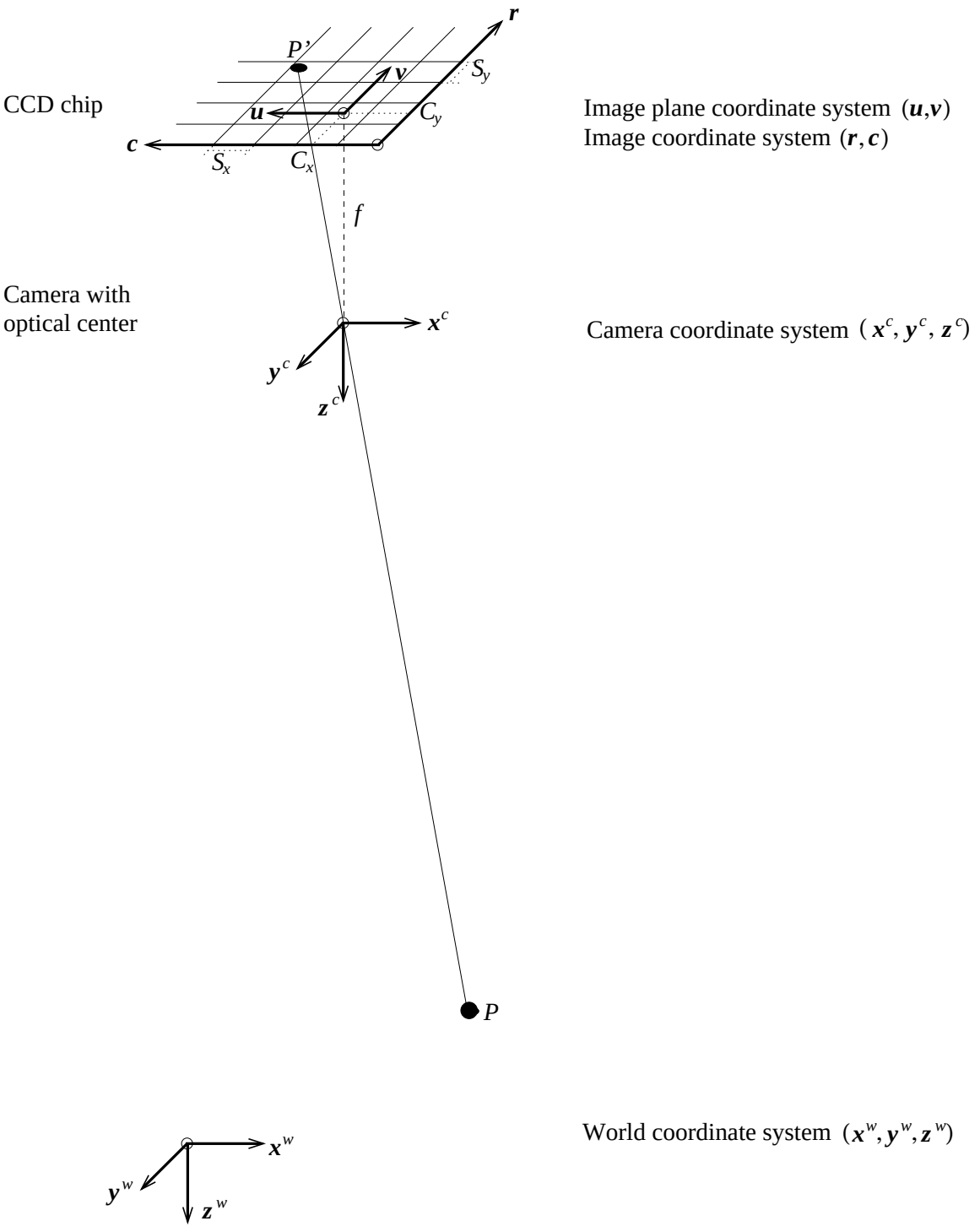


Figure 2.12: Perspective projection by a pinhole camera.

For the telecentric camera model, the projection is a parallel projection, which is given by

$$\mathbf{q}^c = \begin{pmatrix} u \\ v \end{pmatrix} = m \begin{pmatrix} \mathbf{x}^c \\ \mathbf{y}^c \end{pmatrix} \tag{2.26}$$

where m = *magnification*. As can be seen, the distance z of the object to the camera has no influence on the image coordinates.

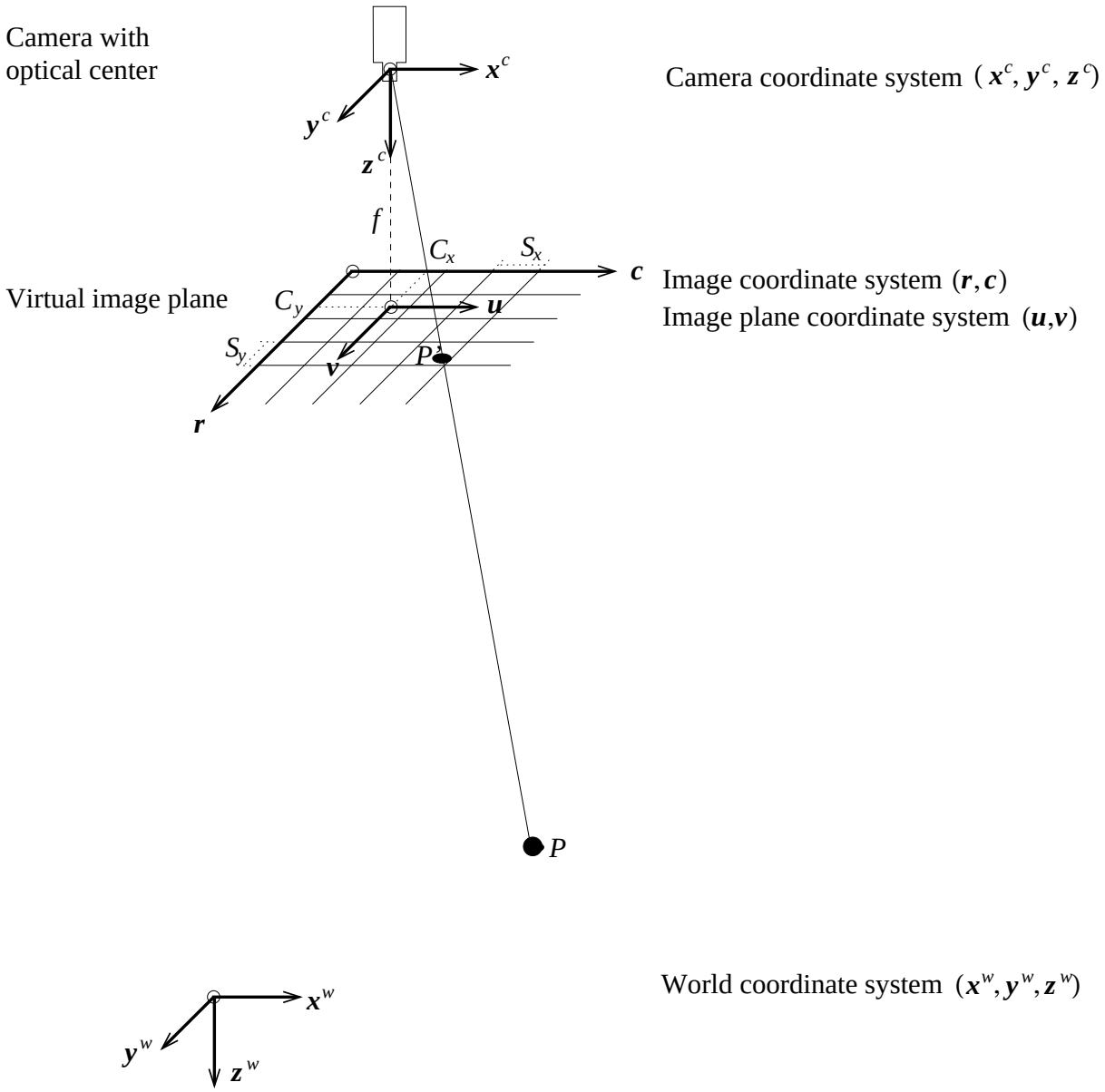


Figure 2.13: Image plane and virtual image plane.

2.2.2.3 Lens Distortion

After the projection into the image plane, the lens distortion modifies the coordinates $(u, v)^T$ of $\mathbf{q}^c$ to $\tilde{\mathbf{q}}^c = (\tilde{u}, \tilde{v})^T$. The effect is illustrated in figure 2.14: If no lens distortion were present, the projected point P' would lie on a straight line from P through the optical center, indicated by the dotted line in figure 2.14. Lens distortions cause the point P' to lie at a different position.

The lens distortion is a transformation that can be modeled in the image plane alone, i.e., 3D information is unnecessary. In HALCON, the distortions can be modeled either by the division model or by the polynomial model.

The **division model** uses one parameter (κ) to model the radial distortions. The following equations transform the distorted image plane coordinates into undistorted image plane coordinates if the division model is used:

$$\begin{aligned} u &= \frac{\tilde{u}}{1+\kappa(\tilde{u}^2+\tilde{v}^2)} \\ v &= \frac{\tilde{v}}{1+\kappa(\tilde{u}^2+\tilde{v}^2)} \end{aligned} \tag{2.27}$$

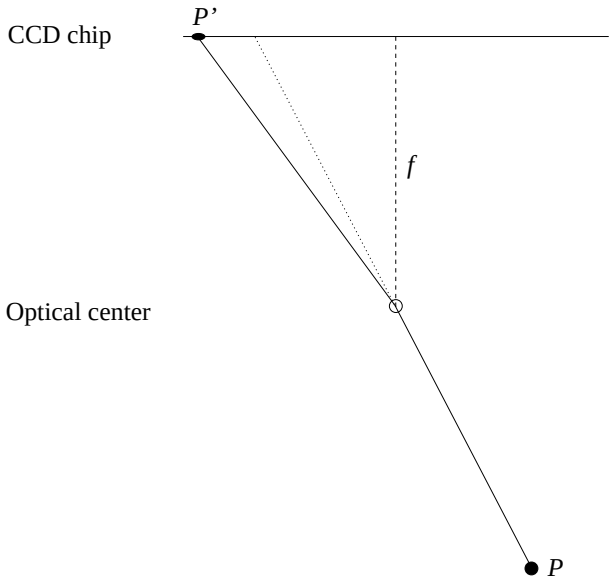


Figure 2.14: Schematic illustration of the effect of the lens distortion.

These equations can be inverted analytically, which leads to the following equations that transform undistorted coordinates into distorted coordinates if the division model is used:

$$\begin{aligned} \tilde{u} &= \frac{2u}{1+\sqrt{1-4\kappa(u^2+v^2)}} \\ \tilde{v} &= \frac{2v}{1+\sqrt{1-4\kappa(u^2+v^2)}} \end{aligned} \tag{2.28}$$

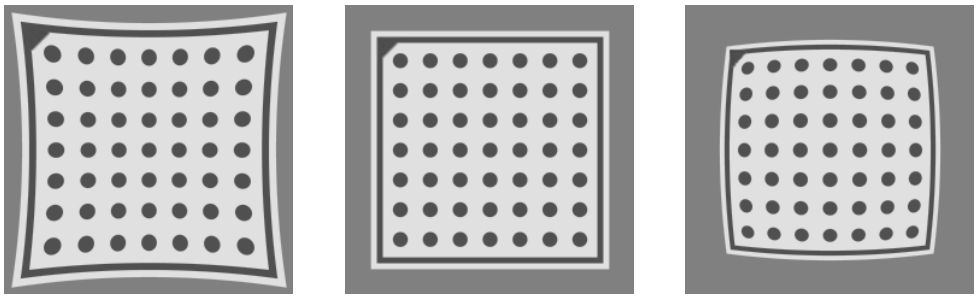


Figure 2.15: Effect of radial distortions modeled with the division model with $\kappa > 0$ (left), $\kappa = 0$ (middle), and $\kappa < 0$ (right).

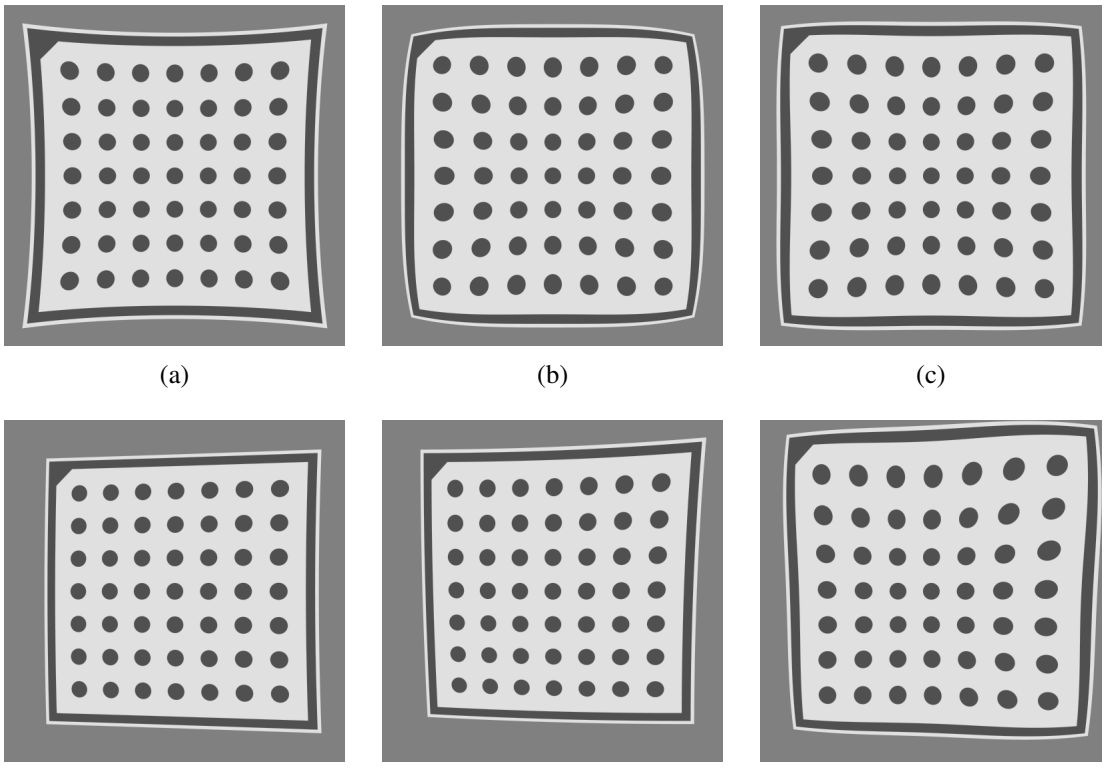
The parameter κ models the magnitude of the radial distortions. If κ is negative, the distortion is barrel-shaped, while for positive κ it is pincushion-shaped (see figure 2.15).

The **polynomial model** uses three parameters (K_1, K_2, K_3) to model the radial distortions, and two parameters (P_1, P_2) to model the decentering distortions. The following equations transform the distorted image plane coordinates into undistorted image plane coordinates if the polynomial model is used:

$$\begin{aligned} u &= \tilde{u} + \tilde{u}(K_1 r^2 + K_2 r^4 + K_3 r^6) + P_1(r^2 + 2\tilde{u}^2) + 2P_2\tilde{u}\tilde{v} \\ v &= \tilde{v} + \tilde{v}(K_1 r^2 + K_2 r^4 + K_3 r^6) + 2P_1\tilde{u}\tilde{v} + P_2(r^2 + 2\tilde{v}^2) \end{aligned} \tag{2.29}$$

with $r = \sqrt{\tilde{u}^2 + \tilde{v}^2}$. These equations cannot be inverted analytically. Therefore, distorted image plane coordinates must be calculated from undistorted image plane coordinates numerically.

Some examples for the kind of distortions that can be modeled with the polynomial model are shown in figure 2.16.



(a) (b) (c)

(d) (e) (f)

	K_1	K_2	K_3	P_1	P_2
(a)	< 0	0	0	0	0
(b)	< 0	> 0	0	0	0
(c)	< 0	> 0	< 0	0	0
(d)	< 0	0	0	< 0	0
(e)	< 0	0	0	< 0	> 0
(f)	< 0	> 0	< 0	< 0	> 0

Figure 2.16: Effect of distortions modeled with the polynomial model with different values for the parameters K_1, K_2, K_3, P_1, and P_2.

2.2.2.4 Tilt Lenses

If the lens is a **tilt lens**, the tilt of the lens with respect to the image plane is described by two parameters: The rotation angle ρ ($0° \leq \rho < 360°$), which describes the direction of the tilt axis, and the tilt angle $\tau (0° \leq \tau < 90°)$, by which the sensor plane is tilted with respect to the optical axis (see figure 2.17).

For tilt lenses, different camera models are available, as they can have different geometries (see figure 2.18). Note further, that different results are obtained with different angles of incidence.

For **projective tilt lenses** and **object-side telecentric tilt lenses**, the projection of $\tilde{\mathbf{q}}^c = (\tilde{u}, \tilde{v})^T$ into the point $\mathbf{q}^t = (\hat{u}, \hat{v})^T$, which lies in the tilted image plane, is described by a projective 2D transformation, i.e., by the homogeneous matrix $\mathbf{H}$:

$$\mathbf{q}^t = \mathbf{H} \cdot \tilde{\mathbf{q}}^c \tag{2.30}$$

where

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} = \begin{pmatrix} q_{11}q_{33} - q_{13}q_{31} & q_{21}q_{33} - q_{23}q_{31} & 0 \\ q_{12}q_{33} - q_{13}q_{32} & q_{22}q_{33} - q_{23}q_{32} & 0 \\ q_{13}/d & q_{23}/d & q_{33} \end{pmatrix} \tag{2.31}$$

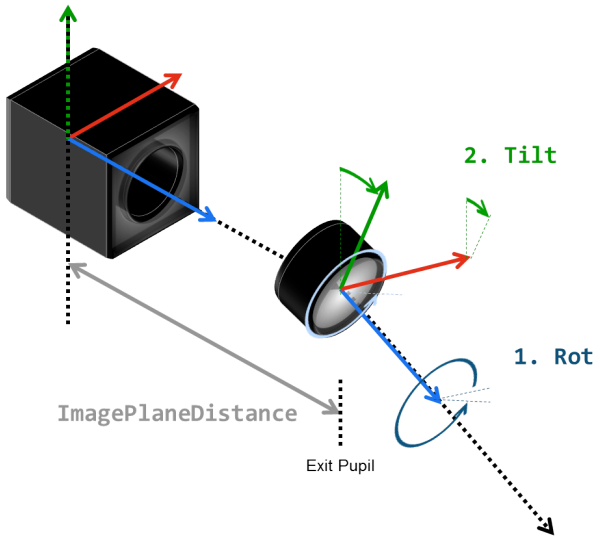
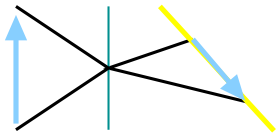
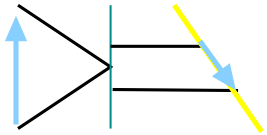


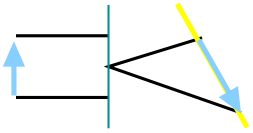
Figure 2.17: The tilt of the lens is described by the two parameters `Tilt` and `Rot`. `Rot` describes the orientation of the tilt axis. `Tilt` describes the actual tilt of the lens.



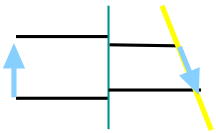
projective tilt lens ('area_scan_tilt_*')



image−side telecentric tilt lens ('area_scan_tilt_image_side_telecentric_*')



object−side telecentric tilt lens ('area_scan_tilt_object_side_telecentric_*')



bilateral telecentric tilt lens ('area_scan_tilt_bilateral_telecentric_*')

Figure 2.18: The ray geometries of the different types of tilt lenses.

with $d = \mathit{ImagePlaneDist}$ and

$$\mathbf{Q} = \begin{pmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{pmatrix} =$$

$$\begin{pmatrix} (\cos\rho)^2(1-\cos\tau)+\cos\tau & \cos\rho\sin\rho(1-\cos\tau) & \sin\rho\sin\tau \\ \cos\rho\sin\rho(1-\cos\tau) & (\sin\rho)^2(1-\cos\tau)+\cos\tau & -\cos\rho\sin\tau \\ -\sin\rho\sin\tau & \cos\rho\sin\tau & \cos\tau \end{pmatrix}$$

with $\mathit{Rot} = \rho$ and $\mathit{Tilt} = \tau$.

For **image-side telecentric tilt lenses** and **bilateral telecentric tilt lenses**, the projection onto the tilted image plane is described by a linear 2D transformation, i.e., by a 2×2 matrix:

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix} = \frac{1}{q_{11}q_{22} - q_{12}q_{21}} \begin{pmatrix} q_{22} & -q_{12} \\ -q_{21} & q_{11} \end{pmatrix} \tag{2.32}$$

where $\mathbf{Q}$ is defined as above for projective lenses.

2.2.2.5 Transformation into Pixel Coordinates

Finally, the point $\tilde{\mathbf{q}}^c = (\tilde{u}, \tilde{v})^T$ (or $\mathbf{q}^t$ if a tilt lens is present) is transformed from the image plane coordinate system into the image coordinate system (the pixel coordinate system):

$$\mathbf{q}^i = \begin{pmatrix} r \\ c \end{pmatrix} = \begin{pmatrix} \frac{\hat{v}}{S_y} + C_y \\ \frac{\hat{u}}{S_x} + C_x \end{pmatrix} \tag{2.33}$$

Here, S_x and S_y are scaling factors. For pinhole cameras, they represent the horizontal and vertical distance of the sensors elements on the CCD chip of the camera. For cameras with telecentric lenses, they represent the size of a pixel in world coordinates (not taking into account the lens distortions). The point $(C_x, C_y)^T$ is the principal point of the image. For pinhole cameras, this is the perpendicular projection of the optical center onto the image plane, i.e., the point in the image from which a ray through the optical center is perpendicular to the image plane. It also defines the center of the radial distortions. For telecentric cameras, no optical center exists. Therefore, the principal point is solely defined by the radial distortions.

The parameters $f, \mathit{Magnification}, \kappa, K_1, K_2, K_3, P_1, P_2, \tau, \rho, S_x, S_y, C_x, C_y$ are called the *internal camera parameters* because they determine the projection from 3D to 2D performed by the camera. Note that in addition, the `CameraType`, the `ImageWidth`, the `ImageHeight` and, only for object-side telecentric tilt lenses, the `ImagePlaneDist` must be given. Depending on the camera type, lens type, and lens distortion model, only a subset of the parameters is actually used, as the following list shows:

'area_scan_division'
['area_scan_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_polynomial'
['area_scan_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_division'
['area_scan_tilt_division', Focus, Kappa, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_polynomial'
['area_scan_tilt_polynomial', Focus, K1, K2, K3, P1, P2, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_image_side_telecentric_division'
['area_scan_tilt_image_side_telecentric_division', Focus, Kappa, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_image_side_telecentric_polynomial'
['area_scan_tilt_image_side_telecentric_polynomial', Focus, K1, K2, K3, P1, P2, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_telecentric_division'
['area_scan_telecentric_division', Magnification, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_telecentric_polynomial'
['area_scan_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_bilateral_telecentric_division'
['area_scan_tilt_bilateral_telecentric_division', Magnification, Kappa, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_bilateral_telecentric_polynomial'
['area_scan_tilt_bilateral_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_object_side_telecentric_division'
['area_scan_tilt_object_side_telecentric_division', Magnification, Kappa, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_object_side_telecentric_polynomial'
['area_scan_tilt_object_side_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_hypercentric_division'
['area_scan_hypercentric_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_hypercentric_polynomial'
['area_scan_hypercentric_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'line_scan_division'
['line_scan_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

'line_scan_polynomial'
['line_scan_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

'line_scan_telecentric_division'
['line_scan_telecentric_division', Magnification, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

'line_scan_telecentric_polynomial'
['line_scan_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

To create camera parameter tuples, you can use the different procedures available in HALCON. There is a procedure available for each camera type, e.g., `gen_cam_par_area_scan_division`.

We can see that camera calibration is the process of determining the internal camera parameter and the external camera parameters $(t_x, t_y, t_z, \alpha, \beta, \gamma)$.

2.2.3 Tilt Lenses and the Scheimpflug Principle

In a normal setup, the image plane is orthogonal to the optical axis of the lens (see figure 2.19). In case of strong magnification, that leads to the sometimes undesired effect, that objects that are viewed from an angle are partly out of focus, because of the limited depth of field (see figure 2.20).

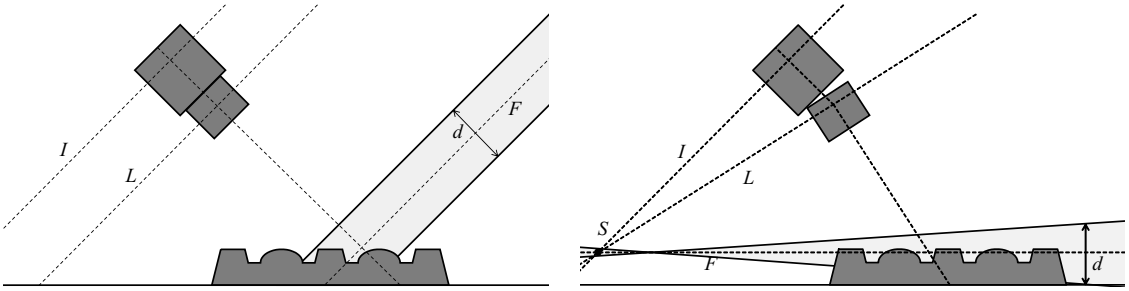


Figure 2.19: The Scheimpflug principle: Left: Without tilt lens the focus plane F is parallel to the image plane I. Therefore, the object is only partly in focus if it exceeds the depth of field d. Right: When using a tilt lens, the focus plane, the image plane, and the lens plane L are intersecting in the Scheimpflug line S. Using this principle, a planar object can be completely in focus even if it is viewed from an angle.

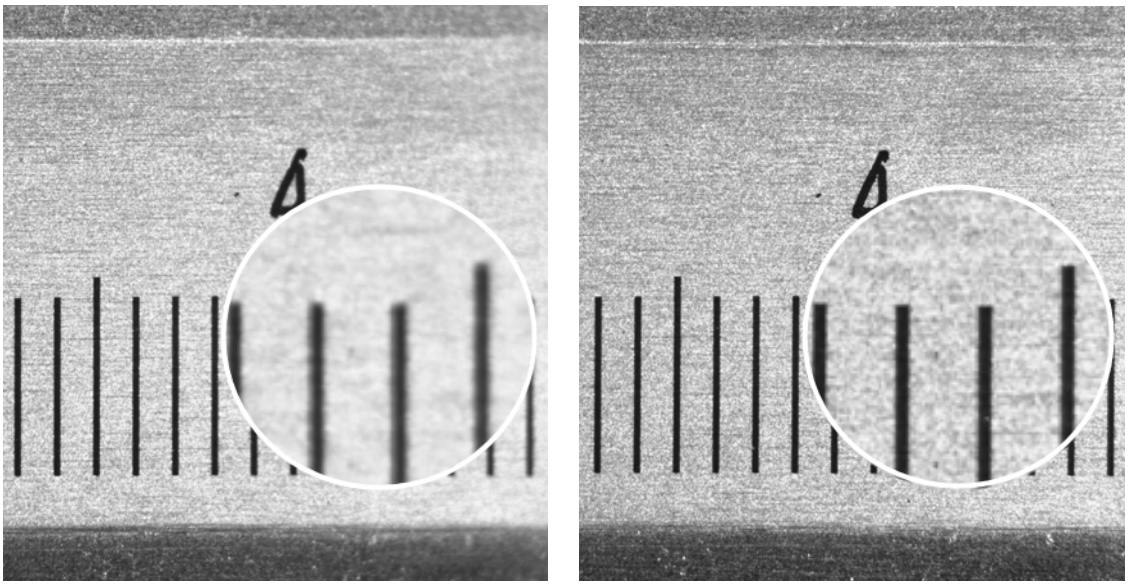


Figure 2.20: Image of a caliper taken from an angle in two different setups: Left: Without tilt lens, parts of the object are out of focus. Right: With tilt lens, the whole object is in focus.

To avoid this situation the lens or the image plane may be tilted to meet the *Scheimpflug condition*: The image plane, the lens plane, and the focus plane intersect in a single line, the *Scheimpflug line*. As a consequence, it is now possible to take images of objects that are completely in focus, even if they are not parallel to the camera.

The use of tilt lenses is especially useful if the depth-of-field is small, e.g., due to a small field of view, a large magnification, or a low f-number. In these cases, it might be necessary to use a tilt lens to adjust the focus plane to better fit the observed scene.

This is typically the case in one of the following conditions:

Physical Obstacles If the space directly above the object is blocked by obstacles, e.g., some machine parts above the conveyor belt, the camera has to be mounted in a way that the objects can only be seen from an angle. Tilt lenses can be used to align the focus plane with the object plane.

Stereo Vision In most stereo setups, the used cameras observe the scene at different angles. That means, when using normal lenses, their focal planes are also not parallel, i.e., each camera covers a different volume that is in focus. With smaller depth-of-field or larger angle between the cameras, the volume that is in focus for all cameras gets smaller, what might lead to serious problems for the reconstruction. Tilt lenses can be used to align the focus planes of the different cameras (see section 5.1.3).

Sheet of Light In a sheet-of-light setup, a laser line is projected onto the scene, while a camera is observing the line's reflection of the object. To get the most accurate results, it is desirable, that the focus plane of the camera is aligned with the sheet of light emitted by the laser. This is not always possible with normal lenses (see section 6.2).

Note that, if no tilt lens is available, in some applications it may be sufficient to increase the depth of field by using a higher f-number, i.e., a smaller aperture. Of course that implies that either the exposure times have to be increased (which may lead to a longer overall cycle time) or a stronger illumination has to be used.

2.2.4 Hypercentric Lenses

Hypercentric lenses allow to image the top and the sides of an object simultaneously with a single view.

Like conventional perspective lenses, hypercentric lenses perform a perspective projection of the world coordinates into the image. In contrast to conventional perspective lenses, however, the optical center (more precisely, the center of the entrance pupil) of hypercentric lenses lies outside in front of the lens (figure 2.21). Furthermore,

objects to be inspected are placed between the optical center and the lens. As a consequence, objects that are closer to the camera appear smaller in the image.

Because of these properties, hypercentric lenses can be used, e.g., for inspection tasks where otherwise multiple images would have to be acquired and stitched together. An example is shown in figure 2.22, where the surface of a vial must be inspected. Using a hypercentric lens allows a 360 degree inspection with a single camera image. See the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Calibration\Multi-View\calibrate_cameras_hypercentric.hdev` for more details.

As for conventional perspective lenses, the origin of the camera coordinate system of hypercentric lenses lies in the optical center of the lens. Note that because the z-axis of the camera coordinate system points in viewing direction, 3D object points have a negative z coordinate in the camera coordinate system.

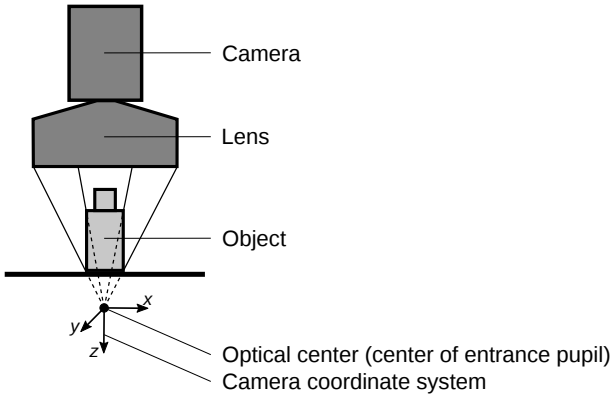


Figure 2.21: The vial is placed between the entrance pupil and the front of the lens. Note that the z coordinates of object points are negative in the camera coordinate system.

Figure 2.22: (left) Side view of the vial that should be inspected, using a regular lense. (middle) Top view using a hypercentric lense. (right) Unrolled surface of the label, after camera calibration and mapping.

2.2.5 Line Scan Cameras

A line scan camera has only a one-dimensional line of sensor elements, i.e., to acquire an image, the camera must move relative to the object (see figure 2.23). This means that the camera moves over a fixed object, the object travels in front of a fixed camera, or camera and object are both moving.

The relative motion between the camera and the object is modeled in HALCON as part of the internal camera parameters. In HALCON, the following assumptions for this motion are made:

1. the camera moves — relative to the object — with constant velocity along a straight line
2. the orientation of the camera is constant with respect to the object
3. the motion is equal for all images

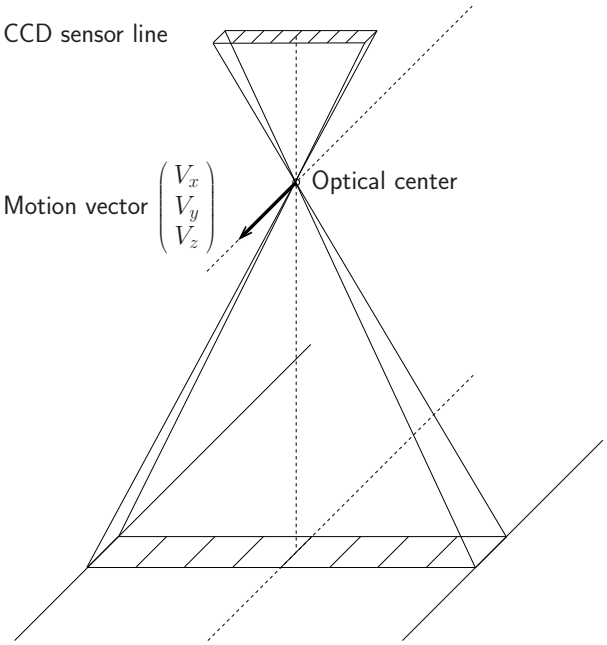


Figure 2.23: Principle of line scan image acquisition.

The motion is described by the motion vector $V = (V_x, V_y, V_z)^T$, which must be given in [meters/scanline] in the camera coordinate system. The motion vector describes the motion of the camera, i.e., it assumes a fixed object. In fact, this is equivalent to the assumption of a fixed camera with the object traveling along $-V$.

HALCON supports a pinhole camera model as well as a telecentric camera model for line scan cameras.

The camera coordinate system of pinhole line scan cameras is defined as follows (see figure 2.24): The origin of the coordinate system is the center of projection. The z-axis is identical to the optical axis and it is directed so that the visible points have positive z coordinates. The y-axis is perpendicular to the sensor line and to the z-axis. It is directed so that the motion vector has a positive y-component, i.e., if a fixed object is assumed, the y-axis points in the direction in which the camera is moving. The x-axis is perpendicular to the y- and z-axis, so that the x-, y-, and z-axis form a right-handed coordinate system. For telecentric line scan cameras, the conventions are identical except for the origin of the coordinate system: it is given by the center of distortion for telecentric line scan cameras.

Similarly to area scan cameras, the projection of a point given in world coordinates into the image is modeled in two steps: First, the point is transformed into the camera coordinate system. Then, it is projected into the image.

As the camera moves over the object during the image acquisition, also the camera coordinate system moves relative to the object, i.e., each image line has been imaged from a different position. This means that there would be an individual pose for each image line. To make things easier, in HALCON all transformations from world coordinates into camera coordinates and vice versa are based on the pose of the first image line only. The motion V is taken into account during the projection of the point $\mathbf{p}^c$ into the image.

The transformation from the WCS to the CCS of the first image line is a rigid transformation, which can be expressed by a pose or, equivalently, by the homogeneous transformation matrix ${}^c\mathbf{H}_w$. Therefore, the camera coordinates $\mathbf{p}^c = (\mathbf{x}^c, \mathbf{y}^c, \mathbf{z}^c)^T$ of point P can be calculated from its world coordinates $\mathbf{p}^w = (\mathbf{x}^w, \mathbf{y}^w, \mathbf{z}^w)^T$ simply by

$$\mathbf{p}^c = {}^c\mathbf{H}_w \cdot \mathbf{p}^w \tag{2.34}$$

The six parameters of this transformation (the three translations t_x, t_y, and t_z and the three rotations α, β, and γ) are called the *external camera parameters* because they determine the position of the camera with respect to the world. In HALCON, they are stored as a pose, i.e., together with a code that describes the order of translation and rotations.

For pinhole line scan cameras, the projection of the point $\mathbf{p}^c$ that is given in the camera coordinate system of the first image line into a (sub-)pixel [r,c] in the image is defined as follows:

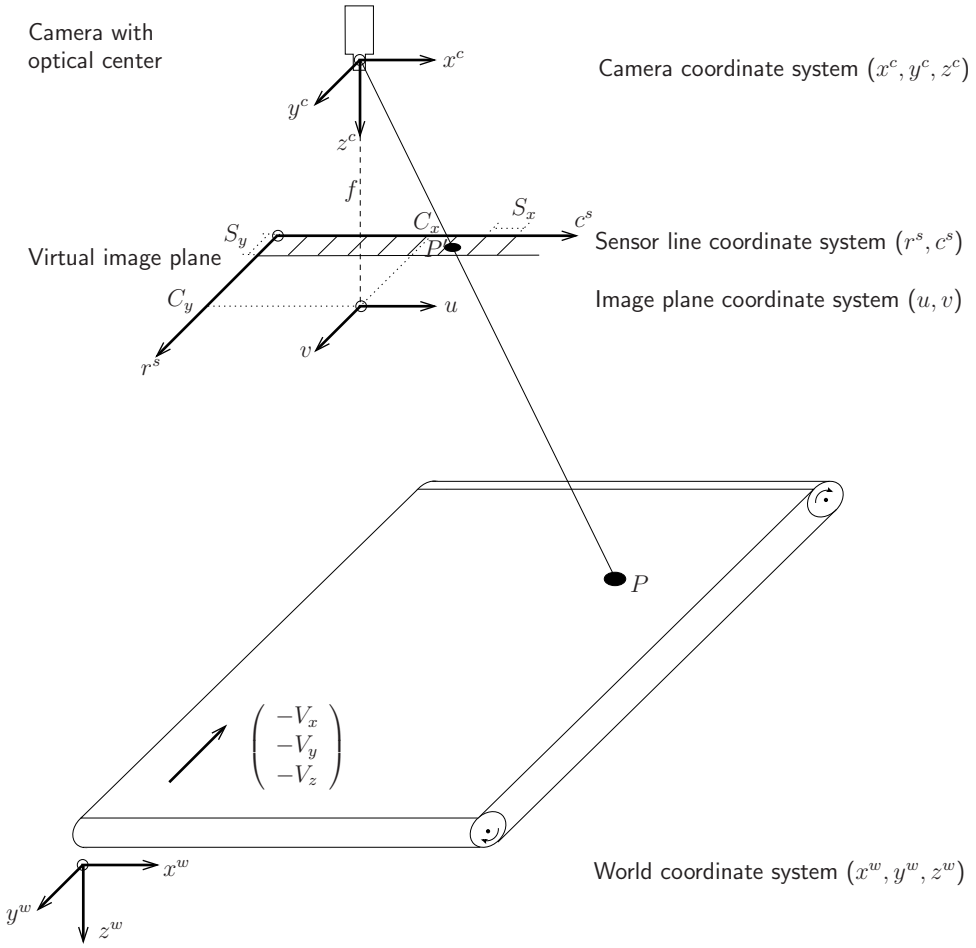


Figure 2.24: Coordinate systems in regard to a line scan camera.

Assuming

$$\mathbf{p}^c = \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

the following set of equations must be solved for λ, $\tilde{u}$, and t:

$$\begin{aligned} \lambda \cdot u(\tilde{u}, p_v) &= x - t \cdot V_x \\ \lambda \cdot v(\tilde{u}, p_v) &= y - t \cdot V_y \\ \lambda \cdot f &= z - t \cdot V_z \end{aligned}$$

where $p_v = -S_y \cdot C_y$ and $u(\tilde{u}, \tilde{v})$ and $v(\tilde{u}, \tilde{v})$ are given by equation 2.27 on page 28 for the division model and equation 2.29 on page 29 for the polynomial model.

For telecentric line scan cameras, the projection of the point $\mathbf{p}^c$ into a (sub-)pixel [r,c] in the image is achieved by solving the following set of equations for $\tilde{u}$ and t:

$$\begin{aligned} u(\tilde{u}, p_v)/m &= x - t \cdot V_x \\ v(\tilde{u}, p_v)/m &= y - t \cdot V_y \end{aligned}$$

where m denotes the magnification of the lens and p_v, $u(\tilde{u}, \tilde{v})$, and $v(\tilde{u}, \tilde{v})$ are defined as above. Note that neither z nor V_z influence the projection for telecentric cameras.

The above formulas already include the compensation for radial distortions.

Finally, the point is transformed into the image coordinate system, i.e., the pixel coordinate system:

$$c = \frac{\tilde{u}}{S_x} + C_x$$

$$r = t$$

S_x and S_y are scaling factors. S_x represents the distance of the sensor elements on the CCD line, S_y is the extent of the sensor elements in y-direction. The point $(C_x, C_y)^T$ is the principal point. Note that in contrast to area scan images, $(C_x, C_y)^T$ does not define the position of the principal point in image coordinates. It rather describes the relative position of the principal point with respect to the sensor line.

The parameters $f, m, \kappa, K_1, K_2, K_3, P_1, P_2, S_x, S_y, C_x, C_y, V_x, V_y, V_z$ are called the *internal camera parameters* because they determine the projection from 3D to 2D performed by the camera.

As for area scan cameras, the calibration of a line scan camera is the process of determining the internal camera parameters and the external camera parameters $t_x, t_y, t_z, \alpha, \beta, \gamma$ of the first image line.

2.3 3D Object Models

A 3D object model is a data structure that describes 3D objects. 3D object models can be obtained by several means and they may contain different types of data. Additionally, the different operations that use 3D object models have different requirements concerning the model's content. Thus, not every operation can be applied to every 3D object model. To provide you with the basic knowledge needed to work with 3D object models, the following sections show

- how to obtain 3D object models (section 2.3.1),
- which information typically is stored in 3D object models (section 2.3.2 on page 40),
- how to modify 3D object models (section 2.3.3 on page 43),
- how to access specific features of 3D object models (section 2.3.4 on page 49),
- how to register 3D object models, i.e., how to match different models of the same object or of overlapping object parts (section 2.3.5 on page 50), and
- how to visualize 3D object models (section 2.3.6 on page 56).

2.3.1 Obtaining 3D Object Models

The following sections give an impression on the various ways that can be used to obtain a 3D object model. Generally, 3D object models can be

- created from scratch by explicitly setting the coordinates of points lying on the object's surface or by explicitly setting the parameters of a simple 3D shape (see section 2.3.1.1),
- obtained from Computer Aided Design (CAD) data (see section 2.3.1.2), or
- derived by one of the available 3D reconstruction approaches (see section 2.3.1.3 on page 40).

2.3.1.1 Creating 3D Object Models from Scratch

3D object models can be created from scratch either by using given points that approximate the surfaces of the objects or by using the parameters of simple 3D shapes like boxes, spheres, cylinders, or planes, which are called "3D primitives". In particular, the following operators are available to create a 3D object model from scratch:

- `gen_empty_object_model_3d` creates an empty 3D object model that can be filled with content, e.g., with the operator `set_object_model_3d_attrib` or `set_object_model_3d_attrib_mod` as is described in section 2.3.3.2 on page 44,
- `gen_object_model_3d_from_points` creates a 3D object model consisting of points,
- `gen_box_object_model_3d` creates a box-shaped 3D primitive,

- gen_sphere_object_model_3d or gen_sphere_object_model_3d_center creates a sphere-shaped 3D primitive,
- gen_cylinder_object_model_3d creates a cylinder-shaped 3D primitive, and
- gen_plane_object_model_3d creates a plane-shaped 3D primitive.

For example, in the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Creation\set_object_model_3d_attrib.hdev an empty 3D object model is created with gen_empty_object_model_3d and filled with the point coordinates of a cube using set_object_model_3d_attrib_mod (see also section 2.3.3.2 on page 44). The resulting 3D object model, which is a simple point set, is shown in figure 2.25 on the left side.

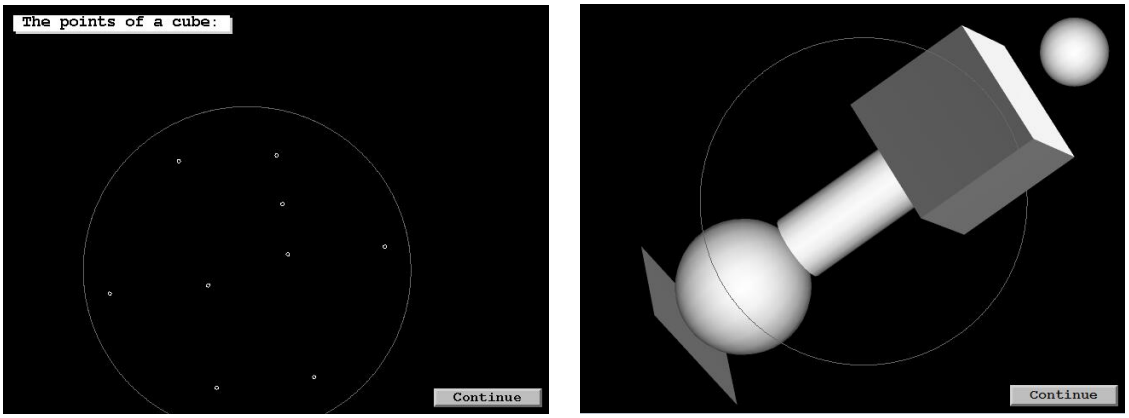


Figure 2.25: 3D object models created from scratch: (left) 3D object model defined by point coordinates, (right) 3D primitives defined by shape parameters.

```
gen_empty_object_model_3d (ObjectModel3D)
PointCoordX := [0.5, 0, 1, 1, 0, 0, 1, 1, 0] - 0.5
PointCoordY := [0, 1, 1, 1, 1, 0, 0, 0, 0] - 0.5
PointCoordZ := [0.5, 0, 0, 1, 1, 0, 0, 1, 1] - 0.5
set_object_model_3d_attrib_mod (ObjectModel3D, ['point_coord_x', \
                                'point_coord_y', 'point_coord_z'], [], \
                                [PointCoordX,PointCoordY,PointCoordZ])
```

In contrast, the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Creation\gen_primitives_object_model_3d.hdev creates different 3D primitives by specifying their parameters. The resulting 3D object models are displayed in figure 2.25 on the right side.

```
gen_plane_object_model_3d ([0, 0, 0, 0, 0, 0, 0], [], [], \
                           ObjectModel3DPlane1)
gen_sphere_object_model_3d ([0, 0, 3, 0, 0, 0, 0], 0.5, \
                            ObjectModel3DSphere1)
gen_sphere_object_model_3d_center (-1, 0, 1, 1, ObjectModel3DSphere2)
gen_cylinder_object_model_3d ([1, -1, 2, 0, 0, 60, 0], 0.5, -1, 1, \
                              ObjectModel3DCylinder)
gen_box_object_model_3d ([-1, 2, 1, 0, 0, 90, 0], 1, 2, 1, ObjectModel3DBox)
```

2.3.1.2 Obtaining 3D Object Models from Computer Aided Design (CAD) Data

If a 3D model of an object is already available as a Computer Aided Design (CAD) model, e.g., as a DXF or PLY file, the model simply can be read as 3D object model with the operator read_object_model_3d as is shown, e.g., in the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Features\smallest_bounding_box_object_model_3d.hdev (see figure 2.26). Please refer to the description of the operator read_object_model_3d in the Reference Manual for the complete list of supported file formats and their descriptions.

```
read_object_model_3d ('pipe_joint', 'm', [], [], ObjectModel3D, Status)
```

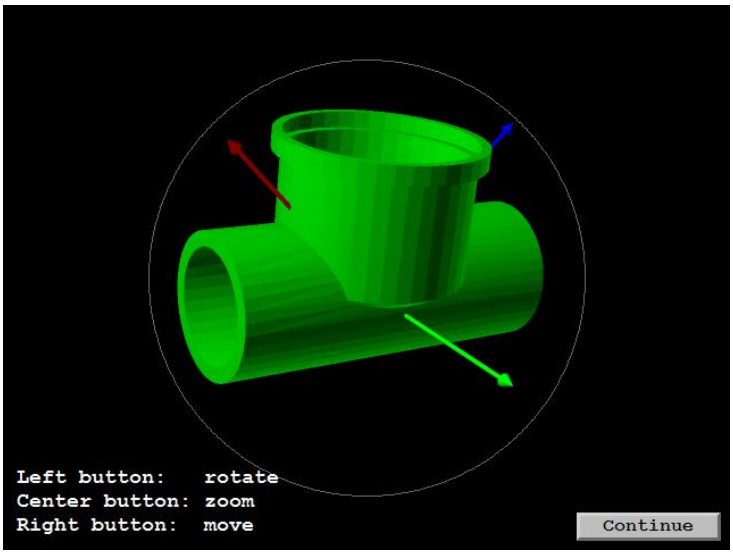


Figure 2.26: 3D object model obtained from a CAD model that is available as a PLY file.

2.3.1.3 Deriving 3D Object Models by 3D Reconstruction

All (calibrated) 3D reconstruction approaches are suitable to explicitly or implicitly derive a 3D object model. For example, with multi-view stereo you can explicitly obtain a 3D object model, whereas with a common 3D sensor X, Y, and Z images and with depth from focus depth images are obtained.

X, Y, and Z images implicitly contain the information needed for a 3D object model. Thus, you can derive a 3D object model from X, Y, and Z images using the operator `xyz_to_object_model_3d`, as is shown, e.g., in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Object-Model\Features\select_object_model_3d.hdev` (see figure 2.27).

```
ImagePath := 'time_of_flight/'
read_image (Image, ImagePath + 'engine_cover_xyz_01')
scale_image (Image, Image, .001, .0)
zoom_image_factor (Image, Image, 2, 2, 'constant')
decompose3 (Image, X, Y, Z)
xyz_to_object_model_3d (X, Y, Z, ObjectModel3DID)
```

If only a (calibrated) depth image, i.e., a "Z image" is available, you can create artificial X and Y images as follows: The X and Y images must have the same size as the Z image. The X image is created by assigning the column numbers of the Z image to each row of the X image and the Y image is created by assigning the row numbers of the Z image to each column of the Y image as is illustrated in a simple (pixel-precise) version in figure 2.28. The thus created X, Y, and Z images can then be transformed again into a 3D object model. Note that the X and Y images generated in this way usually still need to be multiplied by a suitable factor reflecting the distance between adjacent points in the X and Y directions, respectively, given in units of the Z values.

Figure 2.29 on page 42 guides you through the different ways how to derive a 3D object model that can be used, e.g., for a following 3D position recognition approach.

2.3.2 Content of 3D Object Models

The following sections give an impression on

- what kind of information generally may be stored in 3D object models (see section 2.3.2.1) and
- how to query the information contained in a specific 3D object model (see section 2.3.2.2 on page 43).

Figure 2.27: 3D object model obtained from X,Y, and Z images.

X image

0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4

Y image

0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3

Figure 2.28: Partial content of (pixel-precise) X and Y images created to complete a Z image to an X, Y, and Z image.

2.3.2.1 Kind of Information Contained in 3D Object Models

Because 3D object models can be obtained by several means, the information that is contained in 3D object models differs from model to model. For example, if a 3D object model is created explicitly from given points using the operator `gen_object_model_3d_from_points`, the basic information in this model is a set of point coordinates. In contrast, a model that is created explicitly by the parameters of a 3D primitive contains no points but the parameters of the corresponding simple 3D shape.

Generally, the content of a specific 3D object model depends on the specific process that was used to create or derive it. For example, if a 3D object model is derived from a (calibrated) 3D reconstruction method like stereo vision, sheet of light, or depth from focus, it contains points. If within such a 3D object model a 3D primitives fitting is applied, the model of the resulting 3D primitive contains points as well as the primitive's parameters. That is, the 3D object model of such a 3D primitive contains more information than that of a 3D primitive that was created explicitly by its parameters.

A 3D object model that is obtained from X, Y, and Z images typically contains the coordinates of the 3D points and the corresponding 2D mapping, i.e., a mapping of the 3D points to 2D image coordinates, whereas a 3D object model that is obtained by multi-view stereo can contain a lot of further information. For example, the surface may be approximated by triangles or polygons. A triangulation can be applied also explicitly to a 3D object model that contains points using the operator `triangulate_object_model_3d`. Additional operations to modify 3D object models are introduced in section 2.3.3 on page 43.

The following lists the different kind of data that may be contained in a 3D object model:

- Points: Coordinates of the 3D points

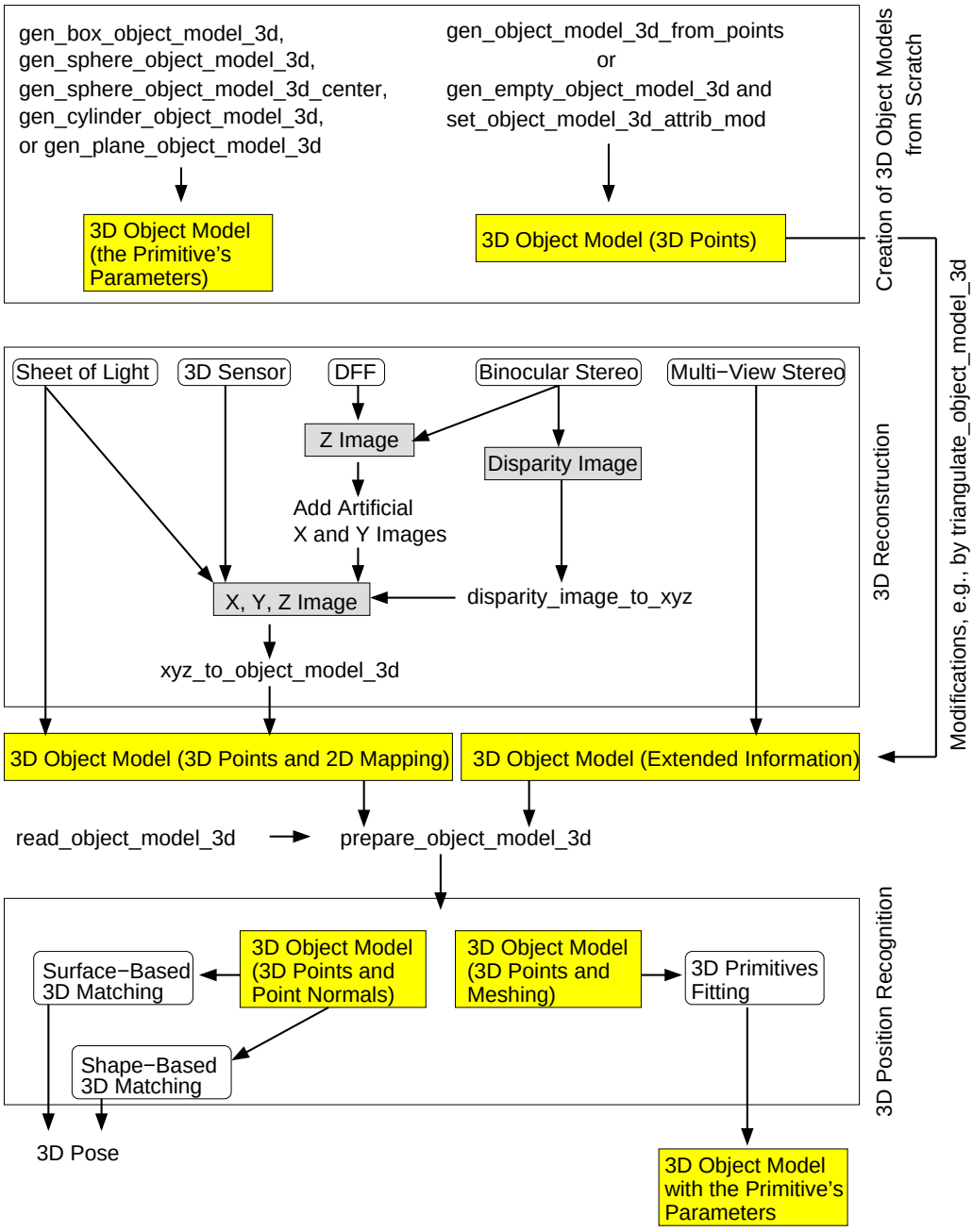


Figure 2.29: Overview on the 3D object model.

If contained in the 3D object model, further parameters can be:

- Triangles: Indices of the 3D points that represent triangles
- Lines: Indices of the 3D points that represent polylines
- Faces: Indices of the 3D points that represent faces
- Normals: Normal vectors
- xyz Mapping: Mapping of a 3D point to image coordinates

- Primitive
  If contained in the 3D object model, further parameters are:
  - Primitive Type: Type of the primitive (plane, sphere, box, cylinder)
  - Primitive Pose: Pose that describes the position and orientation of the primitive

  - Primitive Rms: Accuracy of the primitive parameters (only available if they were determined by fitting the primitive into a point cloud)

  Note that a 3D object model may contain at most one primitive.

- Extended Attributes
  - Attribute Names: Names of extended attributes defined for the 3D object model
  - Attribute Types: Types of extended attributes defined for the 3D object model
- Additional Attributes
  - Shape Based Data: Flag that indicates if the 3D object model has been prepared for shape-based 3D matching
  - Distance Computation: Flag that indicates if the 3D object model has been prepared for distance computation

The content is represented in form of attributes and can be accessed with the operator `get_object_model_3d_params`. For the complete list and a detailed description of the available attributes, see the reference manual entry of `get_object_model_3d_params`.

2.3.2.2 Querying Information from Specific 3D Object Models

Because many approaches exist to obtain and modify a 3D object model, also many different combinations of information can be contained in the models. To query the actual content of a specific 3D object model you can apply the operator `get_object_model_3d_params`. With this operator, you can query if a specific information, a so-called 'attribute', is contained in the model. For example, you can query if the 3D object model contains primitive data, points, point normals, triangles, faces, or a 2D mapping. If an attribute is contained in the model, you can query its explicit values with the same operator.

Note that here only the most common attributes were listed. The complete list of attributes that may be contained in a 3D object model and their descriptions are provided with the operator `get_object_model_3d_params` in the Reference Manual.

Another possibility to check the actual content of a 3D object model is given with a special inspection window in HDevelop. Here, the parametric properties of the 3D object model that are listed in the previous section are displayed as described in the HDevelopUser's Guide, section 6.22.8 on page 176.

2.3.3 Modifying 3D Object Models

The following sections show how to modify 3D object models by

- preparing them for a following 3D pose recognition (see section 2.3.3.1),
- adding attributes to their content (see section 2.3.3.2),
- reducing their content to selected attributes (see section 2.3.3.3 on page 46),
- changing their contained point sets, e.g., by a reduction of the number of points or by a selection of points with specific characteristics (see section 2.3.3.4 on page 46),
- transforming them (see section 2.3.3.5 on page 48), or
- combining several 3D object models to a single 3D object model (see section 2.3.3.6 on page 49).

2.3.3.1 Preparing 3D Object Models for a Following 3D Pose Recognition

Some 3D position recognition approaches need 3D object models as input. Especially the shape-based 3D matching (see section 4.2 on page 95) and the segmentation that is applied in the context of a 3D primitives fitting (see section 4.5 on page 111) need specific information that may be contained only implicitly in the 3D object model. To prepare a 3D object model for one of these 3D position recognition approaches, i.e., to enable a faster internal access to this information, you can use the operator `prepare_object_model_3d`. There, for the shape-based 3D matching the parameter `Purpose` must be set to `'shape_based_matching_3d'`, whereas for the 3D segmentation it must be set to `'segmentation'`. Note that for a surface-based 3D matching (see section 4.3 on page 104) such a preparation is not needed for the actual matching, but may also be required if a visualization with the operator `project_object_model_3d` using one of the generic parameters `'hidden_surface_removal'` or `'min_face_angle'` is applied. Then, similar to the shape-based 3D matching the parameter `Purpose` of `prepare_object_model_3d` must be set to `'shape_based_matching_3d'`.

2.3.3.2 Adding Attributes to 3D Object Models

Sometimes, specific attributes are needed, e.g., for a following operation, that are not contained explicitly but only implicitly in a 3D object model. Then, different operators are available that can be used to make the implicit information explicit or to manually add or modify specific attributes, e.g.,

- `set_object_model_3d_attrib` and `set_object_model_3d_attrib_mod` can be used to manually add attributes to a 3D object model or to modify the already contained attributes. The main difference between both operators is that `set_object_model_3d_attrib` returns a new 3D object model, whereas `set_object_model_3d_attrib_mod` changes the input 3D object model. Besides standard attributes that can be obtained during the creation or modification of a 3D object model and that are expected by various operators, so-called "extended" attributes can be set, i.e., new types of attributes can be defined by the user and attached to a model. These attributes must be indicated in the parameter `Name` by a preceding "&".
- `surface_normals_object_model_3d` can be used to add the normals attribute to 3D object models.
- `triangulate_object_model_3d` can be used to add the triangles attribute to a 3D object model that consists of points and their normals. In particular, the returned 3D object model contains a mesh of triangles that either fits perfectly to the set of points contained in the 3D object model ("greedy" algorithm) or that approximates the set of points ("implicit" algorithm).
- `fit_primitives_object_model_3d` can be used to obtain the attributes that are related to 3D primitives. In particular, it can be used to add the parameters of a 3D primitive that fits best into the set of points given in the original 3D object model. Note that such a 3D primitives fitting is often preceded by a segmentation of the 3D object model into different 3D object models having similar characteristics as is described in section 4.5 on page 111.

How to add attributes using `set_object_model_3d_attrib` or `set_object_model_3d_attrib_mod` is shown, e.g., in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Object-Model\Creation\set_object_model_3d_attrib.hdev`. There, the coordinates of the corner points of a cube were added to an empty 3D object model (see figure 2.30, left) as was already introduced in section 2.3.1.1 on page 38. Additionally, the attribute for triangles and the corresponding triangle information (see figure 2.30, right) is added.

```
T1 := [0, 8, 5]
T2 := [0, 7, 8]
T3 := [0, 6, 7]
T4 := [0, 5, 6]
T5 := [5, 6, 2]
T6 := [5, 2, 1]
T7 := [6, 7, 3]
T8 := [6, 3, 2]
T9 := [7, 3, 4]
T10 := [7, 4, 8]
T11 := [8, 4, 1]
T12 := [8, 1, 5]
T13 := [2, 3, 4]
T14 := [2, 4, 1]
set_object_model_3d_attrib (ObjectModel3D, 'triangles', [], [T1,T2,T3,T4,T5, \
                            T6,T7,T8,T9,T10,T11,T12,T13,T14], \
                             ObjectModel3D2)
```

Note that one of the sides of the cube contains an additional point that is needed to demonstrate how to modify a point coordinate with the operator `set_object_model_3d_attrib_mod` in a following step (see figure 2.31). Thus, this side of the cube is not represented by two but by four triangles.

```
PointCoordY[0, 3, 6, 7] := [-1, 0.3, -.8, -.3]
set_object_model_3d_attrib_mod (ObjectModel3D2, 'point_coord_y', [], \
                                PointCoordY)
```

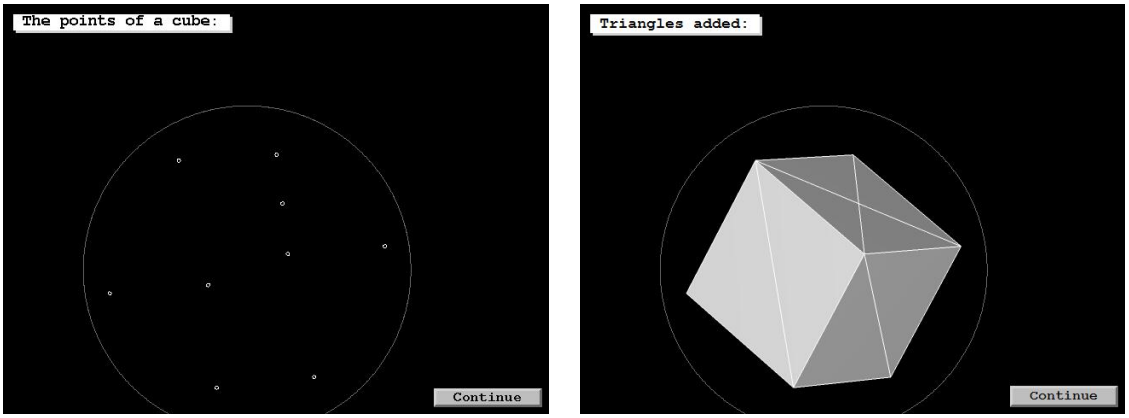


Figure 2.30: 3D object model of a cube: (left) model with point coordinates and (right) with triangles added.

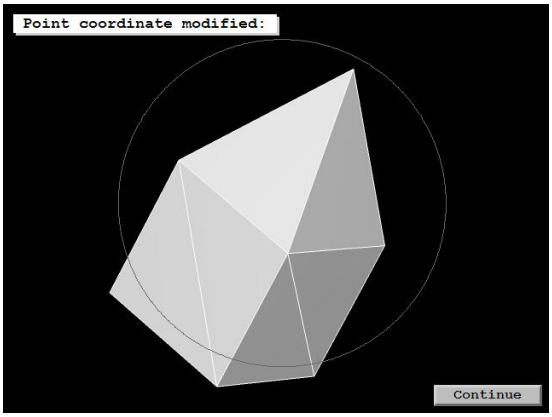


Figure 2.31: 3D object model after modifying a point coordinate.

2.3.3.3 Removing Attributes from 3D Object Models

To reduce the amount of data stored in a 3D object model, selected attributes of a 3D object model can be removed from the model if they are not needed anymore. In particular, the operator copy_object_model_3d can be used to copy a 3D object model such that only selected attributes are copied to the new 3D object model. For example, in the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Segmentation\segment_object_model_3d.hdev a simultaneous segmentation and 3D primitives fitting has been applied to a 3D object model that was obtained from X, Y, and Z images. To save memory, information like the point coordinates and the 2D mapping, which were obtained automatically during the creation of the 3D object model from the X, Y, and Z images, are removed from the model by copying only the data related to the obtained 3D primitives.

```
segment_object_model_3d (ObjectModel3DID, [ParSegmentation,ParFitting], \
                         [ValSegmentation,ValFitting], ObjectModel3DOutID)
for Index := 0 to |ObjectModel3DOutID| - 1 by 1
    copy_object_model_3d (ObjectModel3DOutID[Index], 'primitives_all', \
                          CopiedObjectModel3DID)
endfor
```

2.3.3.4 Modifying the Point Sets of 3D Object Models

Various operators are provided that enable the modification of the 3D object model's point set or the segmentation of a 3D object model into different parts. For example,

- set_object_model_3d_attrib_mod can be used to directly modify point coordinates as was shown already in section 2.3.3.2 on page 44.
- select_points_object_model_3d applies thresholds to selected attributes to reduce the set of points to those points that lie within the specified attribute value ranges.
- reduce_object_model_3d_by_view can be used to reduce the point set to a set of points lying within a specified 2D region that is defined for a specified 2D projection of the 3D object model.
- sample_object_model_3d can be used to achieve 3D object models with a uniform point density with a specified distance between the points. This is suitable to generally reduce the point density, e.g., to enable a faster following operation like a triangulation, or if the 3D object model contains parts for which the point density is higher than for other parts of the object. The latter case may result, e.g., from a multi-view stereo reconstruction (see section 5.4 on page 139) or from the registration and fusion of multiple 3D object models (see section 2.3.5 on page 50). Note that, depending on the specified distance between the points, the point density of the resulting 3D object model may even be higher than that of the original one.
- simplify_object_model_3d can be used to reduce the number of points of triangulated 3D object models, especially for smooth parts of the 3D object model, i.e., for parts where a high point density is often not necessary. This may be used, for example, to speed up subsequent operator calls by using the resulting 3D object model with reduced complexity. Typically, the point density of the simplified 3D object model is nonuniform. The point density is higher for parts where more points are required to represent the object's geometry and it is lower for smoother parts. This is in contrast to the results of sample_object_model_3d, where a uniform point density is achieved.
- smooth_object_model_3d smoothens the surface of the 3D object model. Typically, it is used to prepare a 3D object model for a surface triangulation or to smooth noisy point data within a 3D object model.
- segment_object_model_3d segments a 3D object model into parts with similar characteristics, e.g., the same orientation of the normals or a similar curvature. Such a segmentation can be applied, e.g., in the context of a 3D primitives fitting (see section 4.5 on page 111).
- connection_object_model_3d segments a 3D object model into parts that consist of connected components. Whether two components are considered as being connected depends on user-specified criteria, in particular attributes or distance functions and their corresponding threshold values.

For example, in the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Features\select_object_model_3d.hdev select_points_object_model_3d is used to select points of a 3D object model using a threshold on the z coordinates. Thus the individual parts of the engine cover shown in figure 2.32 on the left side are separated from their background as is shown in figure 2.32 on the right side.

```
select_points_object_model_3d (ObjectModel3DID, 'point_coord_z', MinValue, \
                               MaxValue, ObjectModel3DIDReduced)
```

After excluding the background from the model, the remaining parts are further segmented into connected parts with connection_object_model_3d as is shown in figure 2.33.

```
connection_object_model_3d (ObjectModel3DIDReduced, 'distance_3d', 0.010, \
                            ObjectModel3DIDConnections)
```

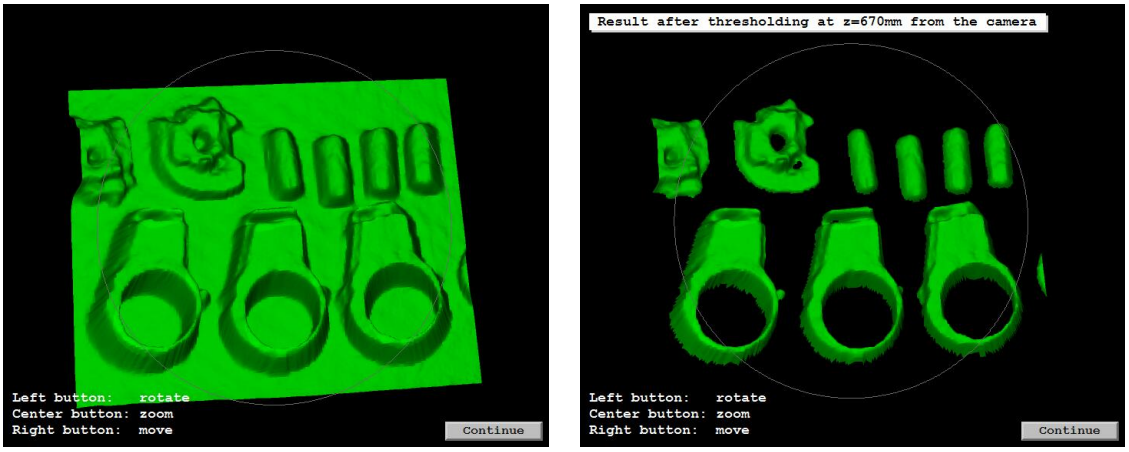


Figure 2.32: 3D object model (left) in its initial state and (right) after removing points of the background using a threshold.

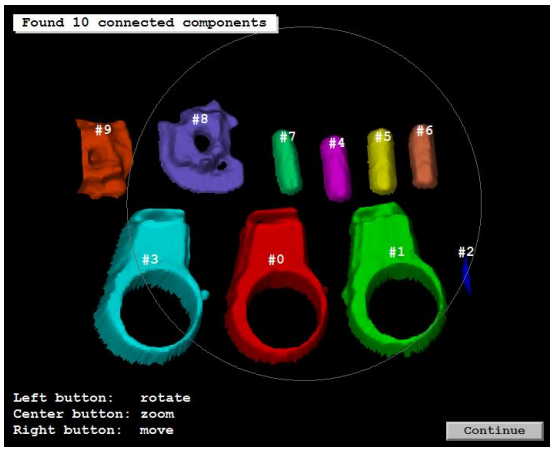


Figure 2.33: 3D object model after the segmentation into its connected components.

Another way to effectively reduce your 3D object model point set for further processing steps is the usage of the 2D mapping of the point data. By calling object_model_3d_to_xyz with *'cartesian_faces'* set for Type, a 3D object model (which needs to contain polygon or triangle faces) is transformed into the three images X, Y, and Z.

Thereby, the three images solely contain information about those parts of the model, that can be observed by a camera with specified pose and parameters, while hidden parts are omitted (see figure 2.34). With xyz_to_object_model_3d you can then perform the reverse transformation and obtain the reduced 3D object model.

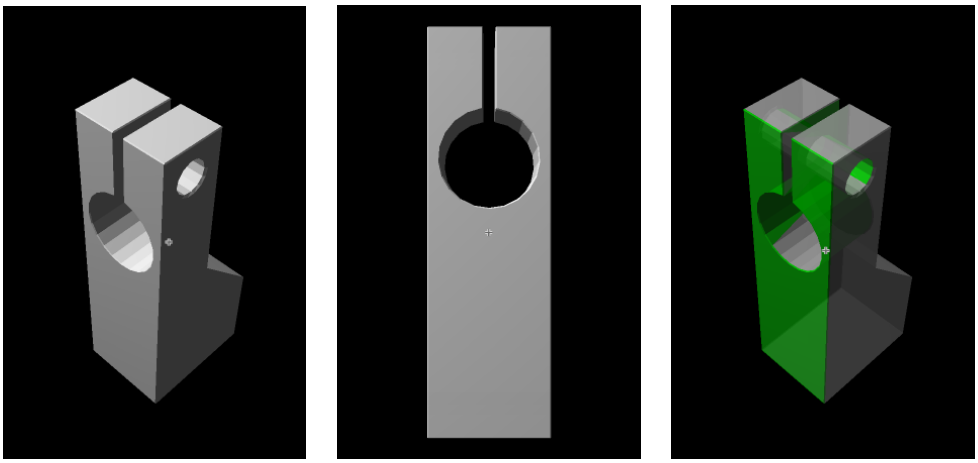


Figure 2.34: 3D object model (left) in its initial state and as viewed from the specified camera pose (center). The model is reduced to the parts visible to the camera, using a 2D mapping. The resulting object model (green) and the initial model (gray) are visualized together for comparison (right).

```
* Transform the 3D model to XYZ images, reducing the data
* according to the chosen camera parameters and pose.
object_model_3d_to_xyz (X, Y, Z, ObjectModel3D, 'cartesian_faces', \
                        CameraParam, Pose)
* Transform the XYZ images to a 3D object model.
xyz_to_object_model_3d (X, Y, Z, IntermediateObjectModel3D)
```

In the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Object-Model\Segmentation\reduce_object_model_3d_to_visible_parts.hdev` the reduction of an 3D object model using a 2D mapping is shown step by step.

Not just for the reduction of model data, but also for tasks like surface comparison, 2D mappings can be very useful, as by handling 2D images instead of 3D point clouds, significant speed-up can be achieved. After locating a 3D object model in the observed scene with any alignment method, potential deviations between model and scene can be detected by comparing their respective 2D mappings. Therefore, use `sub_image` on the `Z` images of scene and model, which results in an image, where higher pixel values represent higher deviation between scene and model. Extract the domain of the defect by applying a suitable threshold. You can call `reduce_object_model_3d_by_view` with `CamParam` set to *'xyz_mapping'* in order to reduce your point cloud to the respective view.

In general, the usage of 2D mapping increases speed when handling point clouds. This can for instance be the case when preparing a 3D object model for surface-based matching, e.g., by removing outliers, smoothing or reducing its domain. Especially the operators `sample_object_model_3d` and `surface_normals_object_model_3d` benefit, if a 3D object model contains a 2D mapping.

2.3.3.5 Transforming 3D Object Models

3D object models can be spatially transformed by several means. In particular, you can transform 3D object models by

- a rigid 3D transformation using the operator `rigid_trans_object_model_3d`,
- an arbitrary affine 3D transformation using the operator `affine_trans_object _model_3d`, or
- an arbitrary projective 3D transformation using the operator `projective_trans_object_model_3d`.

Note that the transformed 3D object model contains only data that can be represented by a 3D object model and that could be transformed. For example, after an affine 3D transformation no 3D primitives will be contained in the transformed 3D object model. For the transformation of 3D primitives only the rigid transformation is suitable.

2.3.3.6 Combining 3D Object Models

To combine several 3D object models to a single 3D object model, you can apply the operator union_object_model_3d. Note that the resulting model contains only those attributes that are contained in all of the input 3D object models. Alternatively, you can use fuse_object_model_3d to fuse multiple 3D object models (that are registered in the same coordinate system) into a surface. See the HDevelop example %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Transformations\fuse_object_model_3d.hdev to see how to fine-tune the parameters for this operator.

2.3.4 Extracting Features of 3D Object Models

The following sections show how to

- calculate or access specific features of 3D object models and
- how to select 3D object models by their specific features.

2.3.4.1 Calculating or Accessing Features of 3D Object Models

Several kinds of features are contained explicitly or implicitly in 3D object models. On the one hand, the 3D object models explicitly contain different attributes as was introduced in section 2.3.2 on page 40. These attributes can be related to features like point coordinates, normals, triangles, faces, or the parameters of 3D primitives and can be accessed with the operator get_object_model_3d_params.

On the other hand, the 3D object models contain implicit information about specific geometric features, i.e., the contained information can be used to explicitly calculate these features using one of the many available operators that are provided by HALCON:

- area_object_model_3d calculates the area of the surface of the 3D object model,
- distance_object_model_3d calculates the distances of the points in one 3D object model to the points, triangles, or primitive in another 3D object model,
- max_diameter_object_model_3d calculates the maximum diameter of the 3D object model,
- moments_object_model_3d calculates the mean or the central moment of second order of the 3D object model,
- smallest_bounding_box_object_model_3d and smallest_sphere_object_model_3d calculate the smallest bounding box or the smallest sphere surrounding the 3D object model,
- volume_object_model_3d_relative_to_plane calculates the volume of the 3D object model relative to a plane if triangles or a list of polygons is contained in the 3D object model, and
- intersect_plane_object_model_3d calculates an intersetion between the 3D object model and a plane and returns the cross section as a set of 3D points that are connected by lines (see, e.g., figure 2.35, which shows the result of the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Transformations\intersect_plane_object_model_3d.hdev).

In the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Features\select_object_model_3d.hdev the maximum diameters and the volumes for the connected components of the 3D object model that were introduced in section 2.3.3.4 on page 46 are calculated (see figure 2.36).

```
volume_object_model_3d_relative_to_plane (ObjectModel3DIDConnections, [0, 0, \
                                          MaxValue,0, 0, 0, 0], 'signed', \
                                           'true', Volume)
max_diameter_object_model_3d (ObjectModel3DIDConnections, Diameter)
```

Figure 2.35: (left) 3D object model of a mug and (right) its intersection with a plane.

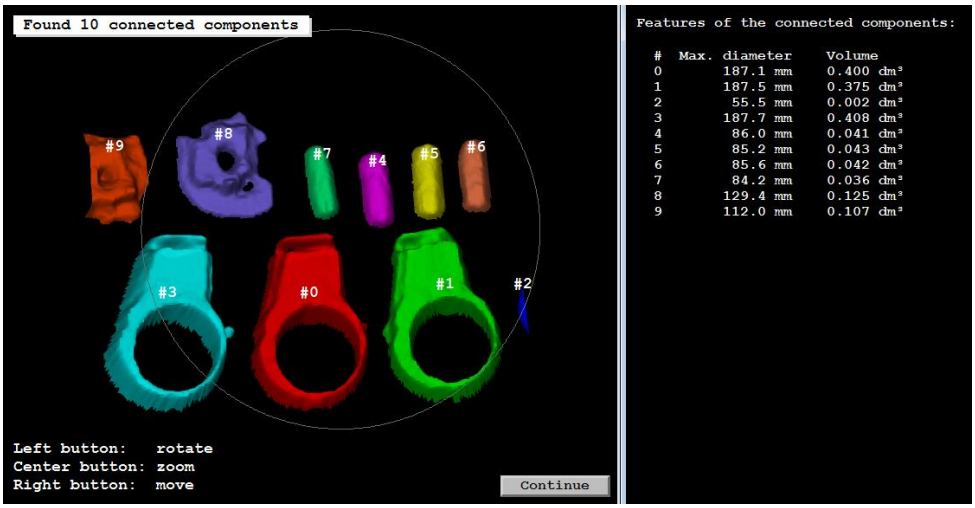


Figure 2.36: Features calculated for the connected components of a 3D object model.

2.3.4.2 Selecting 3D Object Models by Their Features

If many 3D object models are available but only those 3D object models are needed that have specific features, the operator `select_object_model_3d` can be applied. It selects 3D object models from an array of 3D object models according to global features like the existence of specific attributes or specific value ranges for features like the object's mean diameter or the object's volume.

For example, in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Object-Model\Features\select_object_model_3d.hdev` the maximum diameter and the volume of the connected components of the 3D object model are used to select only those components that have a certain size. The selected models are visualized in figure 2.37.

```
select_object_model_3d (ObjectModel3DTranslated, ['volume', \
                        'diameter_object'], 'and', [MinVolume,MinDiameter], \
                        [MaxVolume,MaxDiameter], ObjectModel3DSelected)
```

2.3.5 Matching of 3D Object Models

HALCON provides functionality for matching 3D object models that represent the same object or overlapping parts of the same object. This matching is also called "registration" of 3D object models.

The following sections

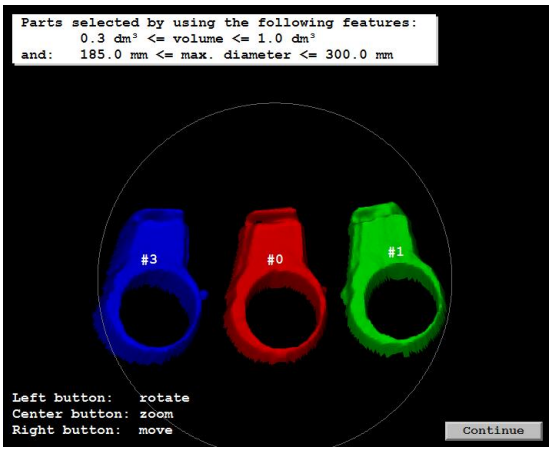


Figure 2.37: Connected components of a 3D object model, selected by their maximum diameter and volume.

- introduce pair-wise and global registration of 3D object models (see section 2.3.5.1) and
- show how registration and further modifications of 3D object models can be used to derive a surface model for a surface-based 3D matching (see section 2.3.5.2).

2.3.5.1 Registering 3D Object Models

With the operator `register_object_model_3d_pair` you can apply a matching between two 3D object models to get the pose that describes the spatial relation between them. This process is also called "pair-wise registration" of 3D object models. The matching determines an initial pose that is then consecutively refined so that the mutual difference between the overlapping parts of both 3D object models becomes minimal. The final pose can be used to transform the first 3D object model into the coordinate system of the second 3D object model. Note that if both models are already available in the same coordinate system, the initial pose between them is already implicitly known. Then, the operator can also be applied in a mode that performs no initial matching but only the pose refinement (`Method='icp'`).

With the operator `register_object_model_3d_global` you can improve the relative positions between many 3D object models, which is also called "global registration" of 3D object models. In particular, if a 3D object consists of several overlapping 3D object models, the specific overlaps are used to determine an array of homogeneous transformation matrices with which the models can be transformed such that the relation between all models is refined, i.e., a minimal mutual difference between all 3D object models is realized, which leads to a better representation of the complete object.

For both operators, the result of the registration is typically used to transform the initial 3D object models with the operator `affine_trans_object_model_3d`.

2.3.5.2 Example Application: Deriving a Surface Model from a Set of 3D Images

The HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Robot-Vision\reconstruct_3d_object_model_for_matching.hdev` shows how registration is used to derive a unique surface model for a surface-based 3D matching (see section 4.3 on page 104) from a set of views on the same object that are obtained by a 3D sensor. In particular, for each view on the object a 3D object model and a corresponding gray value image are available (see figure 2.38).

Step 1: Register 3D object models

With a pair-wise registration, a following global registration, and the corresponding affine transformations, the 3D object models of the initial views on the object are aligned. In particular, a pair-wise registration is applied for all 3D object models of successive views.

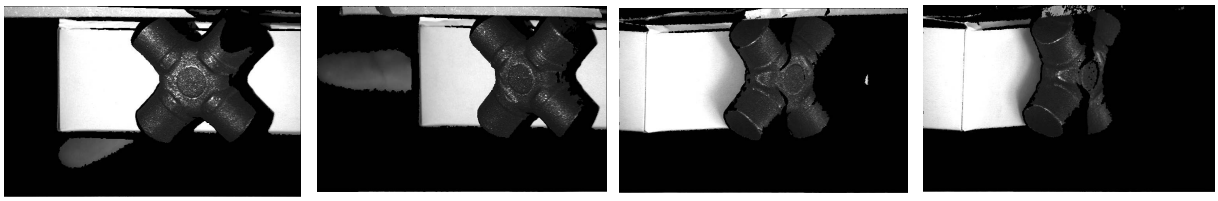


Figure 2.38: Some of the images obtained together with the corresponding 3D object models by a 3D sensor.

```
read_object_model_3d ('universal_joint_part/universal_joint_part_xyz_00.om3', \
                      'm', [], [], ObjectModel3D, Status)
PreviousOM3 := ObjectModel3D
RegisteredOM3s := ObjectModel3D
for Index := 1 to NumTrainingImages - 1 by 1
    read_object_model_3d ('universal_joint_part/universal_joint_part_xyz_' + Index$'02d', \
                          'm', [], [], ObjectModel3D, Status)
    register_object_model_3d_pair (ObjectModel3D, PreviousOM3, 'matching', \
                                   'default_parameters', 'accurate', Pose, \
                                   Score)
pose_to_hom_mat3d (Pose, HomMat3D)
RegisteredOM3s := [RegisteredOM3s,ObjectModel3D]
Offsets := [Offsets,HomMat3D]
PreviousOM3 := ObjectModel3D
endfor
```

For example, in figure 2.39 the corresponding images for the 3D object models of view 12 and view 13 are shown and the result of transforming both 3D object models into the same coordinate system is displayed.

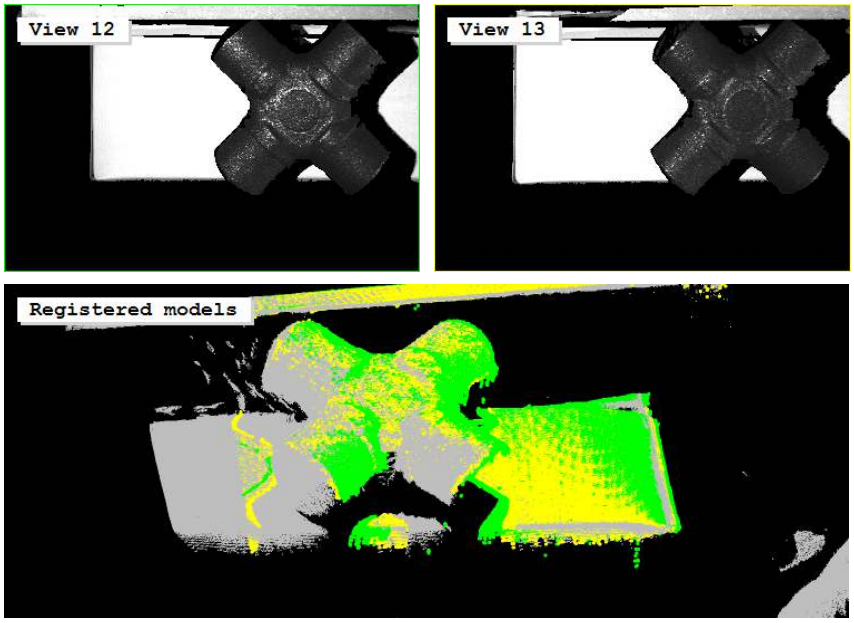


Figure 2.39: Pair-wise registration of two 3D object models: (top) images corresponding to the registered 3D object models, (bottom) visualization of the pair-wise registered and transformed 3D object models.

After all pair-wise registrations have been applied, all 3D object models can be transformed into the coordinate system of the first model (see figure 2.40, left). The relations between the thus transformed models are then refined by a global registration and the corresponding affine transformations (see figure 2.40, right).

```
register_object_model_3d_global (RegisteredOM3s, Offsets, 'previous', [], \
                                 'max_num_iterations', 1, HomMat3DRefined, \
                                 Score)
affine_trans_object_model_3d (RegisteredOM3s, HomMat3DRefined, \
                              GloballyRegisteredOM3s)
```

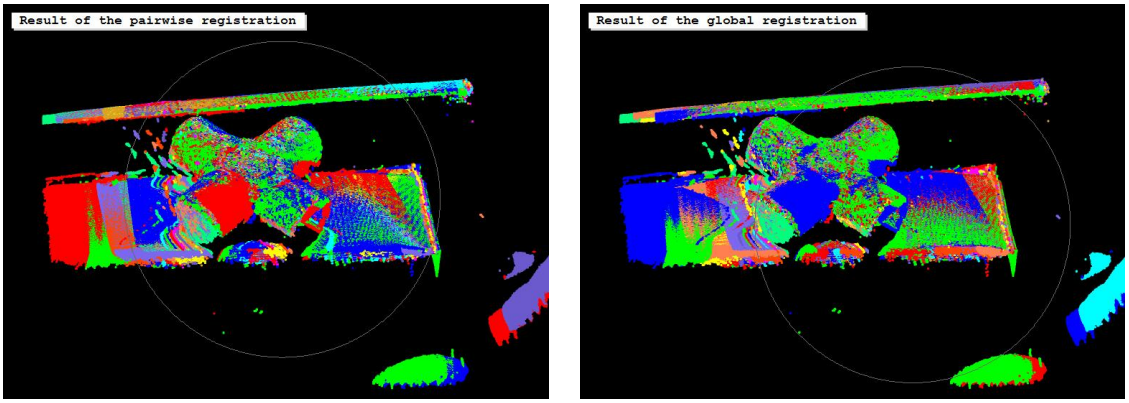


Figure 2.40: The 3D object models (left) after the successively applied pair-wise registration and (right) after the global registration.

Step 2: Extract surface model from the registered 3D object models

The registered 3D object models consist of different structures, in particular of a universal joint, i.e., the actual object that should be used as surface model, and parts of the background. The extraction of the surface model from the set of aligned 3D object models is done in several steps. First, the models are combined into a single 3D object model using `union_object_model_3d` (see figure 2.41, left). To get an object with a smaller and homogeneous point density, `sample_object_model_3d` is applied (see figure 2.41, right).

```
union_object_model_3d (GloballyRegisteredOM3s, 'points_surface', \
                       UnionOptimized)
MinNumPoints := 5
SamplingParam := 0.5
sample_object_model_3d (UnionOptimized, 'accurate', SamplingParam, \
                        'min_num_points', MinNumPoints, SampleExact)
```

After that, the object is further smoothed and the surface normals are calculated. In this example, the object is moved temporarily, so that the origin of the coordinate system lies below the object. This way, in combination with the *'mls_force_inwards'* option of `smooth_object_model_3d`, the normals of the smoothed object model will point downwards making the surface-based 3D matching more robust.

```
get_object_model_3d_params (SampleExact, 'center', Center)
get_object_model_3d_params (SampleExact, 'bounding_box1', BoundingBox)
hom_mat3d_identity (HomMat3DTrans)
hom_mat3d_translate_local (HomMat3DTrans, -Center[0], -Center[1], \
                           -BoundingBox[2], HomMat3DTranslate)
affine_trans_object_model_3d (SampleExact, HomMat3DTranslate, \
                              SampleExactTrans)
smooth_object_model_3d (SampleExactTrans, 'mls', 'mls_force_inwards', \
                        'true', SmoothObject3DTrans)
hom_mat3d_invert (HomMat3DTranslate, HomMat3DInvert)
affine_trans_object_model_3d (SmoothObject3DTrans, HomMat3DInvert, \
                              SmoothObject3D)
```

Then, the resulting model is triangulated with `triangulate_object_model_3d` and the connected components are determined with `connection_object_model_3d` (see figure 2.42).

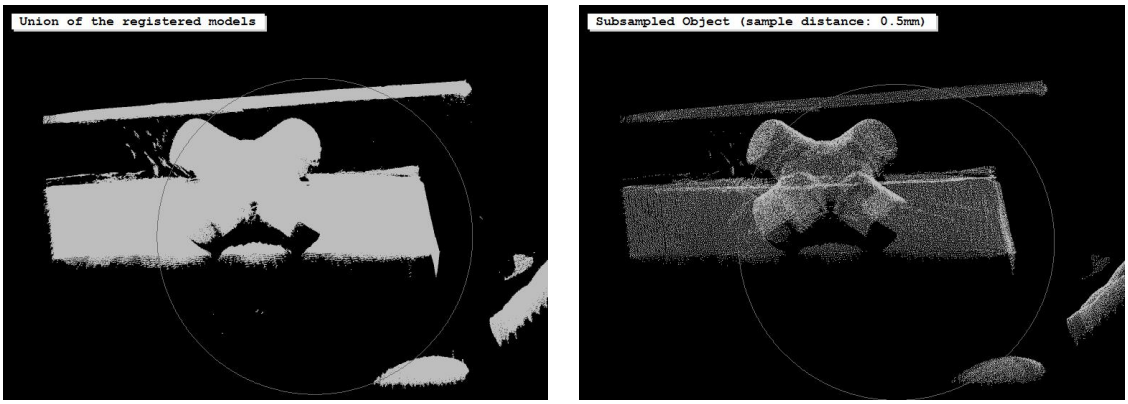


Figure 2.41: The 3D object models (left) united into a single 3D object model and (right) subsampled and smoothed.

```
triangulate_object_model_3d (SmoothObject3D, 'greedy', [], [], Surface3D, \
                             Information)
connection_object_model_3d (Surface3D, 'mesh', 1, ObjectModel3DConnected)
```

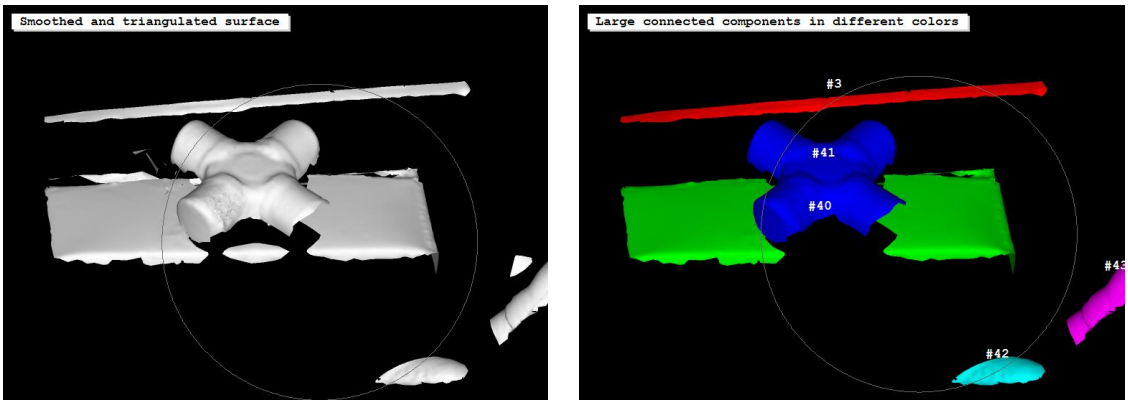


Figure 2.42: 3D object model (left) after the triangulation and (right) after the segmentation into connected components.

To separate the component that should be used as model for the surface-based 3D matching from the background, `select_object_model_3d` is applied with common shape features (see figure 2.43).

```
select_object_model_3d (ObjectModel3DConnected, ['has_triangles', \
                        'num_triangles'], 'and', [1, 2000], [1, 100000], \
                        ObjectModel3DSelected)
select_object_model_3d (ObjectModel3DSelected, ['central_moment_2_x', \
                        'central_moment_2_y'], 'and', [150, 200], [400, \
                        230], ObjectModel3DCross)
```

Alternatively, instead of this workflow to obtain a surface, you can try and use `fuse_object_model_3d`.

Step 3: Use the surface model for a surface-based 3D matching

The obtained 3D object model is now used for a surface-based 3D matching. The matching is demonstrated first on the data that was used to create the surface model, i.e., on the initial views on the model object that were obtained by a 3D sensor (see figure 2.44). Then, a matching is applied in multi-view stereo images representing different views on a set of universal joints (see figure 2.45). For detailed information about surface-based 3D matching and multi-view stereo reconstruction, please refer to section 4.3 on page 104 and section 5.4 on page 139.

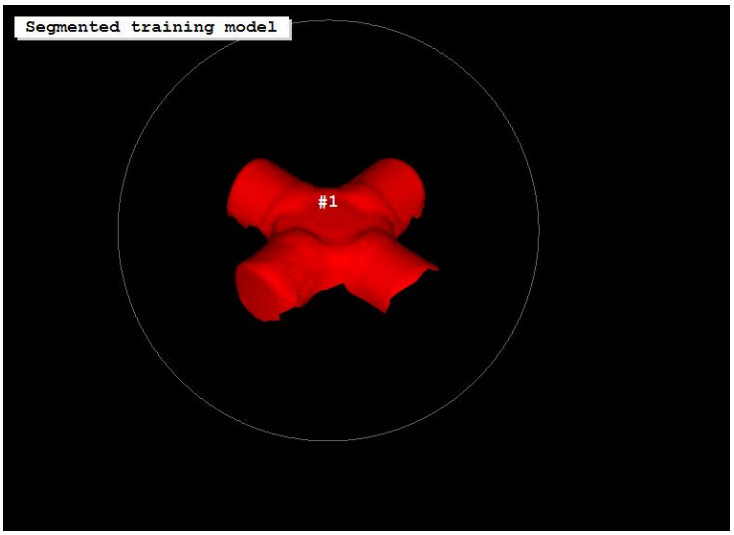


Figure 2.43: 3D object model selected to be used as surface model for a surface-based 3D matching.

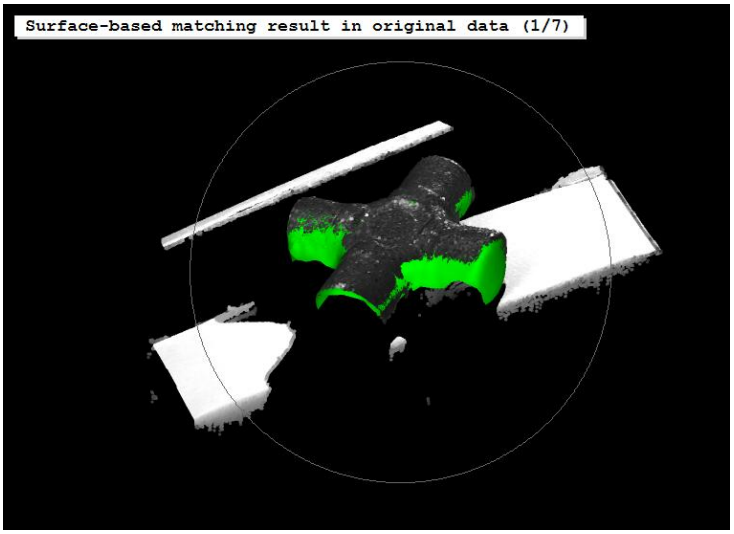


Figure 2.44: Result of surface-based 3D matching in one of the views of the training data.

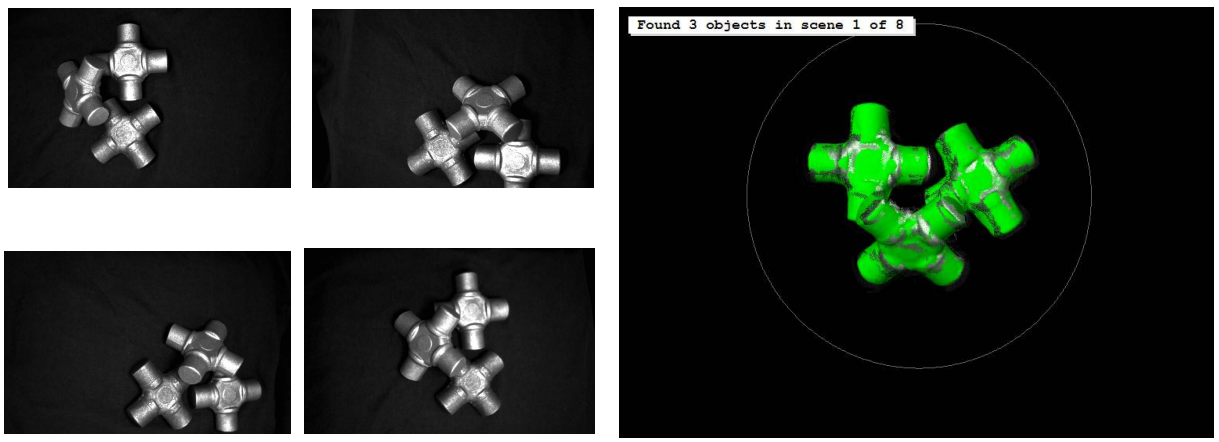


Figure 2.45: Result of surface-based 3D matching in a scene obtained from multi-view stereo images.

2.3.6 Visualizing 3D Object Models

For the visualization of 3D object models, different operators and procedures are available.

- Visualization of individual 3D object models with disp_object_model_3d.
  - Visualization of individual 3D object models without the need to set camera parameters and pose.
  - disp_object_model_3d provides an easy to use interface for the visualization of one or a small number of 3D object models.
- Visualization of multiple 3D object models, a so called '3D scene', with display_scene_3d.
  - Best choice if multiple 3D object models should be displayed.
  - Multiple cameras can be defined to view the 3D scene from different positions.
  - Labels can be attached to specific 3D points.
  - Smooth display of dynamic scenes since the handle can manage the caching of the objects on the graphics card.
- Interactive visualization of 3D object models with the procedure visualize_object_model_3d.
  - Visualization of 3D object models with the possibility to interactively change their position and orientation.

Note that the visualization of 3D object models requires OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit. If these requirements are not fulfilled, e.g., because of the use of Windows Remote Desktop or SSH forwarding, a compatibility mode with less requirements but lower quality is automatically enabled.

2.3.6.1 Visualization of individual 3D object models

To simply display 3D object models within a window on the screen, the operator disp_object_model_3d can be used. Here, camera parameters and pose can be specified to define the view on the objects. In addition, with the operator get_disp_object_model_3d_info you can then query specific information for each position in the displaying window. In particular, for each row and column pair, you can query the index of the 3D object model that is displayed in this pixel or the depth of the 3D object model in this pixel (measured from the camera to the projected point).

The HDevelop example program disp_object_model_3d.hdev shows how to use disp_object_model_3d, e.g., to display three 3D object models with different colors:

```
disp_object_model_3d (WindowHandle, ObjectModel3DIDs, CameraParam, ObjPoses, \
                      'colored', 12)
```

Note that the simplest way to call disp_object_model_3d just requires the handles of the 3D object model and the window, in which it should be displayed:

```
disp_object_model_3d (WindowHandle, ObjectModel3D, [], [], [], [])
```

In this case, the pose and the camera parameters are determined internally such that the complete object is visible.

If a view like that generated by disp_object_model_3d should not only be displayed but stored in an image, you can use render_object_model_3d instead.

If the contours of a 3D object model for a specific view on the object are needed, project_object_model_3d can be used.

2.3.6.2 Visualization of multiple 3D object models using handles

Another way of displaying 3D data are the 3D scene operators.

The HDevelop example program display_scene_3d.hdev shows how to create and display a 3D scene.

First a scene is created via create_scene_3d, which enables the visualization of 3D objects.

```
create_scene_3d (Scene3D)
```

A scene needs at least one camera. A camera can be added to the scene with the operator add_scene_3d_camera. Optionally, the pose of the camera can be set.

```
add_scene_3d_camera (Scene3D, CameraParam, CameraIndex)
set_scene_3d_camera_pose (Scene3D, CameraIndex, [0, 0, -0.4, 0, 0, 0, 0])
```

Optionally, a light source can be selected for the scene. The light source can be added to the scene with the operator add_scene_3d_light and its properties, e.g., the RGB color of its diffuse part, can be set with set_scene_3d_light_param.

```
add_scene_3d_light (Scene3D, [1.0, 1.0, 1.0], 'point_light', LightIndex)
set_scene_3d_light_param (Scene3D, LightIndex, 'diffuse', [0.8, 0.8, 0.8])
set_scene_3d_light_param (Scene3D, LightIndex, 'ambient', [0.2, 0.2, 0.2])
```

Now objects can be added to the scene using add_scene_3d_instance. This operator requires a 3D object model and its pose relative to the coordinate system of the 3D scene. Note that the pose of the 3D scene in the world coordinate system can be set with set_scene_3d_to_world_pose.

```
add_scene_3d_instance (Scene3D, ObjectModel3D, Pose1, Instance1Index)
set_scene_3d_instance_param (Scene3D, Instance1Index, 'alpha', 0.7)
add_scene_3d_instance (Scene3D, ObjectModel3D, Pose2, Instance2Index)
set_scene_3d_instance_param (Scene3D, Instance2Index, 'color', '#f28f26')
```

Optionally, labels can be attached to specific 3D points.

```
add_scene_3d_label (Scene3D, 'Instance 1', Pose1[0:2], 'bottom_left', \
                    'point', Instance1LabelID)
add_scene_3d_label (Scene3D, 'Instance 2', Pose2[0:2], 'top', 'point', \
                    Instance2LabelID)
```

The 3D objects can then be displayed using the operator display_scene_3d.

```
display_scene_3d (WindowHandle, Scene3D, CameraIndex)
```

An image of the scene can also be rendered with render_scene_3d.

```
render_scene_3d (Image, Scene3D, CameraIndex)
```

2.3.6.3 Interactive visualization of 3D object models

A convenient way to visualize 3D object models such that the object can be moved and rotated interactively is provided by the external procedure visualize_object_model_3d.

Chapter 3

Metric Measurements in a Specified Plane With a Single Camera

In HALCON it is easy to obtain undistorted measurements in world coordinates from images. In general, this can only be done if two or more images of the same object are taken at the same time with cameras at different spatial positions. This is the so-called *stereo* approach (see chapter 5 on page 117).

In industrial inspection, we often have only one camera available and time constraints do not allow us to use the expensive process of finding corresponding points in the stereo images (the so-called *stereo matching* process).

Nevertheless, it is possible to obtain measurements in world coordinates for objects acquired through telecentric lenses and objects that lie in a known plane, e.g., on an assembly line, for pinhole cameras. Both of these tasks can be solved by intersecting an optical ray (also called line of sight) with a plane.

With this, it is possible to measure objects that lie in a plane, even when the plane is tilted with respect to the optical axis. The only prerequisite is that the camera has been *calibrated*. In HALCON, the calibration process is very easy as can be seen in the example described in section 3.1, which introduces the operators that are necessary for the calibration process.

The easiest way to perform the calibration is to use the HALCON standard calibration plates. You just need to take a few images of a calibration plate (see figure 3.1 for examples), where in one image the calibration plate has been placed directly on the measurement plane.

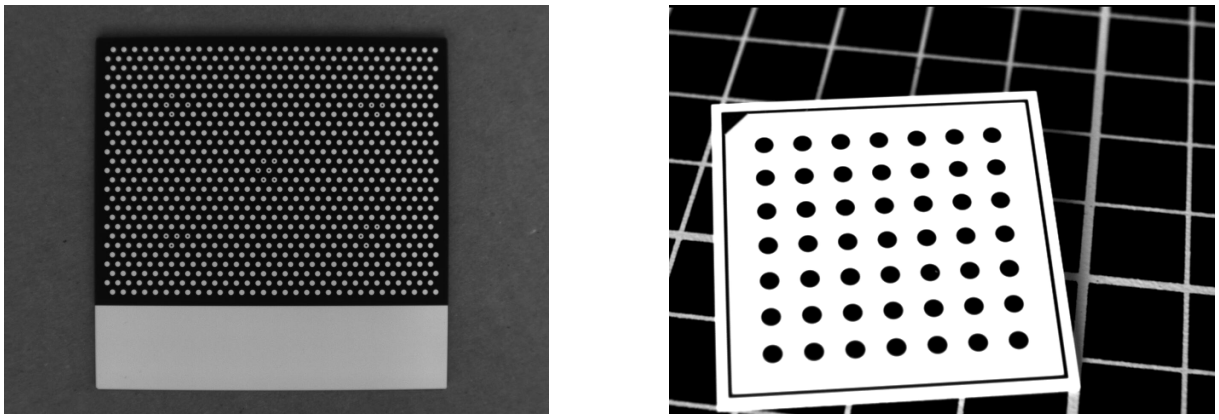


Figure 3.1: The HALCON calibration plates.

The sections that follow the example show how to

- calibrate a camera (section 3.2 on page 61),
- transform image into world coordinates (section 3.3 on page 76), and
- rectify images, i.e., remove perspective and/or lens distortions from an image (section 3.4 on page 80).

Finally, we briefly look at the case of inspecting a non-planar object (section 3.5 on page 87).

3.1 First Example

The HDevelop example %HALCONEXAMPLES%\solution_guide\3d_vision\camera_calibration_multi_image.hdev shows how easy it is to calibrate a camera and use the calibration results to transform measurements into 3D world coordinates.

First, we specify general parameters for the calibration.

```
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], StartCamPar)
set_calib_data_calib_object (CalibDataID, 0, 'calplate_80mm.cpd')
```

Then, images of the calibration plate are read. With the operator find_calib_object, the calibration plate is searched, the contours and centers of the marks are extracted, and the pose of the calibration plate is estimated. The obtained information is stored in the calibration data model.

```
for I := 1 to NumImages by 1
    read_image (Image, ImgPath + 'calib_image_' + I$'02d')
    find_calib_object (Image, CalibDataID, 0, 0, I, [], [])
endfor
```

Now, we perform the actual calibration with the operator calibrate_cameras.

```
calibrate_cameras (CalibDataID, Errors)
```

Afterwards, we can access the calibration results, i.e., the internal camera parameters and the pose of the calibration plate in a reference image.

```
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
get_calib_data (CalibDataID, 'calib_obj_pose', [0, 1], 'pose', Pose)
```

This pose is used as the external camera parameters, i.e., the pose of the 3D world coordinate system in camera coordinates. In the example, the world coordinate system is located on the ruler (see figure 3.2). To compensate for the thickness of the calibration plate, the pose is moved by the corresponding value.

```
set_origin_pose (Pose, 0, 0, 0.002, Pose)
```

Now, we perform the measurement. For that, we have acquired an additional image of the ruler without occlusions by the calibration plate.

```
read_image (Image, ImgPath + 'ruler')
gen_measure_rectangle2 (690, 680, rad(-0.25), 480, 8, 1280, 960, 'bilinear', \
                        MeasureHandle)
measure_pairs (Image, MeasureHandle, 0.5, 5, 'all', 'all', RowEdgeFirst, \
               ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, \
               ColumnEdgeSecond, AmplitudeSecond, IntraDistance, \
               InterDistance)
Row := (RowEdgeFirst + RowEdgeSecond) / 2.0
Col := (ColumnEdgeFirst + ColumnEdgeSecond) / 2.0
```

With the internal and external camera parameters, the measurement results are transformed into 3D world coordinates with the operator image_points_to_world_plane.

```
image_points_to_world_plane (CamParam, Pose, Row, Col, 'mm', X1, Y1)
```

3.1.1 Single Image Calibration

In general, a correct camera calibration needs multiple images as described in the section "How to take a set of suitable images?" in the chapter reference "Calibration". If only a single image is used, correct measurements are restricted to the plane of the calibration plate. Measurements outside of the calibration plane,

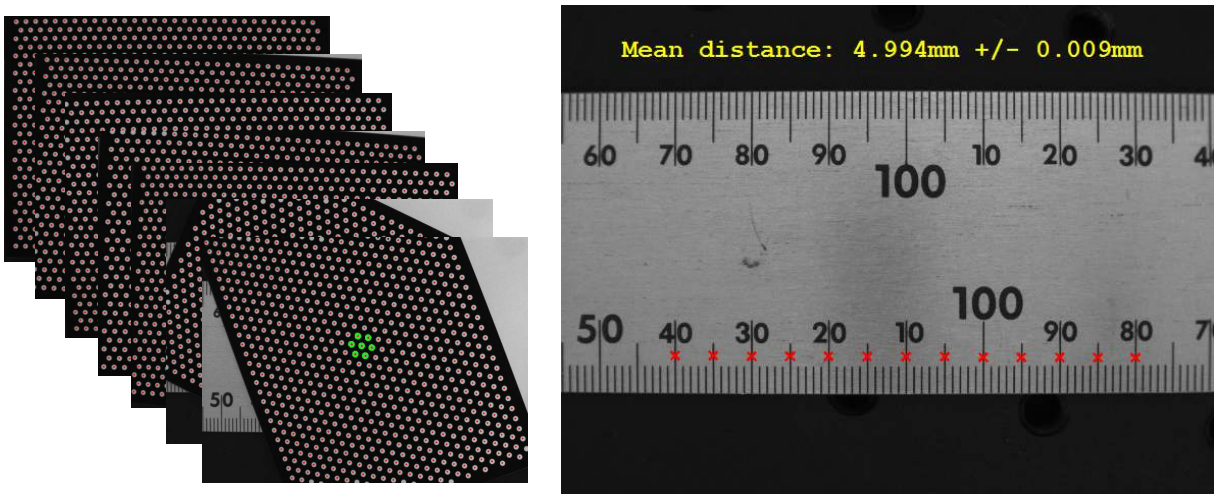


Figure 3.2: After the calibration, marks on the ruler are measured and the results are transformed into 3D world coordinates with the calibration results.

e.g., when using set_origin_pose like in the example %HALCONEXAMPLES%\solution_guide\3d_vision\camera_calibration_multi_image.hdev, will be afflicted with a systematic error.

However, if the focal length is roughly known, it can be kept fixed during the calibration process with set_calib_data.

```
set_calib_data (CalibHandle, 'camera', 'general', \
                'excluded_settings', 'focus')
```

By doing this, the systematic error can be kept to a minimum if the start value for the focal length is close to the real focal length.

The HDevelop example %HALCONEXAMPLES%\solution_guide\3d_vision\camera_calibration_single_image.hdev shows a measurement after a calibration with only a single image.

3.2 3D Camera Calibration

This section describes the process of 3D camera calibration in detail. Note that you can easily calibrate your camera with the help of HDevelop's Calibration Assistant (see the HDevelopUser's Guide, section 7.2 on page 184 for more details).

The calibration process consists of three steps:

- the **preparation**,
- the **calibration**, and
- the **access to the results**.

After accessing the results, you can store them.

Preparations Before Performing the Calibration

All information for the calibration is passed in the so-called *calibration data model*. In particular, you

- create the model and specify basic information like the number of cameras to calibrate (section 3.2.1),
- specify initial values for the internal camera parameters (section 3.2.2),

- describe the calibration object (section 3.2.3 on page 67),
- observe the calibration object in multiple poses (images) and store the extracted information (section 3.2.4 on page 71), and
- optionally restrict the calibration to certain parameters, keeping the others fixed (section 3.2.5 on page 72).

Performing the Actual Calibration

In HALCON, you calibrate single or multiple cameras with the operator `calibrate_cameras`, which needs the calibration data model as input (see section 3.2.6 on page 72 for more information).

Accessing the Results of the Calibration

`calibrate_cameras` again stores its results, in particular the calibrated camera parameters and poses of the calibration objects, in the calibration data model. You can access them (and all other calibration parameters) with the operator `get_calib_data` (see section 3.2.7 on page 72).

3.2.1 Creating the Calibration Data Model

You create a calibration data model with the operator `create_calib_data`, specifying the number of cameras and the numbers of calibration objects. When using a single camera, you typically use a single calibration object as well.

```
create_calib_data ('calibration_object', 1, 1, CalibDataID)
```

Then, you proceed to

- specify initial values for the internal camera parameters (section 3.2.2) and
- describe the calibration object (section 3.2.3 on page 67).

3.2.2 Specifying Initial Values for the Internal Camera Parameters

You set internal camera parameters with the operator `set_calib_data_cam_param`.

```
gen_cam_par_area_scan_division (0.012, 0, 0.00000375, 0.00000375, 640, 480, \
                                1280, 960, StartCamPar)
set_calib_data_cam_param (CalibDataID, 0, [], StartCamPar)
```

Besides the calibration data model, the operator needs the following parameters as input:

- `CameraIdx`: the index of the camera (*0* for a single camera),
- `CameraType`: the camera type, and
- `CameraParam`: a tuple with values for the internal camera parameters.

Below, you find

- a list of the supported camera types and their parameters (section 3.2.2.1),
- tips how to choose the suitable distortion model (section 3.2.2.2 on page 64),
- special tips for area scan cameras (section 3.2.2.3 on page 64), and
- special tips for line scan cameras (section 3.2.2.4 on page 65).

3.2.2.1 Camera Type and Camera Parameters

The values of `CameraParam` depend on the used camera, lens type, and the selected model for the lens distortion. In addition to the internal camera parameters, the `CameraType`, the `ImageWidth`, and the `ImageHeight` must be given. Please, refer to section 2.2 on page 25 for the description of the underlying camera models and to section 3.2.2.2 on page 64 for tips that help you to decide which distortion model to use. The values of `CameraParam` are set as follows:

'area_scan_division'
['area_scan_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_polynomial'
['area_scan_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_division'
['area_scan_tilt_division', Focus, Kappa, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_polynomial'
['area_scan_tilt_polynomial', Focus, K1, K2, K3, P1, P2, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_image_side_telecentric_division'
['area_scan_tilt_image_side_telecentric_division', Focus, Kappa, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_image_side_telecentric_polynomial'
['area_scan_tilt_image_side_telecentric_polynomial', Focus, K1, K2, K3, P1, P2, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_telecentric_division'
['area_scan_telecentric_division', Magnification, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_telecentric_polynomial'
['area_scan_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_bilateral_telecentric_division'
['area_scan_tilt_bilateral_telecentric_division', Magnification, Kappa, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_bilateral_telecentric_polynomial'
['area_scan_tilt_bilateral_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_object_side_telecentric_division'
['area_scan_tilt_object_side_telecentric_division', Magnification, Kappa, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_tilt_object_side_telecentric_polynomial'
['area_scan_tilt_object_side_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_hypercentric_division'
['area_scan_hypercentric_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'area_scan_hypercentric_polynomial'
['area_scan_hypercentric_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

'line_scan_division'
['line_scan_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

'line_scan_polynomial'
['line_scan_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

'line_scan_telecentric_division'
['line_scan_telecentric_division', Magnification, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

'line_scan_telecentric_polynomial'
['line_scan_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]

Also note that for all operators that use internal camera parameters as input, the parameter values are checked as to whether they fulfill certain restrictions. However, these restrictions may differ slightly for some operators. Please see the chapter "Calibration ▷ Multi-View" for details about the respective restrictions.

For most of the internal camera parameters, initial values can be determined from the specifications of the CCD sensor and the lens. The following sections contain additional tips on how to find suitable initial values for

- area scan cameras (section 3.2.2.3) and
- line scan cameras (section 3.2.2.4).

3.2.2.2 Which Distortion Model to Use

For area scan cameras, two distortion models can be used: The division model and the polynomial model. The division model uses one parameter to model the radial distortions while the polynomial model uses five parameters to model radial and decentering distortions (see section 2.2.2 on page 26).

The advantages of the division model are that the distortions can be applied faster, especially the inverse distortions, i.e., if world coordinates are projected into the image plane. Furthermore, if only few calibration images are used or if the field of view is not covered sufficiently, the division model typically yields more stable results than the polynomial model. The main advantage of the polynomial model is that it can model the distortions more accurately because it uses higher order terms to model the radial distortions and because it also models the decentering distortions. Note that the polynomial model cannot be inverted analytically. Therefore, the inverse distortions must be calculated iteratively, which is slower than the calculation of the inverse distortions with the (analytically invertible) division model.

Typically, the division model should be used for the calibration. If the accuracy of the calibration is not high enough, the polynomial model can be used. But note that the calibration sequence used for the polynomial model must provide a complete coverage of the area in which measurements will later be performed. The distortions may be modeled inaccurately outside of the area that was covered by the calibration plate. This holds for the image border as well as for areas inside the field of view that were not covered by the calibration plate.

3.2.2.3 Tips for Area Scan Cameras

If you face problems using a camera model and you suspect that the sensor of your camera may be tilted (e.g., due to a defect), we suggest to check if a calibration using a camera model for object-side telecentric tilt lenses (*'area_scan_tilt_object_side_telecentric_division'* or *'area_scan_tilt_object_side_telecentric_polynomial'*) leads to better results.

In the following, some hints for the determination of the initial values for the internal camera parameters of an **area scan camera** are given:

Focus f: Focal length of the lens (only for lenses that perform a perspective projection on the object side of the lens). The initial value is the nominal focal length of the used lens, e.g., 0.008m.

Magnification m: Magnification of the lens (only for lenses that perform a telecentric projection on the object side of the lens). The initial value is the nominal magnification of the used telecentric lens (the image size divided by the object size), e.g., 0.2.

Kappa (κ): Use 0.0 as initial value (only for the division model).

Or:

K1, K2, K3, P1, P2: Use the initial value 0.0 for each of the five coefficients (only for the polynomial model).

ImagePlaneDist: Distance of the tilted image plane from the perspective projection center (only tilt lenses that perform a perspective projection on the image side of the lens), e.g., 0.02m. The initial value is the distance of the exit pupil of the lens to the image plane. The exit pupil is the (virtual) image of the aperture stop (typically the diaphragm), as viewed from the image side of the lens. Typical values are in the order of a few centimeters to very large values if the lens is close to being image-side telecentric.

Tilt and Rot: The tilt angle τ $(0° \leq \tau \leq 90°)$ describes the angle by which the optical axis is tilted with respect to the normal of the sensor plane. The rotation angle ρ $(0° \leq \rho < 360°)$ describes the direction in which the optical axis is tilted. These parameters are only used if a tilt lens is part of the camera setup. These angles are typically roughly known based on the considerations that led to the use of the tilt lens or can be read off from the mechanism by which the lens is tilted.

Sx and Sy: Scale factors. This corresponds to the horizontal and vertical distance between two neighboring cells on the sensor. Since in most cases the image signal is sampled line-synchronously, S_y is determined by the dimension of the sensor and does not need to be estimated by the calibration process.
The initial values depend on the dimensions of the used chip of the camera. See the technical specification of your camera for the actual values. Attention: These values increase if the image is subsampled!

Cx and Cy: Initial values for the coordinates of the principal point are the coordinates of the image center, i.e., half the image width and half the image height. Notice: The values of Cx and Cy decrease if the image is subsampled! Appropriate initial values are, for example:

	Full image (640*480)	Subsampling (320*240)
Cx	320.0	160.0
Cy	240.0	120.0

ImageWidth and ImageHeight: These two parameters are set by the used frame grabber and therefore are not calibrated. Appropriate initial values are, for example:

	Full image (640*480)	Subsampling (320*240)
ImageWidth	640	320
ImageHeight	480	240

3.2.2.4 Tips for Line Scan Cameras

In the following, some hints for the determination of the initial values for the internal camera parameters of a **line scan camera** are given:

Focus f: The initial value for the focal length (only for pinhole lenses) is the nominal focal length of the used lens, e.g., 0.035 m.

Magnification m: The initial value for the magnification (only for telecentric lenses) is the nominal magnification of the used lens, e.g., 0.2.

Kappa (κ): Use 0.0 as initial value (only for the division model).

Or:

K1, K2, K3, P1, P2:
Use the initial value 0.0 for each of the five coefficients (only for the polynomial model). Note that the parameters P1 and P2 are highly correlated with other parameters in the camera model. Therefore, they typically cannot be determined reliably and should be excluded from the calibration by calling

```
set_calib_data (CalibDataID, 'camera', 'general', \
                'excluded_settings', 'poly_tan_2')
```

Sx:
The horizontal distance between two neighboring sensor elements can be taken from the technical specifications of the camera. Typical initial values are $7{\cdot}10^{-6}$ m, $10{\cdot}10^{-6}$ m, and $14{\cdot}10^{-6}$ m. Notice: The value of Sx increases if the image is subsampled. Note also that Sx will *not* be calibrated for line scan cameras because Sx and Focus (for pinhole lenses) and Sx and Magnification (for telecentric lenses) cannot be determined simultaneously for line scan cameras.

Sy:
The size of a cell in the direction perpendicular to the sensor line can also be taken from the technical specifications of the camera. Typical initial values are $7{\cdot}10^{-6}$ m, $10{\cdot}10^{-6}$ m, and $14{\cdot}10^{-6}$ m. Notice: The value of Sy increases if the image is subsampled. Note also that Sy will *not* be calibrated for line scan cameras because it cannot be determined separately from the parameter Cy.

Cx:
The initial value for the x-coordinate of the principal point is half the image width. Notice: The values of Cx decreases if the image is subsampled! Appropriate initial values are:

Image width:	1024	2048	4096	8192
Cx:	512	1024	2048	4096

Cy:
Normally, the initial value for the y-coordinate of the principal point can be set to 0.

ImageWidth and ImageHeight:
These two parameters are determined by the used frame grabber and therefore are not calibrated.

Vx, Vy, Vz: The initial values for the x-, y-, and z-component of the motion vector depend on the image acquisition setup. Assuming a fixed camera that looks perpendicularly onto a conveyor belt, such that the y-axis of the camera coordinate system is anti-parallel to the moving direction of the conveyor belt (see figure 3.3 on page 68), the initial values are Vx = Vz = 0. The initial value for Vy can then be determined, e.g., from a line scan image of an object with known size (e.g., calibration plate or ruler):

$$\begin{aligned} V_y &= L[m]/L[row] \\ \text{with:} & \\ L[m] &= \text{Length of the object in object coordinates [meter]} \\ L[row] &= \text{Length of the object in image coordinates [rows]} \end{aligned}$$

If, compared to the above setup, the camera is rotated 30 degrees around its optical axis, i.e., around the z-axis of the camera coordinate system (figure 3.4 on page 68), the above determined initial values must be changed as follows:

$$\begin{aligned} V_{zx} &= \sin(30°) \cdot V_y \\ V_{zy} &= \cos(30°) \cdot V_y \\ V_{zz} &= V_z = 0 \end{aligned}$$

If, compared to the first setup, the camera is rotated -20 degrees around the x-axis of the camera coordinate system (figure 3.5 on page 69), the following initial values result:

$$\begin{aligned} V_{xx} &= V_x = 0 \\ V_{xy} &= \cos(-20°) \cdot V_y \\ V_{xz} &= \sin(-20°) \cdot V_y \end{aligned}$$

The quality of the initial values for Vx, Vy, and Vz are crucial for the success of the whole calibration. If they are not accurate enough, the calibration may fail. Section 3.2.10.1 on page 76 provides you with tips what to do in this case.
Note that for telecentric line scan cameras, the value of V_z has no influence on the image position of 3D points and therefore cannot be determined. Consequently, V_z is not optimized and left at its initial value for telecentric line scan cameras. Therefore, the initial value of V_z should be set to 0.

3.2.3 Describing the Calibration Object

With the operator `set_calib_data_calib_object` you specify the needed information about the calibration object.

If you are using a HALCON calibration plate, the name of the corresponding description file is sufficient.

```
set_calib_data_calib_object (CalibDataID, 0, 'calplate_80mm.cpd')
```

How to obtain a HALCON calibration plate is explained in section 3.2.3.1.

However, you can also use your own calibration object. Then, you pass the coordinates of the markers instead of the file name to the operator (see section 3.2.3.2 on page 70 for more information).

3.2.3.1 How to Obtain a Suitable Calibration Plate

The simplest method to determine the camera parameters of a CCD camera is to use a HALCON calibration plate (see figure 3.6 on page 69 for examples). In this case, the whole process of finding the calibration plate, extracting the calibration marks, and determining the correspondences between the extracted calibration marks and the respective 3D world coordinates can be carried out automatically. Even more important, these calibration plates are highly accurate, down to 1 μm or below, which is a prerequisite for high accuracy applications. Therefore, we recommend to obtain such a calibration plate from the local distributor from which you purchased HALCON.

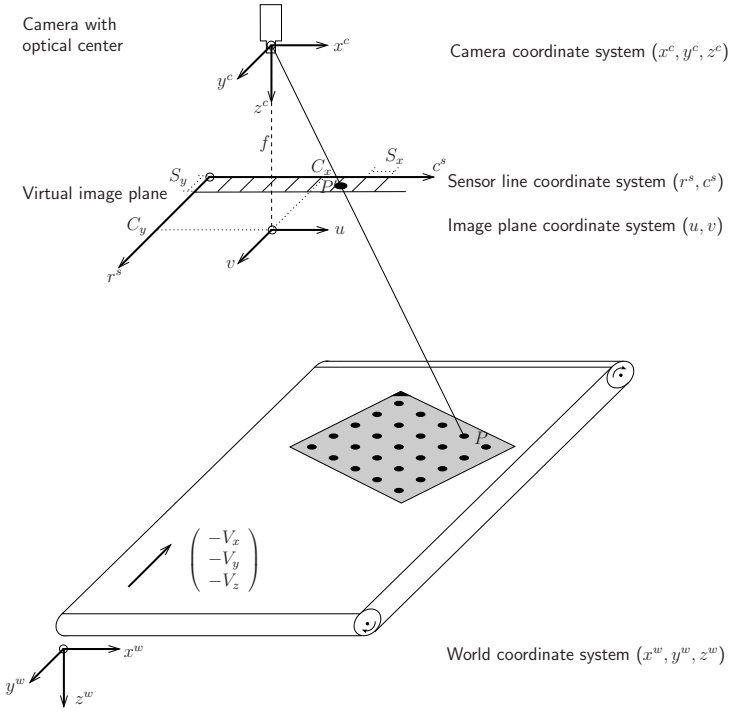


Figure 3.3: Line scan camera looking perpendicularly onto a conveyor belt.

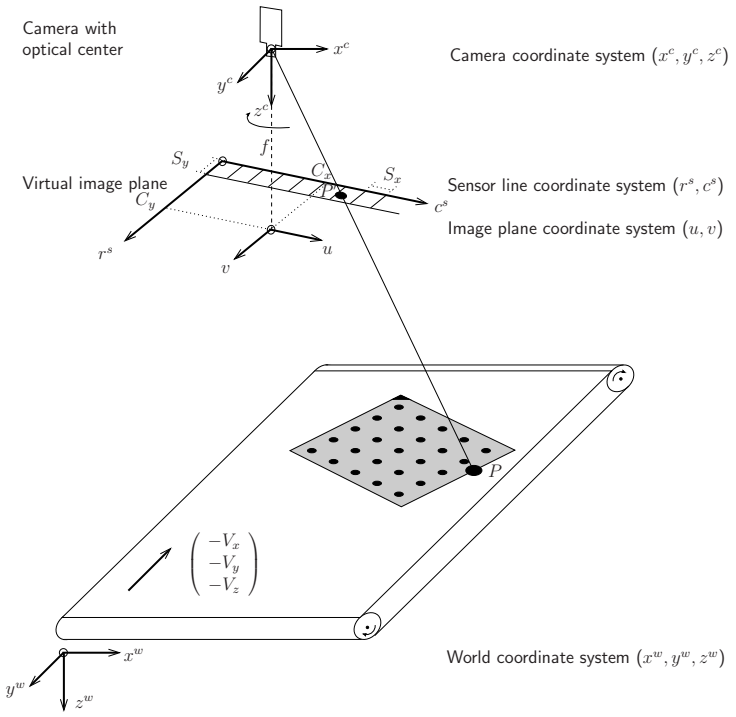


Figure 3.4: Line scan camera rotated around the optical axis.

Two types of HALCON calibration plates are supported. In particular, calibration plates with hexagonally arranged marks and calibration marks with rectangularly arranged marks are available. The calibration plates with hexagonally arranged marks are introduced with HALCON 12 and are recommended for most applications, as they provide the following advantages compared to the calibration plates with rectangularly arranged marks:

- The calibration plates with rectangularly arranged marks must be completely visible in the images whereas plates with hexagonally arranged marks may protrude beyond the rim of the image. Thus, with the latter

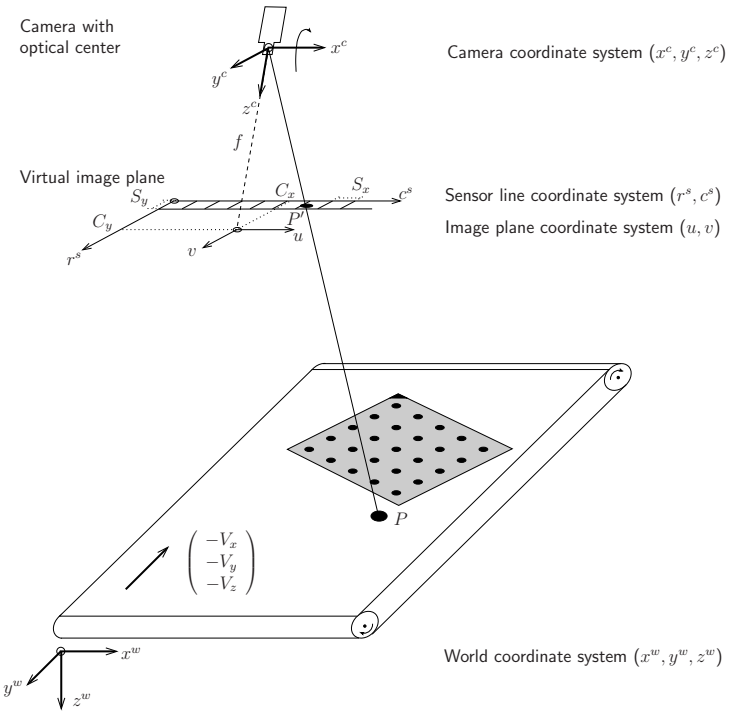


Figure 3.5: Line scan camera rotated around the x-axis.

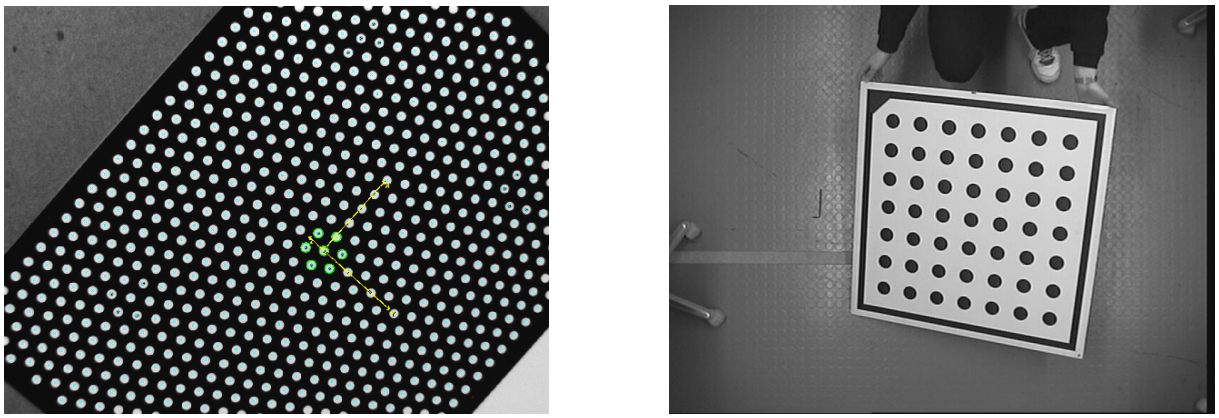


Figure 3.6: HALCON calibration plates of different materials and sizes: (a) with hexagonally arranged marks and (b) with rectangularly arranged marks.

less care must be taken when placing the calibration plate in the images. The acquisition of the calibration images becomes faster and more convenient without a loss of robustness.

- As calibration plates with hexagonally arranged marks contain a significantly larger number of calibration marks and can protrude beyond the rim of the image, less images are needed to get a comparable calibration result.

The calibration plates are available in different materials (ceramics for front light, glass for back light applications, aluminum for very large calibration plates) and sizes. Thus, you can choose the one that is optimal for your application. Detailed information about the available materials, sizes, and the accuracy can be obtained from your distributor.

The main differences between the calibration plates with hexagonally arranged marks and the calibration plates with rectangularly arranged marks are:

- *Different arrangement of the marks:* For calibration plates with hexagonally arranged marks the marks are

arranged in a hexagonal lattice such that each mark has six equidistant neighbors whereas calibration plates with rectangularly arranged marks are arranged in a rectangular grid with equidistant rows and columns.

- *Different number of contained marks:* Calibration plates with hexagonally arranged marks have a significantly larger number of marks than calibration plates with rectangularly arranged marks.
- *Different special patterns:* The calibration plates with hexagonally arranged marks contain one to five unique finder patterns, i.e., special mark hexagons (each consisting of a mark and its six neighbors) where either four or six marks contain a hole. To find the calibration plate in the image and estimate the pose of it relative to the observing camera using `find_calib_object`, at least one finder pattern must be completely visible in the image. The calibration plates with rectangularly arranged marks contain a triangular orientation mark and a surrounding frame. Here, the complete calibration plate must be visible to find the plate and estimate its pose.
- *Different origin:* The coordinate system of the calibration plate with hexagonally arranged marks is located at the center of the central mark of the first finder pattern whereas the coordinate system of the calibration plate with rectangularly arranged marks is located in the barycenter of all marks.
- *Slightly different handling:* Due to the differences described above, the rules for the acquisition of calibration images differ in some points. First of all, for calibration plates with hexagonally arranged marks less calibration images are needed and the calibration plates do not have to be completely visible in the image. Please refer to the section "How to take a set of suitable images?" in the chapter reference "Calibration" for further details.

Each calibration plate comes with a description file. Place this file in the subdirectory `calib` of the directory where you installed HALCON. Then, you can use its file name directly in the operator `caltab_points` (see section 3.2.4). Note that the description files for calibration plates with hexagonally arranged marks have the file extension *'.cpd'* and those of the calibration plates with rectangularly arranged marks have the file extension *'.descr'*. Calibration plates with rectangularly arranged marks always have dark marks on a light background, whereas calibration plates with hexagonally arranged marks usually have light marks on a dark background. Nevertheless, also calibration plates with dark hexagonally arranged marks on a light background are available, e.g., for back light applications. The corresponding description files are indicated by the suffix *'dark_on_light'*.

For test purposes, you can create a calibration plate yourself with the operator `create_caltab` for calibration plates with hexagonally arranged marks and with the operator `gen_caltab` for calibration plates with rectangularly arranged marks. Print the resulting PostScript file and mount it on a planar and rigid surface, e.g., an aluminum plate or a solid cardboard. If you do not mount the printout on a planar and rigid surface, you will not get meaningful results by HALCON's camera calibration as the operators `create_caltab` and `gen_caltab` assume that the calibration marks lie within a plane. Such self-made calibration plates should only be used for test purposes as you will not achieve the high accuracy that can be obtained with an original HALCON calibration plate. Note that the printing process is typically not accurate enough to create calibration plates smaller than 16 cm for calibration plates with hexagonally arranged marks and 3 cm for calibration plates with rectangularly arranged marks.

3.2.3.2 Using Your Own Calibration Object

With HALCON, you are not restricted to using a planar calibration object like the HALCON calibration plate. You can use a 3D calibration object or even arbitrary characteristic points (natural landmarks). The only requirement is that the 3D world position of the model points is known with high accuracy.

Then, you simply pass the 3D coordinates of all points (markers) of the calibration object as a tuple in the parameter `CalibObjDescr` of `set_calib_data_calib_object`. All x, y, and z coordinates of all points must be packed sequentially in the tuple in the form `[X, Y, Z]`.

Note however, that if you use your own calibration object, you cannot use the operator `find_calib_object` anymore. Instead, you must determine the 2D locations of the model points and the correspondence to the 3D points by yourself.

3.2.4 Observing the Calibration Object in Multiple Poses (Images)

The main input data for the calibration are the so-called observations. For this, the calibration object is placed in different poses. For each pose, the camera acquires an image. In this image, the markers of the calibration object are extracted, and their (pixel) coordinates, together with the indices of camera, calibration object, and calibration object pose and with a tuple containing the indices of the corresponding markers, are stored in the calibration data model.

If you are using a standard HALCON calibration plate, you can apply the extraction of the coordinates with the operator `find_calib_object`, which automatically stores the obtained information, including the coordinates of the markers and the list of marker correspondences, in the calibration data model.

```
for I := 1 to NumImages by 1
    read_image (Image, ImgPath + 'calib_image_' + I$'02d')
    find_calib_object (Image, CalibDataID, 0, 0, I, [], [])
endfor
```

If you are using your own calibration object, you must extract its markers and determine the correspondences by yourself and then store the information into the calibration data model with `set_calib_data_observ_points`.

The following sections contain

- recommendations for acquiring suitable calibration images (section 3.2.4.1) and
- additional information about the extraction of the calibration marks of a standard HALCON calibration plate (section 3.2.4.2).

3.2.4.1 Acquiring Calibration Images

If you want to achieve accurate results, please follow the recommendations given in the section “How to take a set of suitable images?” in the chapter reference “Calibration”. It gives guidance regarding the placement of the calibration plates, the camera setup and the image properties.

Note that a good calibration result can be obtained only for a distribution of the calibration marks within the full field of view of the camera. You can imagine the part of the 3D space that corresponds to the field of view as a calibration volume like shown in figure 3.7. There, two poses of calibration plates and the positions of their calibration marks, when seen from different views, are illustrated. You can see, e.g., in the view from side 1, that large parts are not covered by marks. To get a full distribution of the marks and thus enable a good calibration result, you have to place the calibration plates in your other images so that the empty parts of the calibration volume are minimized for all views. Be aware that when having very small calibration plates (compared to the field of view), this means that it may be necessary to use significantly more than the recommended number of calibration images.

If only one image is used for the calibration process or if the orientations of the calibration plate do not vary over the different calibration images, it is not possible to determine both the focal length and the pose of the camera correctly; only the ratio between the focal length and the distance between calibration plate and camera can be determined in this case. Nevertheless, it is possible to measure world coordinates in the plane of the calibration plate. When measuring outside the plane onto which the calibration plate was placed, you will get systematic errors (see section 3.1.1 on page 60).

The accuracy of the resulting world coordinates depends — apart from the measurement accuracy in the image — very much on the number of images used for the calibration process. The more images (with significantly different calibration plate poses) are used, the more accurate results will be achieved.

3.2.4.2 Extracting the Marks from a HALCON Calibration Plate

The operator `find_calib_object` searches for the calibration plate, determines the image coordinates of the calibration marks with high precision, and stores the results in a calibration data model. The calibration mark contours, can be accessed with the operator `get_calib_data_observ_contours`.

If the calibration object is no standard HALCON calibration plate `set_calib_data_observ_points` needs to be used (see section 3.2.3.2 on page 70).

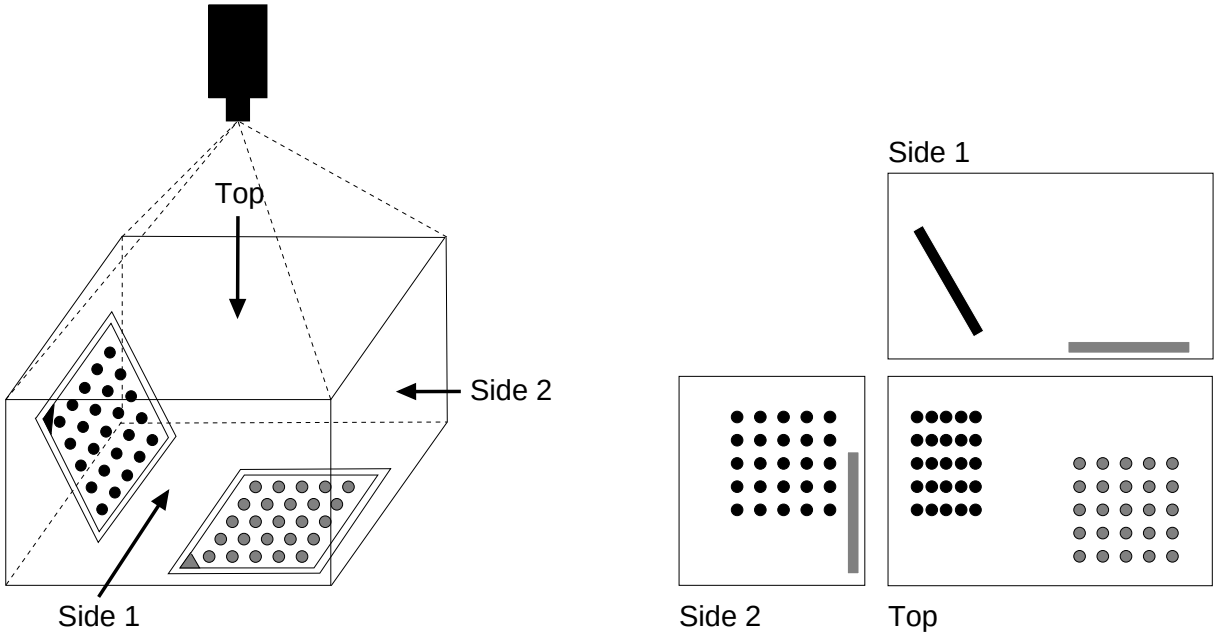


Figure 3.7: Investigation of the calibration volume: (left) calibration volume with two calibration plate poses and (right) the corresponding distribution of calibration marks when seen from different views. For a good calibration result, the areas without calibration marks (which are especially large in the view from side 1) have to be minimized by a cautious selection of the further calibration plate poses.

3.2.5 Restricting the Calibration to Specific Parameters

If certain camera parameters are already known, you can exclude them from the calibration with the operator `set_calib_data`. Analogously, you can restrict the calibration to certain parameters.

Please refer to the Reference Manual for more information and a short example.

3.2.6 Performing the Calibration

After preparing the calibration data model as described in the previous sections, you perform the calibration by calling the operator `calibrate_cameras`, using the calibration data model as input.

```
calibrate_cameras (CalibDataID, Errors)
```

As a direct result, only the calibration error is returned. It corresponds to the average distance (in pixels) between the backprojected calibration points and their extracted image coordinates. An error of up to 0.1 pixels indicates that the calibration was successful. You can further analyze the quality of the calibration results with the operator `get_calib_data` (see the Reference Manual for details).

The main results of the calibration, e.g., the internal camera parameters, are stored in the calibration data model. How to access them is described in the following section.

If the calibration fails, please refer to section 3.2.10 on page 76 for additional information.

3.2.7 Accessing the Results of the Calibration

The main results of the operator `calibrate_cameras` comprise the internal camera parameters and the pose of the calibration plate in each of the images from which the corresponding points were determined. The operator stores them in the calibration data model. You can access them with the operator `get_calib_data`.

The example program `%HALCONEXAMPLES%\solution_guide\3d_vision\camera_calibration_internal.hdev` shows how to access the internal camera parameters and write them into a file.

```
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
write_cam_par (CamParam, 'camera_parameters.dat')
```

The external camera parameters cannot be queried directly, because the needed information about the world coordinate system is not stored in the calibration data model. However, if the calibration plate was placed directly on the measurement plane, its pose can be used to easily derive the external camera parameters, which are the pose of the measurement plane. This is described in the following section.

3.2.7.1 Determining the External Camera Parameters

The external camera parameters describe the relation between the measurement plane and the camera, i.e., only if the external parameters are known it is possible to transform coordinates from the camera coordinate system (CCS) into the coordinate system of the measurement plane and vice versa. In HALCON, the measurement plane is defined as the plane $z = 0$ of the world coordinate system (WCS).

The external camera parameters can be determined in different ways:

1. Use the pose obtained from one of the calibration images in which the calibration plate is placed directly on the measurement plane. In this case, you just need to access this pose with the operator `get_calib_data`.
2. Separate the determination of the internal camera parameters from the determination of the external camera parameters by using an additional image in which the calibration plate is placed directly on the measurement plane. Apply `find_calib_object` to extract the calibration marks and the pose.
3. Determine the correspondences between 3D world points and their projections in the image by yourself and then call `vector_to_pose`.

If you only need to accurately measure the dimensions of an object, regardless of the absolute position of the object in a given coordinate system, one of the first two cases can be used.

The latter two cases have the advantage that the external camera parameters can be determined independently from the internal camera parameters. This is more flexible and might be useful if the measurements should be done in several planes from a single camera or if it is not possible to calibrate the camera in situ.

In the following, the different cases are described in more detail.

Placing the Calibration Plate on the Measurement Plane in One of the Calibration Images

The **first** case is the easiest way of determining the external parameters. The calibration plate must be placed directly on the measurement plane, e.g., the assembly line, in one of the (many) images used for the determination of the internal parameters.

Since the pose of the calibration plate is determined by the operator `calibrate_cameras`, you can simply access its pose with the operator `get_calib_data`. This way, internal and external parameters are determined in one single calibration step as is shown in the HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\camera_calibration_multi_image.hdev`. Here, the pose of the calibration plate (calibration object index 0) in the first calibration image is determined. Please note that each pose consists of seven values.

```
get_calib_data (CalibDataID, 'calib_obj_pose', [0, 1], 'pose', Pose)
```

The resulting pose would be the true pose of the measurement plane if the calibration plate were infinitely thin. Because real calibration plates have a thickness $d > 0$, the pose of the calibration plate is shifted by an amount $-d$ perpendicular to the measurement plane, i.e., along the z axis of the WCS. To correct this, we need to shift the pose by d along the z axis of the WCS. To perform this shift, the operator `set_origin_pose` can be used.

```
set_origin_pose (Pose, 0, 0, 0.002, Pose)
```

In general, the calibration plate can be oriented arbitrarily within the WCS as long as the spatial relation between the calibration plate and the measurement plane is known (see figure 3.8). Then, to derive the pose of the measurement plane from the pose of the calibration plate, a rigid transformation is necessary. In the following example, the pose of the calibration plate is adapted by a translation along the y axis followed by a rotation around the x axis.

```
pose_to_hom_mat3d (FinalPose, HomMat3D)
hom_mat3d_translate_local (HomMat3D, 0, 3.2, 0, HomMat3DTranslate)
hom_mat3d_rotate_local (HomMat3DTranslate, rad(-14), 'x', HomMat3DAdapted)
hom_mat3d_to_pose (HomMat3DAdapted, PoseAdapted)
```

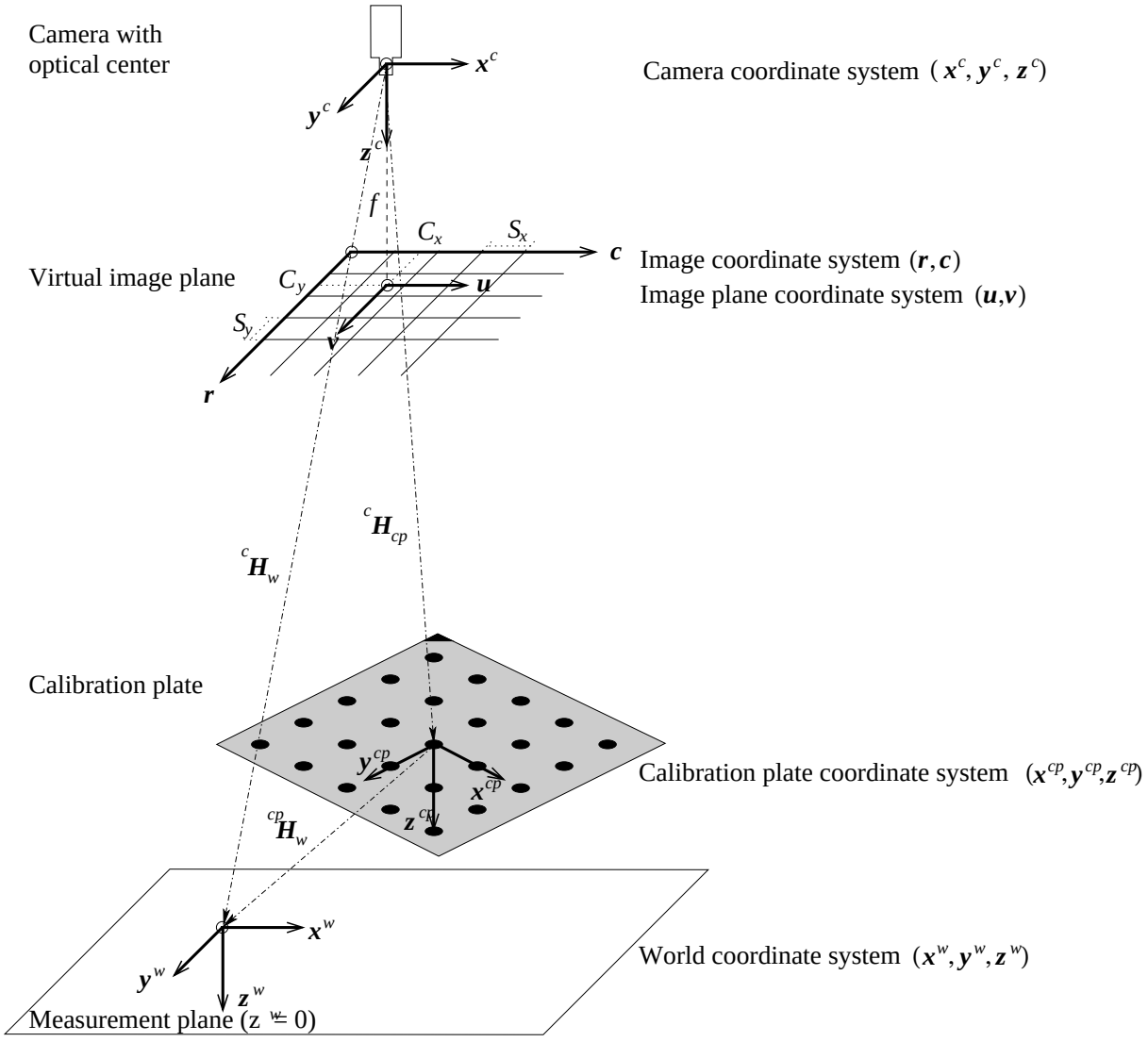


Figure 3.8: Relation between calibration plate and measurement plane.

Placing the Calibration Plate on the Measurement Plane in a Separate Image

If the advantages of using the HALCON calibration plate should be combined with the flexibility given by the separation of the internal and external camera parameters the **second** method for the determination of the external camera parameters can be used.

First, the camera is calibrated as described in the previous sections. This can be done, e.g., prior to the mounting of the camera at its final usage site.

Then, after setting up the camera at its final usage site, the external parameters can be determined. The only thing to be done is to take an additional image in which the calibration plate is placed directly on the measurement plane. From this image the external parameters can be determined as is shown in the HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\camera_calibration_external.hdev`. There, the internal camera parameters, the image in which the calibration plate was placed directly on the measurement plane, and the world coordinates of the calibration marks are read from file.

```
read_cam_par ('camera_parameters.dat', CamParam)
read_image (Image, ImgPath + 'calib_11')
```

Then, the calibration marks and the pose of the calibration plate are extracted.

```
find_calib_object (Image, CalibDataID, 0, 0, 1, [], [])
```

Finally, to take the thickness of the calibration plate into account, the z value of the origin given by the camera pose is translated by the thickness of the calibration plate.

```
set_origin_pose (PoseForCalibrationPlate, 0, 0, 0.00075, \
                 PoseForCalibrationPlate)
```

Note that it is very important to fix the focus of your camera if you want to separate the calibration process into two steps as described in this section, because changing the focus is equivalent to changing the focal length, which is part of the internal parameters.

Using Known 3D Points and Their Corresponding Image Points

If it is necessary to perform the measurements within a given world coordinate system, the **third** case for the determination of the external camera parameters can be used. Here, you need to know the 3D world coordinates of at least three points that do not lie on a straight line. Then, you must determine the corresponding image coordinates of the projections of these points. Now, the operator `vector_to_pose` can be used for the determination of the external camera parameters.

An example for this possibility of determining the external parameters is given in the following program. First, the world coordinates of three points are set.

```
X := [0, 50, 100, 80]
Y := [5, 0, 5, 0]
Z := [0, 0, 0, 0]
```

Then, the image coordinates of the projections of these points in the image are determined. In this example, they are simply set to some approximate values. In reality, they should be determined with subpixel accuracy since they define the external camera parameters.

```
RCoord := [414, 227, 85, 128]
CCoord := [119, 318, 550, 448]
```

Finally, the operator `vector_to_pose` is called with the correspondences and the internal camera parameters.

```
vector_to_pose (X, Y, Z, RCoord, CCoord, CamParam, 'iterative', 'error', \
                FinalPose, Errors)
```

Again, it is very important to fix the focus of your camera because changing the focus is equivalent to changing the focal length, which is part of the internal parameters.

3.2.8 Deleting Observations from the Calibration Data Model

To determine the effect of an observation on the calibration or to remove observations of bad quality from the calibration data model, an observation can be deleted using the operator `remove_calib_data_observ`. For acquiring calibration images of suitable quality please refer to the recommendations listed in the section “How to take a set of suitable images?” in the chapter reference “Calibration”.

```
RemoveObservationIdx := 3
remove_calib_data_observ (CalibDataID, 0, 0, RemoveObservationIdx)
```

When performing the camera calibration again as described in section 3.2.6 on page 72, it can be noted that the calibration error changes.

3.2.9 Saving the Results

After accessing the results you can store them with `write_cam_par` and `write_pose`).

3.2.10 Troubleshooting

Below, you find information for the case that the calibration of a line scan camera fails.

3.2.10.1 Problems With Calibrating Line Scan Cameras

In general, the procedure for the calibration of line scan cameras is identical to the one for the calibration of area scan cameras.

However, line scan imaging suffers from a high degree of parameter correlation. For example, for pinhole line scan cameras, any small rotation of the linear array around the x-axis of the camera coordinate system can be compensated by changing the y-component of the translation vector of the respective pose. Even the focal length is correlated with the z-component of the translation vector of the pose, i.e., with the distance of the object from the camera.

The consequences of these correlations for the calibration of line scan cameras are that some parameters cannot be determined with high absolute accuracy. Nevertheless, the set of parameters is determined consistently, what means that the world coordinates can be measured with high accuracy.

Another consequence of the parameter correlations is that the calibration may fail in some cases where the start values for the internal camera parameters are not accurate enough. If this happens, try the following approach for pinhole line scan cameras: In many cases, the start values for the motion vector are the most difficult to set. To achieve better start values for the parameters Vx, Vy, and Vz, reduce the number of parameters to be estimated such that the camera calibration succeeds. Try first to estimate the parameters Vx, Vy, Vz, α, β, γ, tx, ty, and tz by calling `set_calib_data` with `ItemType` = `'camera'`, `ItemIdx` = `'general'`, `DataName` = `'calib_settings'`, and `DataValue` = *['vx', 'vy', 'vz', 'alpha', 'beta', 'gamma', 'transx', 'transy', 'transz']* and if this does not work, try `DataValue` = *['vx', 'vy', 'vz', 'transx', 'transy', 'transz']*. Then, determine the whole set of parameters using the above determined values for Vx, Vy, and Vz as start values. A similar approach may be used for telecentric line scan cameras. Of course, since tz and Vz have no effect for telecentric line scan cameras, these parameters do not need to be excluded explicitly.

If none of the above proposed tips works, try to determine better start values directly from the camera setup. If possible, change the setup such that it is easier to determine appropriate start values, e.g., mount the camera such that it looks approximately perpendicularly onto the conveyor belt (see figure 3.3 on page 68).

If the calibration plates lie in a plane, you may get the error 8440 ("`Camera calibration did not converge`"), because in this case the parameters are even more correlated. A possible solution may be to exclude the rotation around the x axis, i.e., `'alpha'` from the calibration with the operator `set_calib_data`.

3.3 Transforming Image into World Coordinates and Vice Versa

In this section, you learn how to obtain world coordinates from images based on the calibration data. On the one hand, it is possible to process the images as usual and then to transform the extraction results into the world coordinate system. In many cases, this will be the most efficient way of obtaining world coordinates. On the other hand, some applications may require that the segmentation itself must be carried out in images that are already transformed into the world coordinate system (see section 3.4 on page 80).

In general, the segmentation process reduces the amount of data that needs to be processed. Therefore, rectifying the segmentation results is faster than rectifying the underlying image. What is more, it is often better to perform the segmentation process directly on the original images because smoothing or aliasing effects may occur in the rectified image, which could disturb the segmentation and may lead to inaccurate results. These arguments suggest to rectify the segmentation results instead of the images.

In the following, first some general remarks on the underlying principle of the transformation of image coordinates into world coordinates are given. Then, it is described how to transform points, contours, and regions into the world coordinate system. Finally, we show that it is possible to transform world coordinates into image coordinates as well, e.g., in order to visualize information given in the world coordinate system.

3.3.1 The Main Principle

Given the image coordinates of one point, the goal is to determine the world coordinates of the corresponding point in the measurement plane. For this, the line of sight, i.e., a straight line from the optical center of the camera through the given point in the image plane, must be intersected with the measurement plane (see figure 3.9).

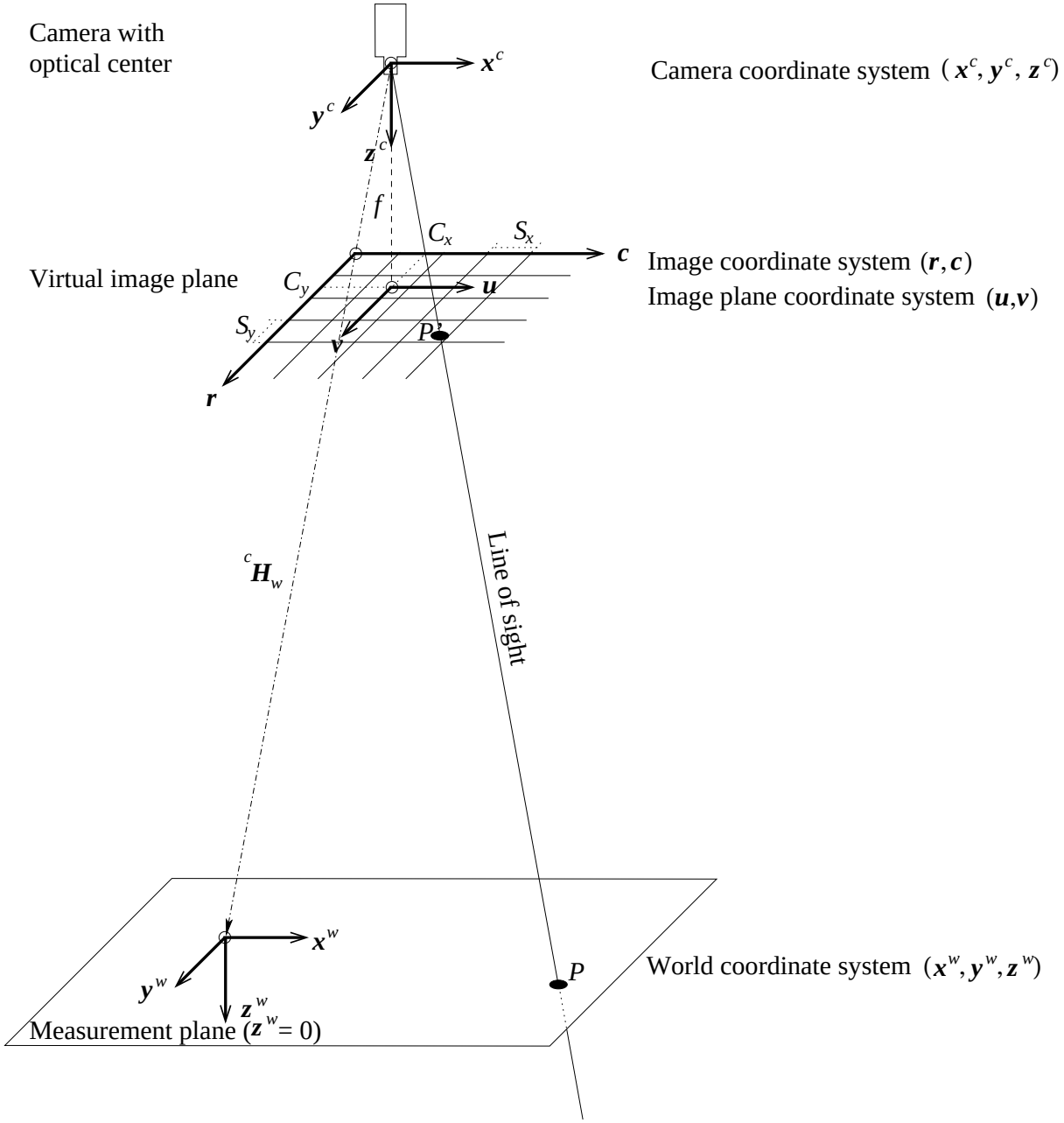


Figure 3.9: Intersecting the line of sight with the measurement plane.

The calibration data is necessary to transform the image coordinates into camera coordinates and finally into world coordinates.

All these calculations are performed by the operators of the family `..._to_world_plane`.

Again, please remember that in HALCON the measurement plane is defined as the plane $z = 0$ with respect to the world coordinate system. This means that all points returned by the operators of the family `..._to_world_plane` have a z-coordinated equal to zero, i.e., they lie in the plane $z = 0$ of the world coordinate system.

3.3.2 World Coordinates for Points

The world coordinates of an image point (r, c) can be determined using the operator image_points_to_world_plane. In the following code example, the row and column coordinates of pitch lines are transformed into world coordinates.

```
image_points_to_world_plane (CamParam, FinalPose, RowPitchLine, \
                             ColPitchLine, 1, X1, Y1)
```

As input, the operator requires the internal and external camera parameters as well as the row and column coordinates of the point(s) to be transformed.

Additionally, the unit in which the resulting world coordinates are to be given is specified by the parameter Scale (see also the description of the operator image_to_world_plane in section 3.4.1 on page 80). This parameter is the ratio between the unit in which the resulting world coordinates are to be given and the unit in which the world coordinates of the calibration target are given (equation 3.1).

$$\texttt{Scale} = \frac{\text{unit of resulting world coordinates}}{\text{unit of world coordinates of calibration target}} \tag{3.1}$$

In many cases the coordinates of the calibration target are given in meters. In this case, it is possible to set the unit of the resulting coordinates directly by setting the parameter Scale to *'m'* (corresponding to the value *1.0*, which could be set alternatively for the parameter Scale), *'cm'* (*0.01*), *'mm' (0.001)*, *'microns'* (*1e-6*), or *'µm'* (again, *1e-6*). Then, if the parameter Scale is set to, e.g., *'m'*, the resulting coordinates are given in meters. If, e.g., the coordinates of the calibration target are given in μm and the resulting coordinates have to be given in millimeters, the parameter Scale must be set to:

$$\texttt{Scale} = \frac{mm}{\mu m} = \frac{1 \cdot 10^{-3}\, m}{1 \cdot 10^{-6}\, m} = 1000 \tag{3.2}$$

3.3.3 World Coordinates for Contours

If you want to convert an XLD object containing pixel coordinates into world coordinates, the operator contour_to_world_plane_xld can be used. Its parameters are similar to those of the operator image_points_to_world_plane, as can be seen from the following example program.

```
lines_gauss (ImageReduced, Lines, 1, 3, 8, 'dark', 'true', 'bar-shaped', \
             'true')
contour_to_world_plane_xld (Lines, ContoursTrans, CamParam, PoseAdapted, 1)
```

3.3.4 World Coordinates for Regions

In HALCON, regions cannot be transformed directly into the world coordinate system. Instead, you must first convert them into XLD contours using the operator gen_contour_region_xld, then apply the transformation to these XLD contours as described in the previous section.

If the regions have holes and if these holes would influence your further calculations, set the parameter Mode of the operator gen_contour_region_xld to *'border_holes'*. Then, in addition to the outer border of the input region the operator gen_contour_region_xld returns the contours of all holes.

3.3.5 Transforming World Coordinates into Image Coordinates

In this section, the transformation between image coordinates and world coordinates is performed in the opposite direction, i.e., from world coordinates to image coordinates. This is useful if you want to visualize information given in world coordinates or it may be helpful for the definition of meaningful regions of interest (ROI).

First, the world coordinates must be transformed into the camera coordinate system. For this, the homogeneous transformation matrix ${}^{CCS}\mathbf{H}_{WCS}$ is needed, which can easily be derived from the pose of the measurement plane

with respect to the camera by the operator pose_to_hom_mat3d. The transformation itself can be carried out using the operator affine_trans_point_3d. Then, the 3D coordinates, now given in the camera coordinate system, can be projected into the image plane with the operator project_3d_point. An example program is given in the following:

There, the world coordinates of four points defining a rectangle in the WCS are defined.

```
ROI_X_WCS := [-2, -2, 112, 112]
ROI_Y_WCS := [0, 0.5, 0.5, 0]
ROI_Z_WCS := [0, 0, 0, 0]
```

Then, the transformation matrix ${}^{CCS}\mathbf{H}_{WCS}$ is derived from the respective pose.

```
pose_to_hom_mat3d (FinalPose, CCS_HomMat_WCS)
```

With this transformation matrix, the world points are transformed into the camera coordinate system.

```
affine_trans_point_3d (CCS_HomMat_WCS, ROI_X_WCS, ROI_Y_WCS, ROI_Z_WCS, \
                       CCS_RectangleX, CCS_RectangleY, CCS_RectangleZ)
```

Finally, the points are projected into the image coordinate system.

```
project_3d_point (CCS_RectangleX, CCS_RectangleY, CCS_RectangleZ, CamParam, \
                  RectangleRow, RectangleCol)
```

3.3.6 Compensate for Lens Distortions Only

All operators discussed above automatically compensate for lens distortions. In some cases, you might want to compensate for lens distortions only without transforming results or images into world coordinates.

Note that in the following, only the compensation for radial distortions using the division model is described. The compensation for radial and decentering distortions using the polynomial model is done analogously by replacing $\kappa = 0$ with $K_1 = K_2 = K_3 = P_1 = P_2 = 0$.

The procedure is to specify the original internal camera parameters and those of a virtual camera that does not produce lens distortions, i.e., with $\kappa = 0$.

The easiest way to obtain the internal camera parameters of the virtual camera would be to simply set κ to zero. This can be done directly by changing the respective value of the internal camera parameters.

```
CamParVirtualFixed := CamParOriginal
set_cam_par_data (CamParVirtualFixed, 'kappa', 0, CamParVirtualFixed)
```

Alternatively, the operator change_radial_distortion_cam_par can be used with the parameter Mode set to *'fixed'* and the parameter DistortionCoeffs set to *0*.

```
change_radial_distortion_cam_par ('fixed', CamParOriginal, 0, \
                                  CamParVirtualFixed)
```

Then, for the rectification of the segmentation results, the HALCON operator change_radial_distortion_contours_xld can be used, which requires as input parameters the original and the virtual internal camera parameters. If you want to change the lens distortion of image coordinates (Row, Col), you can alternatively use change_radial_distortion_points.

```
change_radial_distortion_contours_xld (Edges, EdgesRectifiedFixed, \
                                       CamParOriginal, CamParVirtualFixed)
```

The rectification of the segmentation results changes the visible part of the scene (see figure 3.10b). To obtain virtual camera parameters such that the whole image content lies within the visible part of the scene, the parameter

Mode of the operator change_radial_distortion_cam_par must be set to *'fullsize'* (see figure 3.10c). Again, to eliminate the lens distortions, the parameter DistortionCoeffs must be set to *0*, or all coefficients of the polynomial model must be set to zero, respectively.

```
change_radial_distortion_cam_par ('fullsize', CamParOriginal, 0, \
                                  CamParVirtualFullsize)
```

If the lens distortions are eliminated in the image itself using the rectification procedure described in section 3.4.2 on page 86, the mode *'fullsize'* may lead to undefined pixels in the rectified image. The mode *'adaptive'* (see figure 3.10d) slightly reduces the visible part of the scene to prevent such undefined pixels.

```
change_radial_distortion_cam_par ('adaptive', CamParOriginal, 0, \
                                  CamParVirtualAdaptive)
```

The mode *'preserve_resolution'* (see figure 3.10e) works similar to the mode *'fullsize'* but prevents undefined pixels by additionally increasing the size of the modified image so that the image resolution does not decrease in any part of the image.

```
change_radial_distortion_cam_par ('preserve_resolution', CamParOriginal, 0, \
                                  CamParVirtualPreservedResolution)
```

Note that this compensation for lens distortions is not possible for pinhole line scan images because of the acquisition geometry of pinhole line scan cameras. To eliminate radial distortions from segmentation results of pinhole line scan images, the segmentation results must be transformed into the WCS (see section 3.3.2 on page 78, section 3.3.3 on page 78, and section 3.3.4 on page 78).

For telecentric line scan cameras, the lens distortion compensation works in an identical manner as for area scan cameras. In addition, for telecentric line scan cameras, the motion vector also influences the perceived distortion. For example, a nonzero V_x component leads to skewed pixels. Furthermore, if $V_y \neq S_x/m$, where m is the magnification of the lens, the pixels appear to be non-square. Therefore, for telecentric line scan cameras, up to three more components can be passed in addition to κ in DistortionCoeffs. These specify the new V_x, V_y, and V_z components of the motion vector. In particular, setting V_x to 0 and V_y to S_x/m results in square pixels in the output image if Mode is set to *'fixed'*.

3.4 Rectifying Images

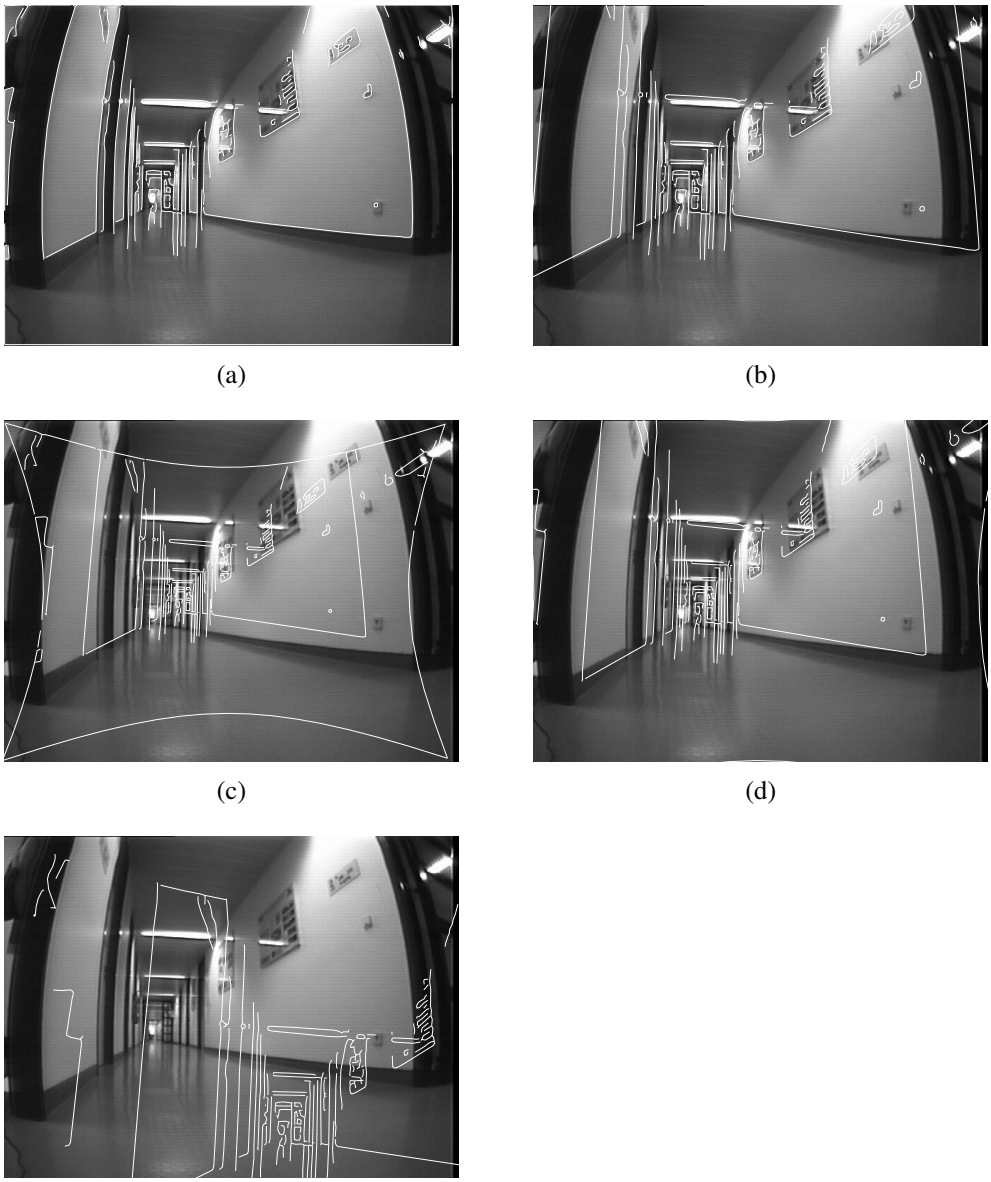
For applications like blob analysis or OCR, it may be necessary to have undistorted images. Imagine that an OCR has been trained based on undistorted image data. Then, it will not be able to recognize characters in heavily distorted images. In such a case, the image data must be rectified, i.e., the lens and perspective distortions must be eliminated before the OCR can be applied.

3.4.1 Transforming Images into the WCS

The operator image_to_world_plane rectifies an image by transforming it into the measurement plane, i.e., the plane $z = 0$ of the WCS. The rectified image shows no lens and no perspective distortions. It corresponds to an image captured by a camera that produces no lens distortions and that looks perpendicularly to the measurement plane.

```
image_to_world_plane (Image, ImageMapped, CamParam, PoseForCenteredImage, \
                      WidthMappedImage, HeightMappedImage, \
                      ScaleForCenteredImage, 'bilinear')
```

If more than one image must be rectified, a projection map can be determined with the operator gen_image_to_world_plane_map, which is used analogously to the operator image_to_world_plane, followed by the actual transformation of the images, which is carried out by the operator map_image.



(a) (b) (c) (d) (e)

Figure 3.10: Eliminating radial distortions: The original image overlaid with (a) edges extracted from the original image; (b) edges rectified by setting κ to zero; (c) edges rectified with mode *'fullsize'*; (d) edges rectified with mode *'adaptive'*; (e) edges rectified with mode *'preserved_resolution'*.

```
gen_image_to_world_plane_map (Map, CamParam, PoseForCenteredImage, \
                              WidthOriginalImage, HeightOriginalImage, \
                              WidthMappedImage, HeightMappedImage, \
                              ScaleForCenteredImage, 'bilinear')
map_image (Image, Map, ImageMapped)
```

The size of the rectified image can be chosen with the parameters `Width` and `Height` for the operator `image_to_world_plane` and with the parameters `WidthMapped` and `HeightMapped` for the operator `gen_image_to_world_plane_map`. The size of the rectified image must be given in pixels.

The pixel size of the rectified image is specified by the parameter `Scale` (see also the description of the operator `image_points_to_world_plane` in section 3.3.2 on page 78). This parameter is the ratio between the pixel size of the rectified image and the unit in which the world coordinates of the calibration target are given (equation 3.3).

$$\texttt{Scale} = \frac{\text{pixel size of rectified image}}{\text{unit of world coordinates of calibration target}} \tag{3.3}$$

In many cases the coordinates of the calibration targets are given in meters. In this case, it is possible to set the pixel size directly by setting the parameter `Scale` to *'m'* (corresponding to the value *1.0*, which could be set alternatively for the parameter `Scale`), *'cm'* (*0.01*), *'mm'* (*0.001*), *'microns'* (*1e-6*), or *'μm'* (again, *1e-6*). Then, if the parameter `Scale` is set to, e.g., 'μm', one pixel of the rectified image has a size that corresponds to an area of 1 μm $\times$ 1 μm in the world. The parameter `Scale` should be chosen such that in the center of the area of interest the pixel size of the input image and of the rectified image is similar. Large scale differences would lead to aliasing or smoothing effects. See below for examples of how the scale can be determined.

The parameter `Interpolation` specifies whether bilinear interpolation (*'bilinear'*) should be applied between the pixels in the input image or whether the gray value of the nearest neighboring pixel (*'none'*) should be used.

The rectified image `ImageWorld` is positioned such that its upper left corner is located exactly at the origin of the WCS and that its column axis is parallel to the x-axis of the WCS. Since the WCS is defined by the external camera parameters `CamPose` the position of the rectified image `ImageWorld` can be translated by applying the operator `set_origin_pose` to the external camera parameters. Arbitrary transformations can be applied to the external camera parameters based on homogeneous transformation matrices. See below for examples of how the external camera parameters can be set.

In figure 3.11, the WCS has been defined such that the upper left corner of the rectified image corresponds to the upper left corner of the input image. To illustrate this, in figure 3.11, the full domain of the rectified image, transformed into the virtual image plane of the input image, is displayed. As can be seen, the upper left corner of the input image and of the projection of the rectified image are identical.

Note that it is also possible to define the WCS such that the rectified image does not lie or lies only partly within the imaged area. The domain of the rectified image is set such that it contains only those pixels that lie within the imaged area, i.e., for which gray value information is available. In figure 3.12, the WCS has been defined such that the upper part of the rectified image lies outside the imaged area. To illustrate this, the part of the rectified image for which no gray value information is available is displayed dark gray. Also in figure 3.12, the full domain of the rectified image, transformed into the virtual image plane of the input image, is displayed. It can be seen that for the upper part of the rectified image no image information is available.

If several images must be rectified using the same camera parameters the operator `gen_image_to_world_plane_map` in combination with `map_image` is much more efficient than the operator `image_to_world_plane` because the transformation must be determined only once. In this case, a projection map that describes the transformation between the image plane and the world plane is generated first by the operator `gen_image_to_world_plane_map`. Then, this map is used by the operator `map_image` to transform the image very efficiently.

The following example from `%HALCONEXAMPLES%\solution_guide\3d_vision\transform_image_into_wcs.hdev` shows how to perform the transformation of images into the world coordinate system using the operators `gen_image_to_world_plane_map` together with `map_image` as well as the operator `image_to_world_plane`.

In the first part of the example program the parameters `Scale` and `CamPose` are set such that a given point appears in the center of the rectified image and that in the surroundings of this point the scale of the rectified image is similar to the scale of the original image.

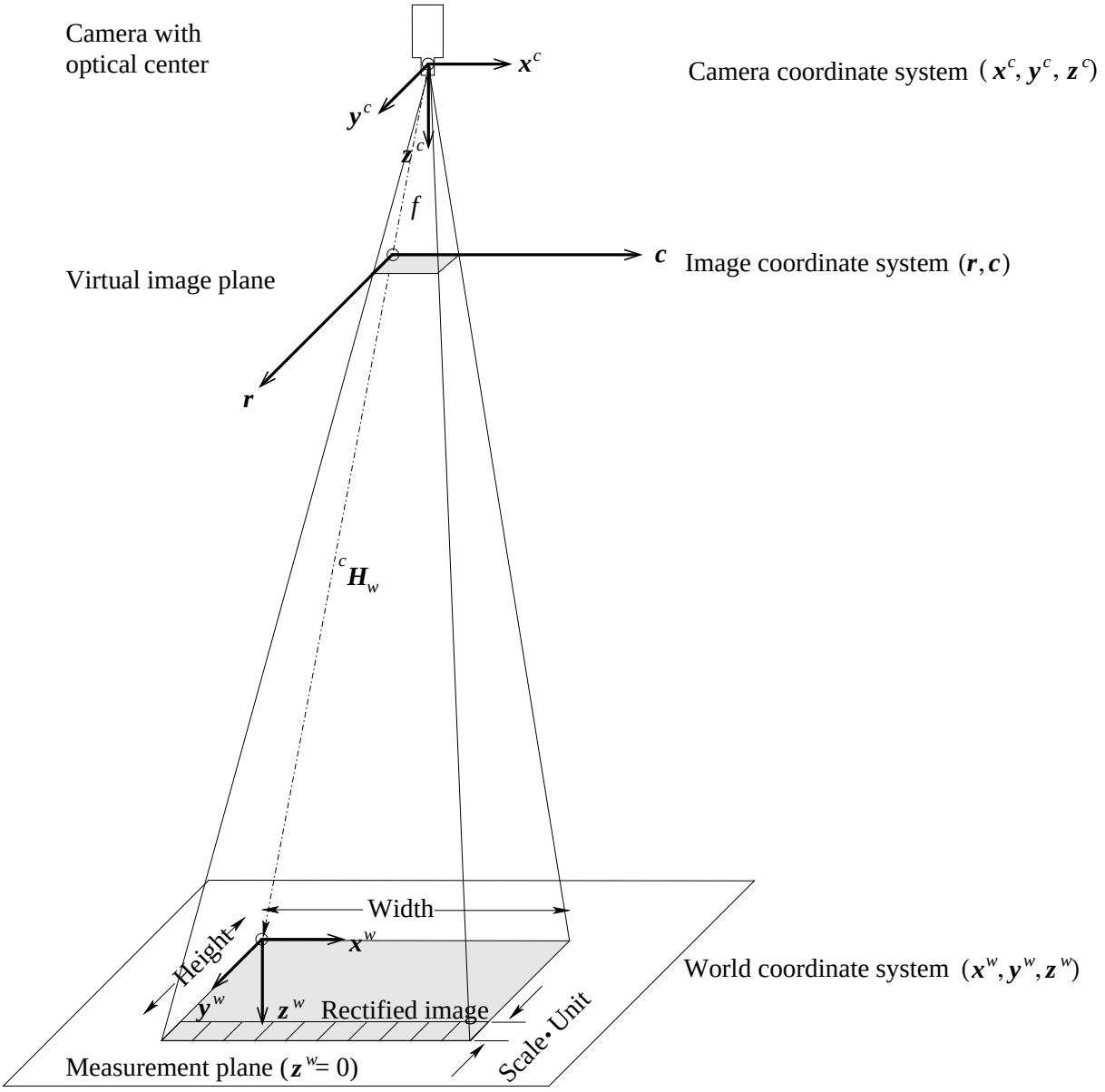


Figure 3.11: Projection of the image into the measurement plane.

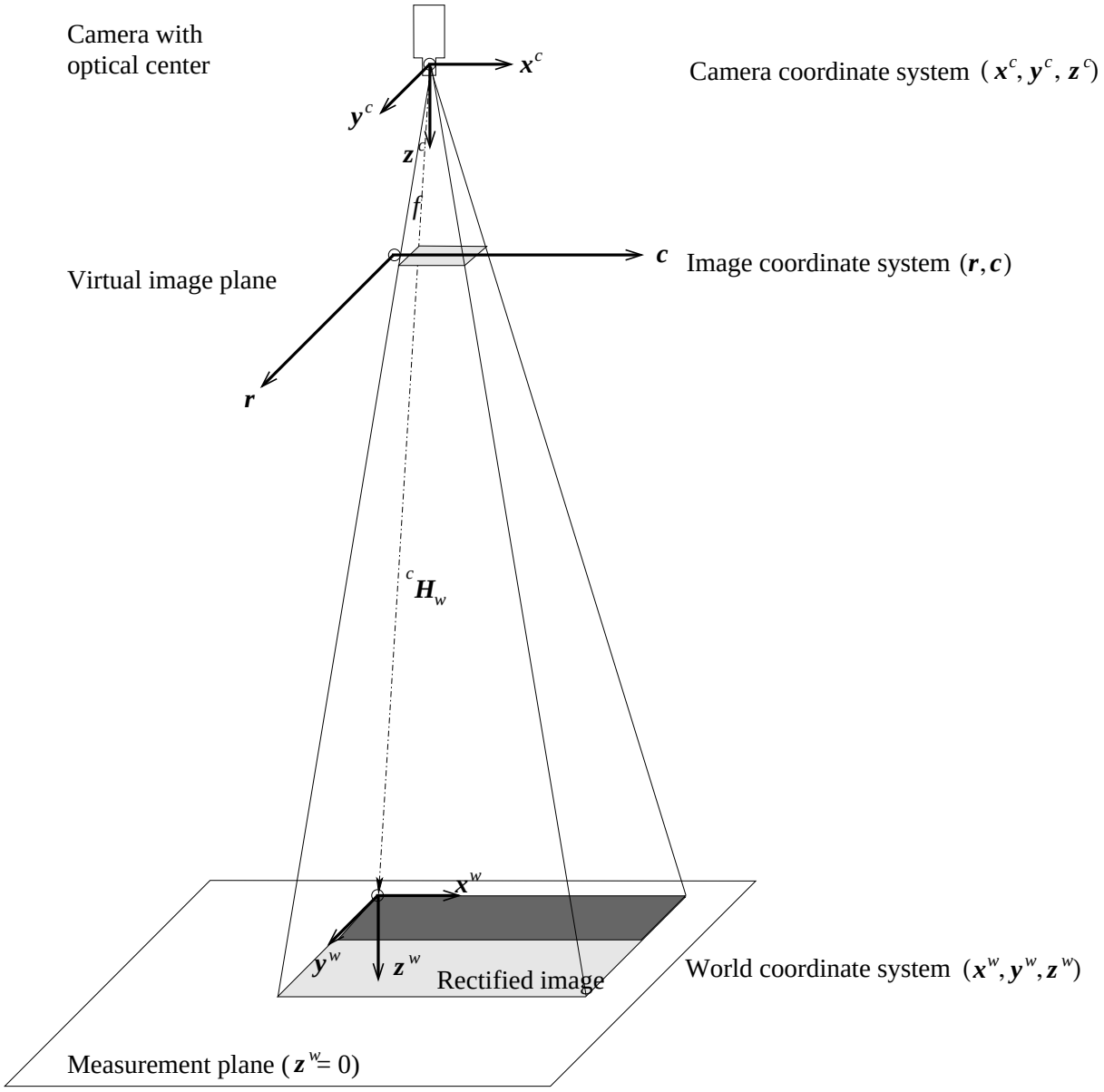


Figure 3.12: Projection of the image into the measurement plane with part of the rectified image lying outside the image area.

First, the size of the rectified image is defined.

```
WidthMappedImage := 652
HeightMappedImage := 494
```

Then, the scale is determined based on the ratio of the distance between points in the WCS and of the respective distance in the ICS.

```
Dist_ICS := 1
image_points_to_world_plane (CamParam, Pose, CenterRow, CenterCol, 1, \
                             CenterX, CenterY)
image_points_to_world_plane (CamParam, Pose, CenterRow + Dist_ICS, \
                             CenterCol, 1, BelowCenterX, BelowCenterY)
image_points_to_world_plane (CamParam, Pose, CenterRow, \
                             CenterCol + Dist_ICS, 1, RightOfCenterX, \
                             RightOfCenterY)
distance_pp (CenterY, CenterX, BelowCenterY, BelowCenterX, \
             Dist_WCS_Vertical)
distance_pp (CenterY, CenterX, RightOfCenterY, RightOfCenterX, \
             Dist_WCS_Horizontal)
ScaleVertical := Dist_WCS_Vertical / Dist_ICS
ScaleHorizontal := Dist_WCS_Horizontal / Dist_ICS
ScaleForCenteredImage := (ScaleVertical + ScaleHorizontal) / 2.0
```

Now, the pose of the measurement plane is modified such that a given point will be displayed in the center of the rectified image.

```
DX := CenterX - ScaleForCenteredImage * WidthMappedImage / 2.0
DY := CenterY - ScaleForCenteredImage * HeightMappedImage / 2.0
DZ := 0
set_origin_pose (Pose, DX, DY, DZ, PoseForCenteredImage)
```

These calculations are implemented in the HDevelop procedure `parameters_image_to_world_plane_centered`.

```
procedure parameters_image_to_world_plane_centered (: : CamParam, Pose,
                                                    CenterRow, CenterCol,
                                                    WidthMappedImage,
                                                    HeightMappedImage:
                                                    ScaleForCenteredImage,
                                                    PoseForCenteredImage)
```

which is part of the HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\transform_image_into_wcs.hdev` (see appendix A.2 on page 232).

Finally, the image can be transformed.

```
gen_image_to_world_plane_map (Map, CamParam, PoseForCenteredImage, \
                              WidthOriginalImage, HeightOriginalImage, \
                              WidthMappedImage, HeightMappedImage, \
                              ScaleForCenteredImage, 'bilinear')
map_image (Image, Map, ImageMapped)
```

The second part of the example program `%HALCONEXAMPLES%\solution_guide\3d_vision\transform_image_into_wcs.hdev` shows how to set the parameters `Scale` and `CamPose` such that the entire image is visible in the rectified image.

First, the image coordinates of the border of the original image are transformed into world coordinates.

```
full_domain (Image, ImageFull)
get_domain (ImageFull, Domain)
gen_contour_region_xld (Domain, ImageBorder, 'border')
contour_to_world_plane_xld (ImageBorder, ImageBorderWCS, CamParam, Pose, 1)
```

Then, the extent of the image in world coordinates is determined.

```
smallest_rectangle1_xld (ImageBorderWCS, MinY, MinX, MaxY, MaxX)
ExtentX := MaxX - MinX
ExtentY := MaxY - MinY
```

The scale is the ratio of the extent of the image in world coordinates and of the size of the rectified image.

```
ScaleX := ExtentX / WidthMappedImage
ScaleY := ExtentY / HeightMappedImage
```

Now, the maximum value must be selected as the final scale.

```
ScaleForEntireImage := max([ScaleX,ScaleY])
```

Finally, the origin of the pose must be translated appropriately.

```
set_origin_pose (Pose, MinX, MinY, 0, PoseForEntireImage)
```

These calculations are implemented in the HDevelop procedure

```
procedure parameters_image_to_world_plane_entire (Image: : CamParam, Pose,
                                                  WidthMappedImage,
                                                  HeightMappedImage:
                                                  ScaleForEntireImage,
                                                  PoseForEntireImage)
```

which is part of the example program `%HALCONEXAMPLES%\solution_guide\3d_vision\transform_image_into_wcs.hdev` (see appendix A.3 on page 232).

If the object is not planar the projection map that is needed by the operator `map_image` may be determined by the operator `gen_grid_rectification_map`, which is described in section 11.3 on page 223.

If only the lens distortions should be eliminated the projection map can be determined by the operator `gen_radial_distortion_map`, which is described in the following section.

3.4.2 Compensate for Lens Distortions Only

The principle of the compensation for lens distortions has already be described in section 3.3.6 on page 79.

If only one image must be rectified the operator `change_radial_distortion_image` can be used. It is used analogously to the operator `change_radial_distortion_contours_xld` described in section 3.3.6, with the only exception that a region of interest (ROI) can be defined with the parameter `Region`.

```
change_radial_distortion_image (GrayImage, ROI, ImageRectifiedAdaptive, \
                                CamParOriginal, CamParVirtualAdaptive)
```

Again, the internal parameters of the virtual camera that does not show lens distortions can be determined by setting κ to zero for the division model or K_1, K_2, K_3, P_1, and P_2 to zero for the polynomial model (see figure 3.13b). Alternatively, the internal parameters of the virtual camera can be obtained by using the operator `change_radial_distortion_cam_par` with the parameter `Mode` set to *'fixed'* (equivalent to setting κ or the coefficients of the polynomial model to zero; see figure 3.13b), *'adaptive'* (see figure 3.13c), *'fullsize'* (see figure 3.13d), or *'preserve_resolution'* (see figure 3.13e).

If more than one image must be rectified, a projection map can be determined with the operator `gen_radial_distortion_map`, which is used analogously to the operator `change_radial_distortion_image`, followed by the actual transformation of the images, which is carried out by the operator `map_image`, described in section 3.4.1 on page 80. If a ROI is to be specified, it must be rectified separately (see section 3.3.4 on page 78).

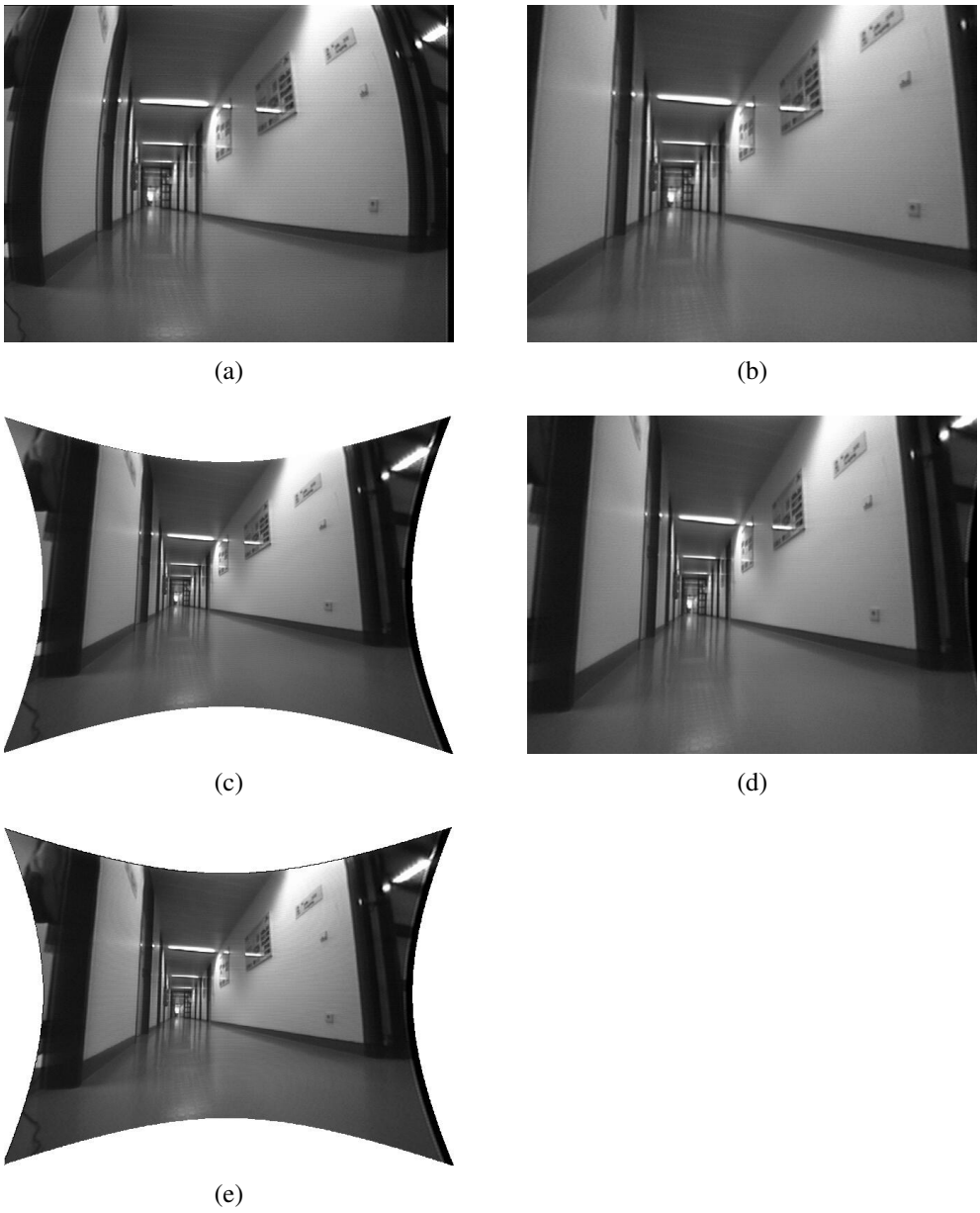


Figure 3.13: Eliminating radial distortions: (a) The original image; (b) the image rectified by setting κ to zero; (c) the image rectified with mode *'fullsize'*; (d) the image rectified with mode *'adaptive'*; (e) the image rectified with mode *'preserved_resolution'*.

```
gen_radial_distortion_map (MapFixed, CamParOriginal, CamParVirtualFixed, \
                           'bilinear')
map_image (GrayImage, MapFixed, ImageRectifiedFixed)
```

Note that this compensation for lens distortions is not possible for pinhole line scan images because of the acquisition geometry of pinhole line scan cameras. To eliminate radial distortions from pinhole line scan images, the images must be transformed into the WCS (see section 3.4.1 on page 80).

For telecentric line scan cameras, the compensation of lens distortions works in and identical manner as for area scan cameras. See section 3.3.6 on page 79 for further details.

3.5 Inspection of Non-Planar Objects

Note that the measurements described so far will only be accurate if the object to be measured is planar, i.e., if it has a flat surface. If this is not the case the perspective projection of the pinhole camera (see equation 2.24 on

page 26) will make the parts of the object that lie closer to the camera appear bigger than the parts that lie farther away. In addition, the respective world coordinates are displaced systematically. If you want to measure the top side of objects with a flat surface that have a significant thickness that is equal for all objects it is best to place the calibration plate onto one of these objects during calibration. With this, you can make sure that the optical rays are intersected with the correct plane.

The displacement that results from deviations of the object surface from the measurement plane can be estimated very easily. Figure 3.14 shows a vertical section of a typical measurement configuration. The measurement plane is drawn as a thick line, the object surface as a dotted line. Note that the object surface does not correspond to the measurement plane in this case. The deviation of the object surface from the measurement plane is indicated by Δz, the distance of the projection center from the measurement plane by z, and the displacement by Δr. The point N indicates the perpendicular projection of the projection center (PC) onto the measurement plane.

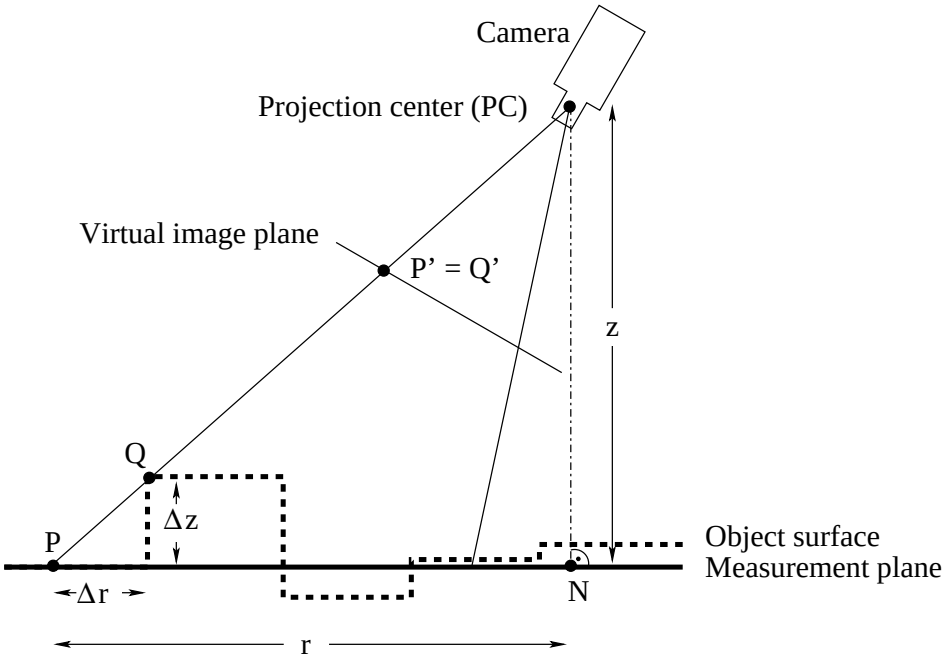


Figure 3.14: Displacement Δr caused by a deviation of the object surface from the measurement plane.

For the determination of the world coordinates of point Q, which lies on the object surface, the optical ray from the projection center of the camera through Q', which is the projection of Q into the image plane, is intersected with the measurement plane. For this reason, the operators of the family `..._to_world_plane` do not return the world coordinates of Q, but the world coordinates of point P, which is the perspective projection of point Q' onto the measurement plane.

If we know the distance r from P to N, the distance z, which is the shortest distance from the projection center to the measurement plane, and the deviation Δz of the object's surface from the measurement plane, the displacement Δr can be calculated by:

$$\Delta r = \Delta z \cdot \frac{r}{z} \tag{3.4}$$

Often, it will be sufficient to have just a rough estimate for the value of Δr. Then, the values r, z, and Δz can be approximately determined directly from the measurement setup.

If you need to determine Δr more precisely, you first have to calibrate the camera. Then you have to select a point Q' in the image for which you want to know the displacement Δr. The transformation of Q' into the WCS using the operator `image_points_to_world_plane` yields the world coordinates of point P. Now, you need to derive the world coordinates of the point N. An easy way to do this is to transform the camera coordinates of the projection center PC, which are $(0, 0, 0)^T$, into the world coordinate system, using the operator `affine_trans_point_3d`. To derive the homogeneous transformation matrix ${}^{WCS}\mathbf{H}_{CCS}$ needed for the above mentioned transformation, first, generate the homogeneous transformation matrix ${}^{CCS}\mathbf{H}_{WCS}$ from the pose of the measurement plane via the operator `pose_to_hom_mat3d` and then, invert the resulting homogeneous transformation matrix (`hom_mat3d_invert`). Because N is the perpendicular projection of PC onto the measurement plane, its x and y world coordinates are equal to the respective world coordinates of PC and its z coordinate is

equal to zero. Now, r and z can be derived as follows: r is the distance from P to N, which can be calculated by the operator distance_pp; z is simply the z coordinate of PC, given in the WCS.

The following HALCON program (%HALCONEXAMPLES%\solution_guide\3d_vision\height_displacement.hdev) shows how to implement this approach. First, the camera parameters are read from file.

```
read_cam_par ('camera_parameters.dat', CamParam)
read_pose ('pose_from_three_points.dat', Pose)
```

Then, the deviation of the object surface from the measurement plane is set.

```
DeltaZ := 2
```

Finally, the displacement is calculated, according to the method described above.

```
get_mbutton (WindowHandle, RowQ, ColumnQ, Button)
image_points_to_world_plane (CamParam, Pose, RowQ, ColumnQ, 1, WCS_PX, \
                             WCS_PY)
pose_to_hom_mat3d (Pose, CCS_HomMat_WCS)
hom_mat3d_invert (CCS_HomMat_WCS, WCS_HomMat_CCS)
affine_trans_point_3d (WCS_HomMat_CCS, 0, 0, 0, WCS_PCX, WCS_PCY, WCS_PCZ)
distance_pp (WCS_PX, WCS_PY, WCS_PCX, WCS_PCY, r)
z := fabs(WCS_PCZ)
DeltaR := DeltaZ * r / z
```

Assuming a constant Δz, the following conclusions can be drawn for Δr:

- Δr increases with increasing r.
- If the measurement plane is more or less perpendicular to the optical axis, Δr increases towards the image borders.
- At the point N, Δr is always equal to zero.
- Δr increases the more the measurement plane is tilted with respect to the optical axis.

The maximum acceptable deviation of the object's surface from the measurement plane, given a maximum value for the resulting displacement, can be derived by the following formula:

$$\Delta z = \Delta r \cdot \frac{z}{r} \tag{3.5}$$

The values for r and z can be determined as described above.

If you want to inspect an object that has a surface that consists of several parallel planes you can first use equation 3.5 to evaluate if the measurement errors stemming from the displacements are acceptable within your project or not. If the displacements are too large, you can calibrate the camera such that the measurement plane corresponds to, e.g., the uppermost plane of the object. Now, you can derive a pose for each plane, which is parallel to the uppermost plane simply by applying the operator set_origin_pose. This approach is also useful if objects of different thickness may appear on the assembly line. If it is possible to classify these objects into classes corresponding to their thickness, you can select the appropriate pose for each object. Thus, it is possible to derive accurate world coordinates for each object.

Note that if the plane in which the object lies is severely tilted with respect to the optical axis, and if the object has a significant thickness, the camera will likely see some parts of the object that you do not want to measure. For example, if you want to measure the top side of a cube and the plane is tilted, you will see the side walls of the cube as well, and therefore might measure the wrong dimensions. Therefore, it is usually best to align the camera so that its optical axis is perpendicular to the plane in which the objects are measured. If the objects do not have significant thickness, you can measure them accurately even if the plane is tilted.

What is more, it is even possible to derive world coordinates for an object's surface that consists of several non-parallel planes if the relation between the individual planes is known. In this case, you may define the relative pose of the tilted plane with respect to an already known measurement plane.

```
RelPose := [0, 3.2, 0, -14, 0, 0, 0]
```

Then, you can transform the known pose of the measurement plane into the pose of the tilted plane.

```
pose_to_hom_mat3d (FinalPose, HomMat3D)
pose_to_hom_mat3d (RelPose, HomMat3DRel)
hom_mat3d_compose (HomMat3D, HomMat3DRel, HomMat3DAdapted)
hom_mat3d_to_pose (HomMat3DAdapted, PoseAdapted)
```

Alternatively, you can use the operators of the family `hom_mat3d_..._local` to adapt the pose.

```
hom_mat3d_translate_local (HomMat3D, 0, 3.2, 0, HomMat3DTranslate)
hom_mat3d_rotate_local (HomMat3DTranslate, rad(-14), 'x', HomMat3DAdapted)
hom_mat3d_to_pose (HomMat3DAdapted, PoseAdapted)
```

Now, you can obtain world coordinates for points lying on the tilted plane, as well.

```
contour_to_world_plane_xld (Lines, ContoursTrans, CamParam, PoseAdapted, 1)
```

If the object is to complex to be approximated by planes, or if the relations between the planes are not known, it is not possible to perform precise measurements in world coordinates using the methods described in this section. In this case, it is necessary to use two cameras and to apply the HALCON stereo operators described in chapter 5 on page 117.

Chapter 4

3D Position Recognition of Known Objects

Estimating the 3D pose of an object is an important task in many application areas, e.g., during completeness checks or for 3D alignment in robot vision applications (see section 8.7.1 on page 185). HALCON provides multiple methods to determine the position or pose of known 3D objects.

The most general approach determines the pose of a known 3D object using at least three **corresponding points**, i.e., points with known 3D object coordinates for which the corresponding image coordinates are extracted. The approach is also known as "mono 3D" (section 4.1).

If a model of a known 3D object is available, **3D matching** can be applied to locate the object. The available 3D matching approaches perform a full 3D object recognition, i.e., they not only estimate a pose but first locate the object in the respective search data. The following approaches are available:

- *Shape-based 3D matching* (section 4.2 on page 95) can be used to locate a complex 3D object in a single 2D image. The model of the 3D object must be available as a Computer Aided Design (CAD) model in, e.g., DXF, OFF, or PLY format (see the Reference Manual entry of `read_object_model_3d` for details about the supported formats) and the object needs "hard geometric edges" to be recognized.
- *Surface-based 3D matching* (section 4.3 on page 104) can be used to quickly locate a complex 3D object in a 3D scene, i.e., in a set of 3D points that is available as a so-called 3D object model (see also section 2.3 on page 38). The model of the 3D object must be available also as a 3D object model and can be obtained either from a CAD model (see the Reference Manual entry of `read_object_model_3d` for details about the supported formats) or from a reference 3D scene that is obtained by a 3D reconstruction approach, e.g., stereo or sheet of light. Here, the object may also consist of a "smooth surface". Note that this approach is also known as "volume matching".

If the poses of simple 3D shapes like boxes, cylinders, spheres, or planes, which are called "3D primitives", are searched in a 3D scene that is available as a 3D object model, the **3D primitives fitting** (section 4.5 on page 111) can be used. There, the 3D scene is segmented into sub-parts so that into each sub-part a primitive of a selected type can be fitted. For each sub-part the fitting returns the parameters of the best fitting primitive, e.g., the pose for a fitted plane.

Sometimes, a full 3D object recognition or 3D matching is not necessary because you can estimate the pose of the object with simpler means. For example, if the object contains a characteristic planar part, you can estimate its 3D pose from a single image using **perspective matching**. Similar to the 3D matching approaches, they locate the object before they estimate its pose. Two approaches are available:

- The *calibrated perspective deformable matching* determines the 3D pose of a planar object that was defined by a template object by using automatically derived contours of the object (section 4.6 on page 114).
- The *calibrated descriptor-based matching* determines the 3D pose of a planar object that was defined by a template object by using automatically derived distinctive points of the object, which are called "interest points" (section 4.7 on page 114).

If a circle or rectangle is contained in the plane for which the 3D pose is needed, the pose estimation can be applied also by a simple **circle pose** (section 4.8 on page 115) **or rectangle pose** (section 4.9 on page 116) **estimation**. There, the circle or rectangle must be extracted from the image and the internal camera parameters as well as the dimensions of the circle or rectangle must be known.

An example application for pose estimation in a robot vision system is described in section 8.7.3 on page 187. Note that an introduction to the different 3D matching approaches can be found also in the Solution Guide I, chapter 11 on page 101. The approaches for the perspective matching are described in more detail in the Solution Guide II-B.

4.1 Pose Estimation from Points

If the internal camera parameters are known, the pose of an object can be determined by a call of the operator `vector_to_pose`.

The individual steps are illustrated based on the example program `%HALCONEXAMPLES%\solution_guide\3d_vision\pose_of_known_3d_object.hdev`, which determines the pose of a metal part with respect to a given world coordinate system.

First, the camera must be calibrated, i.e., the internal camera parameters and, if the pose of the object is to be determined relative to a given world coordinate system, the external camera parameters must be determined. See section 3.2 on page 61 for a detailed description of the calibration process. The world coordinate system can either be identical to the calibration plate coordinate system belonging to the calibration plate from one of the calibration images, or it can be modified such that it fits to some given reference coordinate system (figure 4.1). This can be achieved, e.g., by using the operator `set_origin_pose`

```
set_origin_pose (PoseOfWCS, -0.0568, 0.0372, 0, PoseOfWCS)
```

or if other transformations than translations are necessary, via homogeneous transformation matrices (section 2.1 on page 13).

```
pose_to_hom_mat3d (PoseOfWCS, camHwcs)
hom_mat3d_rotate_local (camHwcs, rad(180), 'x', camHwcs)
hom_mat3d_to_pose (camHwcs, PoseOfWCS)
```

With the homogeneous transformation matrix ${}^{c}\mathbf{H}_{w}$, which corresponds to the pose of the world coordinate system, world coordinates can be transformed into camera coordinates.

Then, the pose of the object can be determined from at least three points (control points) for which both the 3D object coordinates and the 2D image coordinates are known.

The 3D coordinates of the control points need to be determined only once. They must be given in a coordinate system that is attached to the object. You should choose points that can be extracted easily and accurately from the images. The 3D coordinates of the control points are then stored in three tuples, one for the x coordinates, one for the y coordinates, and the last one for the z coordinates.

In each image from which the pose of the object should be determined, the control points must be extracted. This task depends heavily on the object and on the possible poses of the object. If it is known that the object will not be tilted with respect to the camera the detection can, e.g., be carried out by shape-based matching (for a detailed description of shape-based matching, please refer to the Solution Guide II-B, section 3.2 on page 53).

Once the image coordinates of the control points are determined, they must be stored in two tuples that contain the row and the column coordinates, respectively. Note that the 2D image coordinates of the control points must be stored in the same order as the 3D coordinates.

In the example program, the centers of the three holes of the metal part are used as control points. Their image coordinates are determined with the HDevelop procedure `determine_control_points`,

```
procedure determine_control_points (Image: Intersections: : RowCenter,
                                    ColCenter)
```

which is part of the example program `%HALCONEXAMPLES%\solution_guide\3d_vision\pose_of_known_3d_object.hdev`.

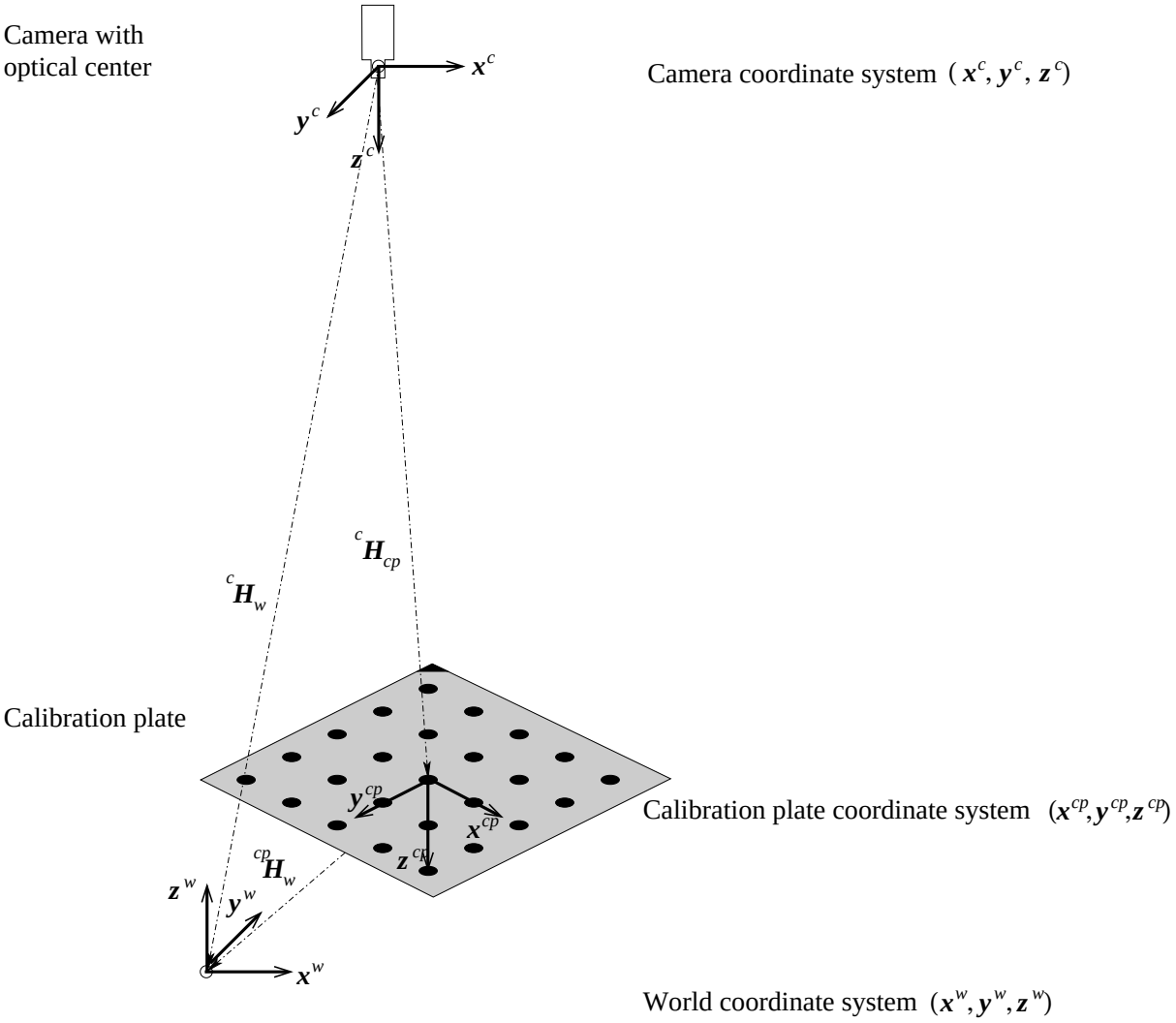


Figure 4.1: Determination of the pose of the world coordinate system.

Now, we simply call the operator `vector_to_pose`, passing the 3D object coordinates and the 2D image coordinates of the control points together with the internal camera parameters.

```
vector_to_pose (ControlX, ControlY, ControlZ, RowCenter, ColCenter, \
                CamParam, 'iterative', 'error', PoseOfObject, Errors)
```

If both the pose of the world coordinate system and the pose of the object coordinate system are known with respect to the camera coordinate system (see figure 4.2), it is easy to determine the transformation matrices for the transformation of object coordinates into world coordinates and vice versa:

$$
\begin{aligned}
{}^w\mathbf{H}_o &= {}^w\mathbf{H}_c \cdot {}^c\mathbf{H}_o && (4.1)\\
&= ({}^c\mathbf{H}_w)^{-1} \cdot {}^c\mathbf{H}_o && (4.2)
\end{aligned}
$$

where ${}^w\mathbf{H}_o$ is the homogeneous transformation matrix for the transformation of object coordinates into world coordinates and ${}^c\mathbf{H}_w$ and ${}^c\mathbf{H}_o$ are the homogeneous transformation matrices corresponding to the pose of the world coordinate system and the pose of the object coordinate system, respectively, each with respect to the camera coordinate system.

The transformation matrix for the transformation of world coordinates into object coordinates can be derived by:

$$
{}^o\mathbf{H}_w = ({}^w\mathbf{H}_o)^{-1} \qquad (4.3)
$$

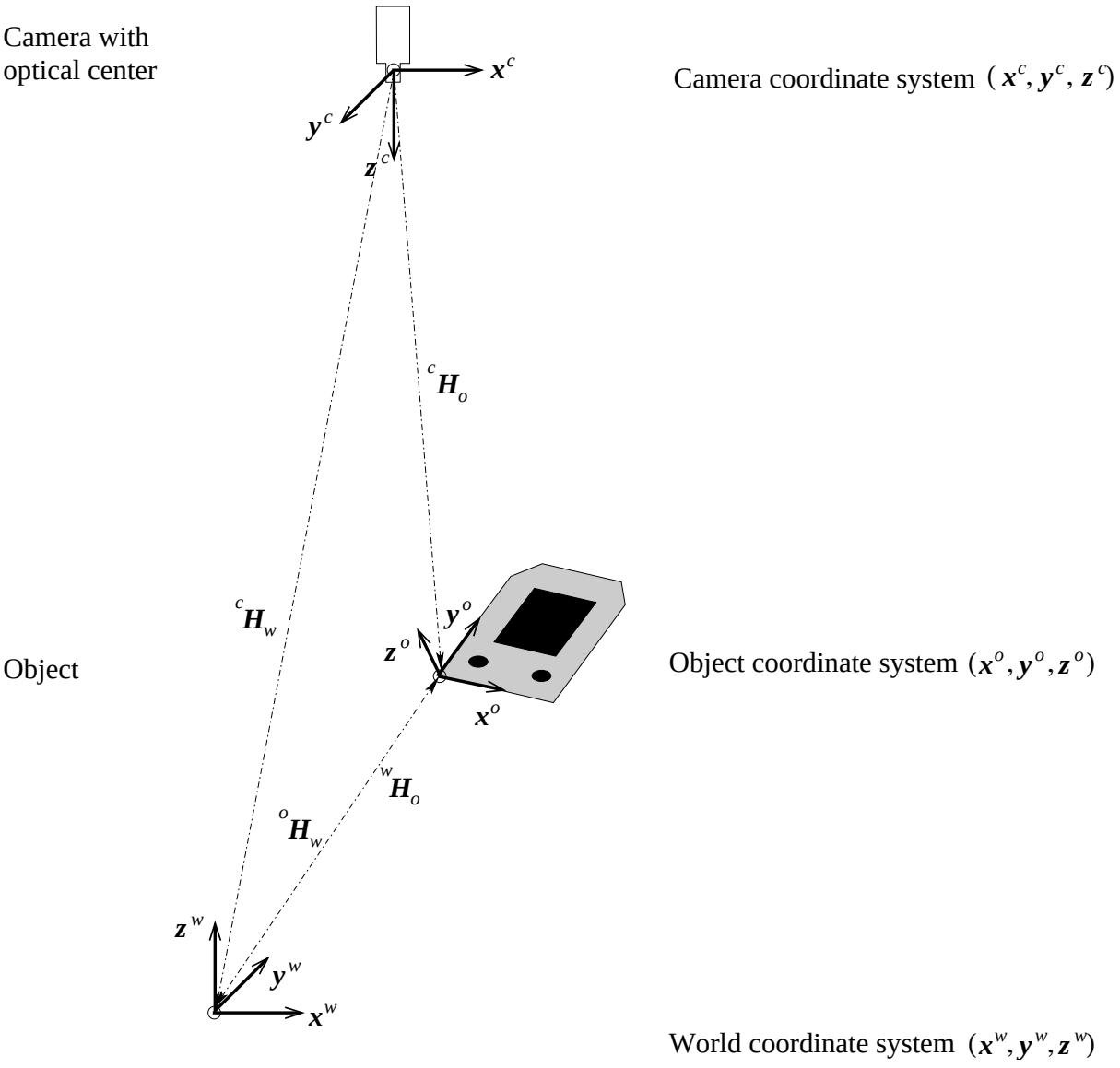


Figure 4.2: Pose of the object coordinate system and transformation between object coordinates and world coordinates.

The calculations described above can be implemented in HDevelop as follows. First, the homogeneous transformation matrices are derived from the respective poses.

```
pose_to_hom_mat3d (PoseOfWCS, camHwcs)
pose_to_hom_mat3d (PoseOfObject, camHobj)
```

Then, the transformation matrix for the transformation of object coordinates into world coordinates is derived.

```
hom_mat3d_invert (camHwcs, wcsHcam)
hom_mat3d_compose (wcsHcam, camHobj, wcsHobj)
```

Now, known object coordinates can be transformed into world coordinates with affine_trans_point_3d.

```
affine_trans_point_3d (wcsHobj, CornersXObj, CornersYObj, CornersZObj, \
                       CornersXWCS, CornersYWCS, CornersZWCS)
```

In the example program %HALCONEXAMPLES%\solution_guide\3d_vision\pose_of_known_3d_object.hdev, the world coordinates of the four corners of the rectangular hole of

the metal part are determined from their respective object coordinates. The object coordinate system and the world coordinate system are visualized as well as the respective coordinates for the four points (see figure 4.3).

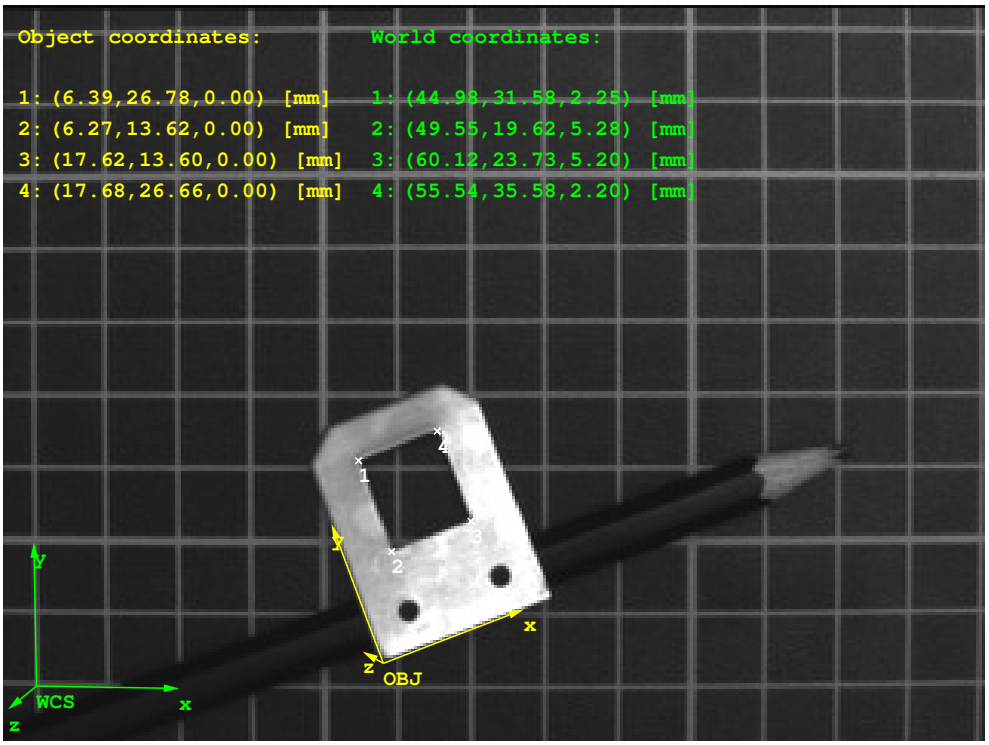


Figure 4.3: Object coordinates and world coordinates for the four corners of the rectangular hole of the metal part.

4.2 Pose Estimation Using Shape-Based 3D Matching

For the 3D pose estimation with shape-based 3D matching, a 3D shape model is generated from a 3D computer aided design (CAD) model. The 3D shape model consists of 2D projections of the 3D object seen from different views. To restrain the needed memory and runtime for the shape-based 3D matching, you should restrict the allowed pose range of the shape model and thus minimize the number of 2D projections that have to be computed and stored in the 3D shape model. Analogously to the shape-based matching of 2D structures described in the Solution Guide II-B, section 3.2 on page 53, the 3D shape model is used to recognize instances of the object in the image. But here, instead of a 2D position, orientation, and scaling, the 3D pose of each instance is returned.

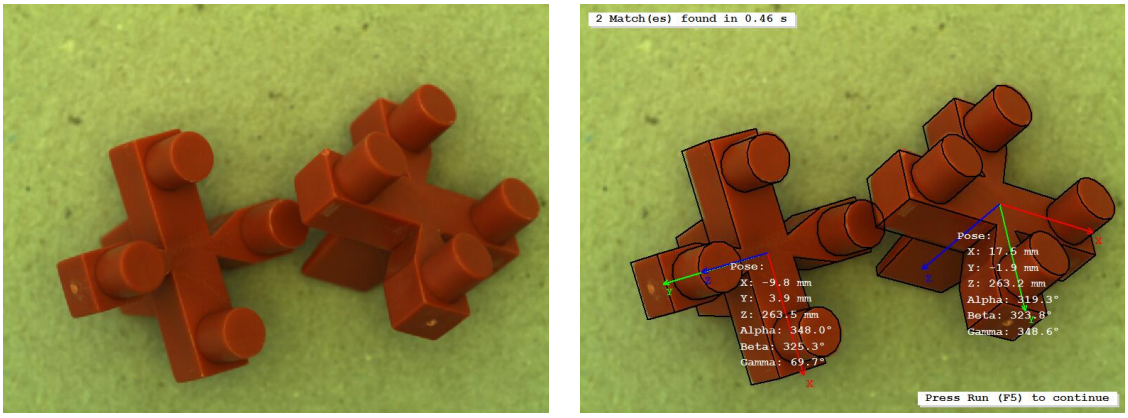


Figure 4.4: Shape-based 3D matching: (left) original image containing two tile spacers, (right) 3D shape model of the tile spacer projected in the image with the poses of the found model instances.

If you need the 3D pose of a planar object or a planar object part, we recommend to use the calibrated perspective deformable matching (see section 4.6 on page 114) or the calibrated descriptor-based matching (see section 4.7 on page 114). Both are significantly faster, because no 2D projections of the model must be computed. Instead, a single 2D model is derived from images.

In the following, the general proceeding for shape-based 3D Matching is introduced (see section 4.2.1), it is shown how to generally enhance the matching robustness and speed (see section 4.2.2 on page 99), and tips and tricks for the handling of specific problems are provided (see section 4.2.3 on page 101).

4.2.1 General Proceeding for Shape-Based 3D Matching

Shape-based 3D matching consists of the following basic steps:

- the 3D object model is accessed from file,
- the 3D shape model is created from it, and
- the 3D shape model is used to search the object in search images.

An example for the shape-based 3D matching of tile spacers is the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Matching\Shape-Based\create_shape_model_3d_lowest_model_level.hdev` (see figure 4.4 on page 95).

Step 1: Read the 3D object model

The 3D object model describing the search object is loaded in HALCON with the operator `read_object_model_3d`. It must be available as a CAD model in one of the supported formats, e.g., DXF, STL, or PLY. The list of supported CAD formats and tips on how to obtain a suitable model, including the specification of the requirements the CAD models must fulfill, are provided with the description of the operator in the Reference Manual. For additional tips on handling selected CAD formats, please contact your distributor.

```
read_object_model_3d ('tile_spacer.dxf', 0.0001, [], [], ObjectModel3DID, \
                      DXFStatus)
```

Step 2: Create the 3D shape model

The 3D shape model is created with the operator `create_shape_model_3d`. It needs the 3D object model and camera parameters as input. Additionally, a set of parameters has to be adjusted. The camera parameters can be obtained by a camera calibration as is described in detail in section 3.2 on page 61. In the example, the camera parameters are known and just assigned to the variable `CamParam`. Before creating the 3D shape model, it is recommended to prepare the 3D object model for the shape-based 3D matching using `prepare_object_model_3d`. Otherwise, the preparation is applied internally within `create_shape_model_3d`, which may slow down the application if the same 3D object model is used several times.

```
gen_cam_par_area_scan_division (0.0269462, -354.842, 1.27964e-005, \
                                1.28e-005, 254.24, 201.977, 512, 384, \
                                CamParam)
prepare_object_model_3d (ObjectModel3DID, 'shape_based_matching_3d', 'true', \
                         [], [])
create_shape_model_3d (ObjectModel3DID, CamParam, 0, 0, 0, 'gba', -rad(60), \
                       rad(60), -rad(60), rad(60), 0, rad(360), 0.26, 0.27, \
                       10, 'lowest_model_level', 3, ShapeModel3DID)
```

The 3D shape model 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.

An important task is to specify the pose range. To ease this task, imagine a sphere that surrounds the object. On the surface of the sphere, a camera is placed that looks at the object. Now, the pose range can be defined by restricting the position of the camera to a part of the sphere's surface. Additionally, the minimum and maximum distance of the camera to the object, i.e., the radii of different spheres, must be specified.

In the following, the position of the sphere relative to the object and the definition of the surface part are described. The position of the sphere is defined by placing its center at the center of the object's bounding box. The radius of the sphere corresponds to the distance of the camera to the center of the object. To define a specific part of the sphere's surface, the geographical coordinates longitude (λ) and latitude (φ) are used (see figure 4.5a). For these, minimum and maximum values are specified so that a quadrilateral on the sphere is obtained (see figure 4.5b).

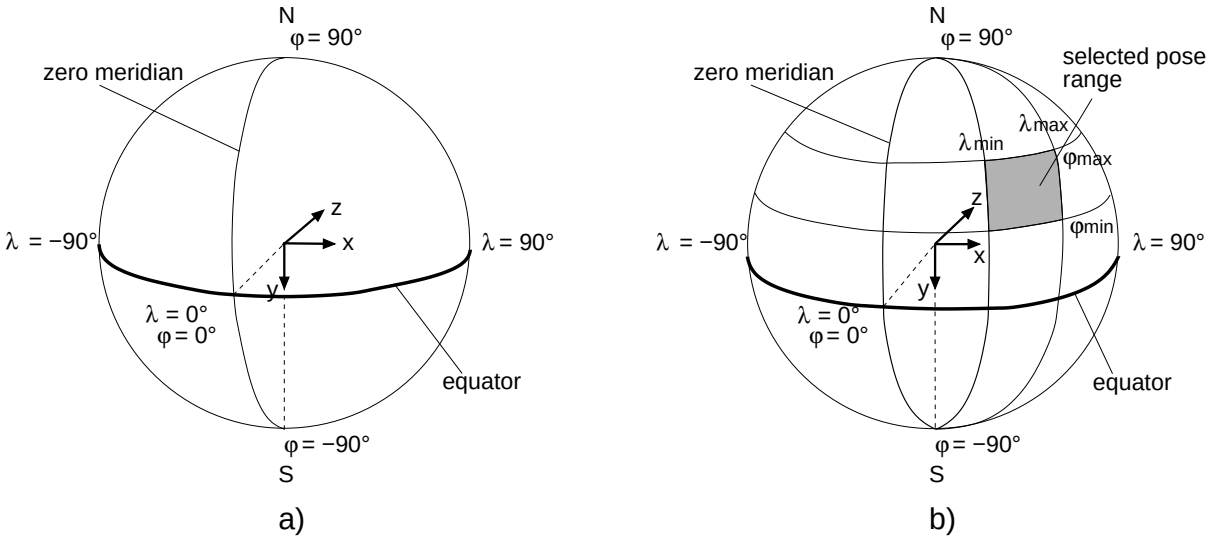


Figure 4.5: Geographic coordinate system: (a) the geographical coordinates longitude (λ) and latitude (φ) describe positions on the sphere's surface, (b) the minimum and maximum values for λ and φ describe a quadrilateral on the sphere, which defines the pose range.

To describe the orientation of the geographical coordinate system we introduce an object-centered coordinate system. This is obtained by moving the origin of the object coordinate system (CAD model) to the center of the sphere. The xz-plane of the object-centered coordinate system defines the equatorial plane of the geographical coordinate system. The north pole lies on the negative y-axis. The origin of the geographical coordinate system ($\lambda = \varphi = 0°$), i.e., the intersection of the equator with the zero meridian, lies in the negative z-axis.

To illustrate the above description, let us assume that we have the object model shown in figure 4.6a. For illustrative purpose additionally the object coordinate system is visualized. In figure 4.6b, the corresponding geographical coordinate system is shown together with a camera placed at its origin ($\lambda = \varphi = 0°$). Consequently, this camera view corresponds to a bottom view of the object.

Note that the coordinate system introduced here is only used to specify the pose range. The pose resulting from the shape-based 3D matching always refers to the original object coordinate system used in the CAD file and not to the center of the object's bounding box.

In most cases, the specification of the pose range can be simplified by changing the origin of the geographical coordinate system (i.e., the orientation of the sphere) such that it coincides with a mean viewing direction of the real camera to the object. This can be achieved by rotating the object-centered coordinate system, which defines the geographical coordinate system as described above. The rotation can be specified by passing the rotation angles to the parameters `RefRotX`, `RefRotY`, and `RefRotZ` of the operator `create_shape_model_3d`. That is, you can specify the pose range either by adjusting the longitude and latitude, leaving the origin of the sphere at its initial position, or by rotating the object-centered coordinate system to get a mean reference view and then specify the pose range in a more intuitive way (see figure 4.7), which is recommended in most cases.

Figure 4.8 on page 99 shows a rotation-symmetric object, for which a reference view is specified. Rotation-symmetric objects have the specific advantage that they have the same appearance from all directions that are perpendicular to their rotation axis. Thus, if the mean reference view is selected such that the camera view is perpendicular to the rotation axis (and the rotation axis crosses the poles of the sphere), the longitude range can collapse to a single value so that the number of calculated 2D projections is significantly reduced. That is, less memory is needed and the matching becomes faster. Here, the initial z-axis of the object-centered coordinate system corresponds to the rotation axis of the object. To obtain the intended reference view, the orientation of the sphere is changed by rotating the object-centered coordinate system by -90° around its x-axis. Now, the pose range for φ is specified like described before and the pose range for λ is restricted to 0°.

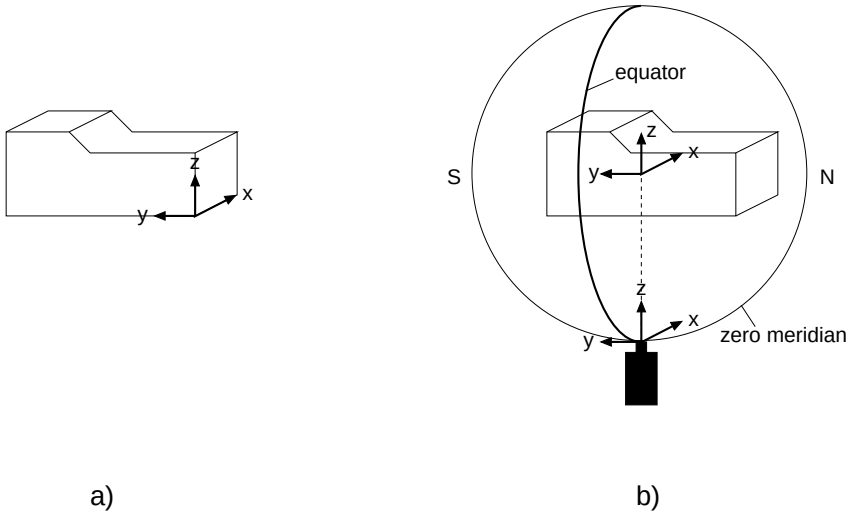
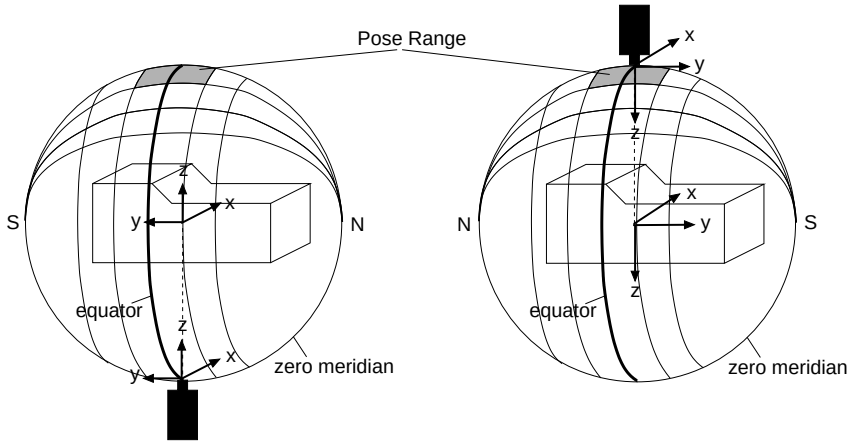


Figure 4.6: Object coordinate systems: (a) original object coordinate system of the CAD model, (b) object-centered geographical coordinate system obtained by moving the object to the center of the sphere: the camera is placed on the sphere's surface at the position $\lambda = \varphi = 0°$ and the object is placed at the center of the sphere. Note that in contrast to the previous image, the sphere is tilted, i.e., it is visualized in a way that the equator is vertical and the zero meridian is completely visible.



	λ min/max	φ min/max	RefRotX	RefRotY	RefRotZ
a)	180° +/− α	0° +/− α	0°	0°	0°
b)	0° +/− α	0° +/− α	180°	0°	0°

Figure 4.7: Specify pose range: (a) only by longitude and latitude, or (b) by additionally rotating the object-centered coordinate system to a reference view. Note that because of the rotation around the x-axis, the positions of the poles have changed.

Besides the definition of the pose range, also the camera roll angle, i.e., the allowed range for the rotation of the virtual camera around its z-axis, must be set. In most cases, it is recommended to allow a full circle for the camera roll angle. For details, we recommend to read the description of the operator `create_shape_model_3d` in the Reference Manual.

Step 3: Find the 3D shape model in search images

With the 3D shape model that was created by `create_shape_model_3d` or read from file by `read_shape_model_3d`, the object can be searched for in images. For the search, the operator `find_shape_model_3d` is applied.

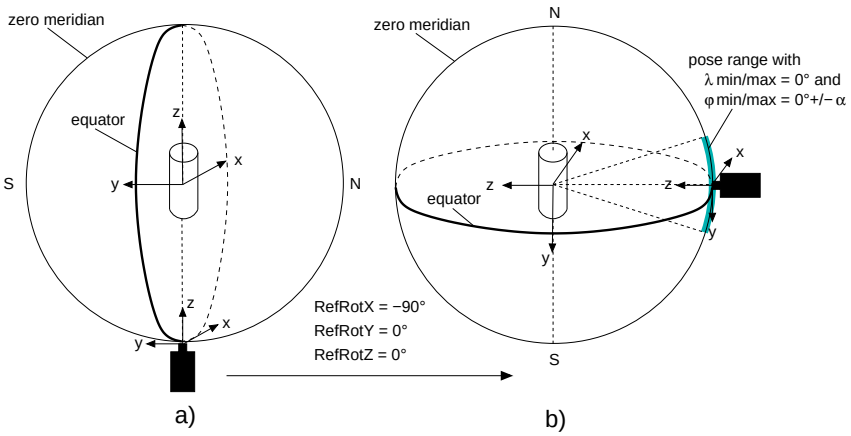


Figure 4.8: Change the origin of the geographical coordinate system for (a) a rotation-symmetric object with the z-axis of the object-centered coordinate system corresponding to the rotation axis such that (b) the z-axis becomes perpendicular to the rotation axis.

```
for I := 1 to NumImages by 1
    read_image (Image, 'tile_spacers/tile_spacers_color_' + I$'02')
    find_shape_model_3d (Image, ShapeModel3DID, 0.7, 0.85, 0, \
                         ['num_matches', 'max_overlap', 'border_model'], \
                         [3, 0.75, 'true'], Pose, CovPose, Score)
endfor
```

Several parameters can be set to control the search process. For detailed information, we recommend to read the description of the operator in the Reference Manual. The operator returns the pose of the matching model, the standard deviation of the pose, and the score of the found instances of the 3D shape model that describes how much of the model is visible in the image.

Besides the basic steps, it is often required to inspect the 3D object model or the 3D shape model, to re-use the 3D shape model, or to visualize the result of the matching. These steps are described in the Solution Guide I, chapter 11 on page 101.

4.2.2 Enhance the Shape-Based 3D Matching

The following sections generally show how to enhance the robustness (section 4.2.2.1) and speed (section 4.2.2.2) of shape-based 3D matching.

4.2.2.1 Enhance the Robustness

For a robust shape-based 3D matching it is important that the edges of the object are clearly visible in the image. Thus, the following general tips may help you to enhance your application already when acquiring the images of the object:

- If possible, use a background with a good contrast to the object, so that the background can be clearly separated from the object.
- Carefully adjust the lighting for the image acquisition.

  To get clearly visible edges of the object in your images, take special care of the lighting conditions. In general, the edges that are included in the 3D shape model should also be visible in the image. The edges that are included in the model can be adjusted with the generic parameter `min_face_angle` of `create_shape_model_3d`. The effect of this parameter can be inspected by visualizing the resulting edges with the procedure `inspect_object_model_3d`, which can be found in the example program `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\3d_matching_clamps.hdev`.

- If possible, use multi-channel images.

  Multi-channel, e.g., color images contain more information and thus typically lead to a more robust edge extraction. An especially robust edge extraction can be obtained if color images are used and the object is illuminated from different directions by three differently colored, typically red, green and blue, light sources (see figure 4.9).

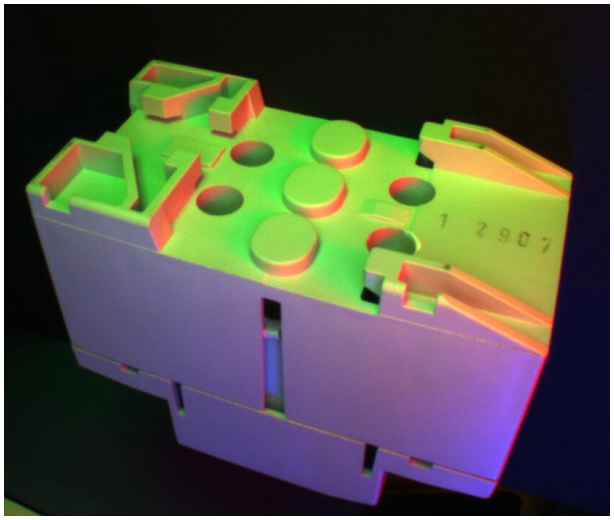


Figure 4.9: A fuse illuminated from three different directions by three differently colored light sources.

4.2.2.2 Enhance the Speed

There are several means to speed up the shape-based 3D matching:

- Use a homogeneous background during the image acquisition.

  Generally, you should try to adjust the lighting for the image acquisition so that the edges of the object but no surface texture are visible in the images.

- Reduce the resolution of the image.

  A reduced resolution of the image can speed up the online as well as the offline phase of the shape-based 3D matching, because less 2D projections have to be generated. Note that if you reduce the resolution of the images (e.g., from 1 Megapixel to 640x480 pixels), you must also change the camera parameters used for the creation of the 3D shape model by adapting `Sx`, `Sy`, `Cx`, `Cy`, `ImageWidth`, and `ImageHeight` (see the description of `write_cam_par` in the Reference Manual). For example, if the image is scaled down by a factor of 0.5, `Sx` and `Sy` must be multiplied by 2, whereas `Cx`, `Cy`, `ImageWidth`, and `ImageHeight` must be multiplied by 0.5.

- Use a region of Interest.

  Using a region of interest, you can speed up the search. 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 Solution Guide I, chapter 3 on page 25 or the Solution Guide II-B, section 2.1.2 on page 18.

- Restrict the pose range.

  The more the pose range is restricted while creating a 3D shape model with `create_shape_model_3d`, the faster is the search process. But note that only those object instances are found that correspond to the selected pose range.

- Eliminate unnecessary edges from the 3D shape model.

  If edges are contained in the 3D shape model that are not visible in the search image, the performance, i.e., the robustness and speed, of the shape-based 3D matching decreases. Especially for objects that contain curved surfaces, which are approximated by multiple planar faces in the 3D object model, the generic parameter `min_face_angle` should be adjusted within `create_shape_model_3d` to eliminate unnecessary edges from the 3D shape model. To check which edges are visible with a specific minimum face angle, you can display the corresponding contours with the operator `project_object_model_3d`.

- Select the value for the number of used pyramid levels as large as possible.

  In `create_shape_model_3d` as well as in `find_shape_model_3d`, the number of used pyramid levels can be set by the parameter `num_levels`. To speed up the search process, the value should be as large as possible, but the object should be still recognizable in the model. To check a view on the object in a specific pyramid level, you can query the corresponding contours by the operator `get_shape_model_3d_contours`.

- Disable the pregeneration of the model views on lower pyramid levels.

  When specifying a large pose range, the number of model views on lower pyramid levels may become very large, which leads to a slow model generation and high memory consumption. To speed up the model generation, you can disable the pregeneration of the model views on lower pyramid levels using the generic parameter `'lowest_model_level'` within `create_shape_model_3d`. Note that you nevertheless obtain a high accuracy during the search with `find_shape_model_3d`, because the pose is still refined (but now on the fly) on the original pyramid level.

- Select the value for `MinScore` as large as possible.

  In `find_shape_model_3d` you can adjust the parameter `MinScore`. A large `MinScore` speeds up the search process, but allows less invisible edges for the object, so that possibly some objects are not recognized.

- Select the value for `Greediness` as large as possible.

  In `find_shape_model_3d` you can adjust the parameter `Greediness`. A large `Greediness` speeds up the search process. But because the search becomes less robust, possibly some objects are not recognized.

- Adjust the pose refinement.

  In `find_shape_model_3d` you can adjust the generic parameter `pose_refinement`. If it is set to `none`, the search is fast but the pose is determined with limited accuracy. A trade-off between runtime and accuracy is to set the pose refinement to `least_squares_high`. For complex models with a large number of faces it is reasonable to speed up the pose refinement by splitting it such that some of the needed calculations are already performed during the creation of the model. Thus, the generic parameter `'fast_pose_refinement'` of `create_shape_model_3d` is by default set to `'true'`. This leads to a faster matching but also to a higher memory consumption. If the storage is more critical than the speed of the matching, you can set the parameter to 'false'.

- Check the setting of `border_model`.

  In `find_shape_model_3d` you can adjust the generic parameter `border_model`. If it is set to `true`, also objects that extend beyond the image borders can be found. Because this is rather time-consuming if you only search for objects that are completely contained in the image, we recommend to leave the default value `false` unchanged.

- Speed up the visualization.

  When visualizing the 3D object model or the 3D shape model by projecting them into the image using `project_object_model_3d` or `project_shape_model_3d`, respectively, you can speed up the visualization by setting the parameter `HiddenLineRemoval` to `false`. Then, also those edges of the model are visualized that would be hidden by faces.

4.2.3 Tips and Tricks for Problem Handling

In section 4.2.2 on page 99, general tips to enhance the robustness and speed of shape-based 3D matching were listed. In the following, the focus is on the handling of specific problems, i.e., possible reasons for an unsuccessful, erroneous, or very slow recognition or model generation are introduced. In particular, possible reasons for problems and tips to solve them are provided for the cases that

- the model generation is very slow (see section 4.2.3.1),
- the recognition is not successful, i.e., the object is not found (see section 4.2.3.2),
- the object is found, but the estimated object pose is wrong (see section 4.2.3.3 on page 103),
- the object is found in the right pose, but the pose is estimated with low accuracy (see section 4.2.3.4 on page 103), or
- the recognition is successful but very slow (section 4.2.3.5 on page 103).

4.2.3.1 The Model Generation is Very Slow

The computation time of the model generation increases quadratically with the number of faces in the CAD model. Thus, if the model generation with `create_shape_model_3d` is very slow, most possibly the CAD model is too complex. Note that in most cases, a very coarse model is sufficient to locate a 3D object with shape-based 3D matching. Thus, in case of a very slow model generation, you should eliminate all unimportant details from your model using suitable CAD software. Alternatively, you can also increase the value of the generic parameter `'lowest_model_level'` to work with a coarser model. Additionally, only the part of the object that is relevant for the search should be contained in the model. Thus, sometimes further modifications of the CAD model using suitable CAD software might be necessary.

4.2.3.2 The Object is not Found

If the object is not found with `find_shape_model_3d`, typically the reason can be found in at least one of the following problems:

- The value of the parameter `MinScore` was chosen too large. Check if the object can be found if matches with a smaller score are accepted. But note that the possibility of finding false matches increases with a decreasing `MinScore` value.
- The value of the parameter `Greediness` was chosen too large. A large value leads to a fast but less robust search. Check if the object can be found with a smaller `Greediness` value.
- The value of the parameter `NumLevels` was estimated or chosen too large. Note that the shape representation of the views on the highest pyramid level must still be recognizable and must contain enough model points. You can visually check the views on the specific pyramid levels using `get_shape_model_3d_contours`.
- The value of the generic parameter `'min_face_angle'` was chosen too small when generating the 3D shape model with `create_shape_model_3d`. Thus, the 3D shape model contains also non-visible object edges. Note that after creating a model, you can visually check the model edges on the specific pyramid levels using `get_shape_model_3d_contours`. Before creating a model, you can display the contours of the underlying 3D object model with a specific minimum face angle using the operator `project_object_model_3d`.
- The chosen reference pose or the pose range are not correct. Check whether the created views cover the desired pose range. Note that you can query the number of created views for each pyramid level using `get_shape_model_3d_params` (setting `GenParamName` to `'num_views_per_level'`). Then, you can visually check selected views out of this range of views using `get_shape_model_3d_contours`.
- The value of the parameter `MinContrast` was chosen too large when generating the 3D shape model with `create_shape_model_3d`. Thus, edges that belong to the object are not extracted in the search image and the matching score decreases. Check the value of `MinContrast` by extracting edges in the search image using `edges_image` with the parameter `Filter` set to `'sobel_fast'` and the parameters `Low` and `High` set to the value of `MinContrast`.
- The camera parameters are not accurate enough. Thus, the projected model and the imaged object do not accurately fit together. If possible, improve the accuracy of the calibration. Otherwise, stop the search on a higher pyramid level where the differences between the projected model and the imaged object are small enough. To stop the search on a higher pyramid level, the parameter `NumLevels` is set as a tuple consisting of the highest and lowest used pyramid levels.
- The projected model and the imaged object do not accurately fit together because the CAD file is not modeled accurately enough or the objects do not exactly correspond to the model, e.g., because of tolerances at the fabrication. In the first case, if possible, improve the accuracy of the CAD model. Otherwise, stop the search on a higher pyramid level where the differences between the projected model and the imaged object are small enough. To stop the search on a higher pyramid level, the parameter `NumLevels` is set as a tuple consisting of the highest and lowest used pyramid levels.
- Some of the object edges are no “sharp” edges. Model the round edges in the CAD model or stop the search on a higher pyramid level where the differences between the projected model and the imaged object are small enough. To stop the search on a higher pyramid level, the parameter `NumLevels` is set as a tuple consisting of the highest and lowest used pyramid levels.
- The object edges are not visible in the image. To enhance the visibility of the edges during the image acquisition, follow the advice for a robust shape-based matching that is given in section 4.2.2.1 on page 99.

4.2.3.3 The Object is Found in a Wrong Pose

If the object is found with `find_shape_model_3d`, but the estimated pose is wrong, typically the reason can be found in at least one of the following problems:

- The value of the parameter `MinScore` was chosen too small so that false matches could be found. Check if the object still can be found with a larger `MinScore` value.
- The value of the generic parameter `'min_face_angle'` was chosen too large when generating the 3D shape model with `create_shape_model_3d`. Thus, the 3D shape model does not contain all visible object edges. Note that after creating a model, you can visually check the model edges on the specific pyramid levels using `get_shape_model_3d_contours`. Before creating a model, you can display the contours of the underlying 3D object model with a specific minimum face angle using the operator `project_object_model_3d`.
- The value of the parameter `MinContrast` was chosen too small when generating the 3D shape model with `create_shape_model_3d`. This leads to clutter edges in the search image. Check the value of `MinContrast` by extracting edges in the search image using `edges_image` with the parameter `Filter` set to `'sobel_fast'` and the parameters `Low` and `High` set to `MinContrast`.
- The background contains too much clutter. To solve this problem for one-channel images, you can set `'metric'` to `'ignore_part_polarity'`. But if possible, you should use another background, which is more homogeneous. If the background cannot be changed, you can try to optimize its appearance in the images by using a diffuse lighting source and by adjusting the light's direction. Note that you can check the background by extracting edges in the search image using `edges_image` with the parameter `Filter` set to `'sobel_fast'` and the parameters `Low` and `High` set to the value of `MinContrast`.
- The pose range contains degenerated views, i.e., views that do not significantly represent the object anymore. For example, if a cube is viewed exactly orthogonal to one of its faces, the view collapses from a perspective representation of a 3D cube to a simple square. Such a view may lead to many false matches. An even more extreme example is an exact side-view on a flat object. There, the view may collapse to a straight line. Such a view would lead to many matches in any search image, even if the actual 3D object is not contained. Therefore, you should limit the pose range as much as possible and ensure that no degenerated views are contained in the pose range.
- There are too many clutter edges around the object, which is typical for some bin picking applications where objects touch or overlap each other. Try to separate the objects, e.g., by dumping the objects onto a plane and/or shaking the bin or plane to equally distribute the objects.

4.2.3.4 The Object Pose is Estimated With Low Accuracy

If the object is found with `find_shape_model_3d`, but the accuracy of the estimated object pose is low, typically the reason can be found in at least one of the following problems:

- Some of the object edges are no "sharp" edges. Model the round edges in the CAD model or stop the search on a higher pyramid level where the differences between the projected model and the imaged object are small enough. To stop the search on a higher pyramid level, the parameter `NumLevels` is set as a tuple consisting of the highest and lowest used pyramid levels.
- The camera parameters are not accurate enough. Thus, the projected model and the imaged object do not accurately fit together. If possible, improve the accuracy of the calibration.
- The projected model and the imaged object do not accurately fit together because the CAD file is not modeled accurately enough or the objects do not exactly correspond to the model, e.g., because of tolerances at the fabrication. In the first case, if possible, improve the accuracy of the CAD model.

4.2.3.5 The Object Recognition is Very Slow

If the object is found with `find_shape_model_3d`, but the recognition is very slow, typically the reason can be found in at least one of the following problems:

- The value of the parameter `Greediness` was chosen too small. A larger value speeds up the search, but because the search becomes less robust, possibly some objects are not recognized. Thus, you should check if the object can still be found with a larger `Greediness` value.

- The value of the parameter `MinScore` was chosen too small so that many false matches could be found. Check if the object still can be found with a larger `MinScore` value.
- The value of the parameter `MinContrast` was chosen too small when generating the 3D shape model with `create_shape_model_3d`. This leads to clutter edges in the search image. These clutter edges generate many matching candidates that must be examined. Check the value of `MinContrast` by extracting edges in the search image using `edges_image` with the parameter `Filter` set to `'sobel_fast'` and the parameters `Low` and `High` set to the value of `MinContrast`.
- The CAD model is very complex and `'fast_pose_refinement'` was set to `'false'` when generating the 3D shape model with `create_shape_model_3d`. As the computation time of the least-squares refinement increases quadratically with the number of faces in the CAD model, you can speed up the recognition by eliminating all important details from the model using your CAD software and setting `'fast_pose_refinement'` to `'true'`.
- The image size, especially the extent of the search object in pixels, is very large so that the number of model views on lower pyramid levels becomes very large, too. If you cannot work with images of a reduced resolution as is described in section 4.2.2.2 on page 100, we strongly recommend to restrict the search in the image to an ROI and to increase the value of `'lowest_model_level'` when generating the 3D shape model with `create_shape_model_3d`.
- The pose range is too large. When generating the 3D shape model with `create_shape_model_3d`, restrict the pose range as much as possible, especially take care of `DistMin`. Additionally, for objects with multiple stable poses you should use multiple models. That is, instead of generating one model that covers the complete pose range, multiple models each covering one stable pose should be used. Note that you can also restrict the pose range by exploiting rotational symmetries of the object (see section 4.2.1 on page 97).
- The background contains too much clutter so that the clutter edges in the search image lead to many matching candidates that must be examined. To solve this problem for one-channel images, you can set `'metric'` to `'ignore_part_polarity'`. But if possible, you should use another background, which is more homogeneous. If the background cannot be changed, you can try to optimize its appearance in the images by using a diffuse light source and by adjusting the light's direction. Note that you can check the background by extracting edges in the search image using `edges_image` with the parameter `Filter` set to `'sobel_fast'` and the parameters `Low` and `High` set to the value of `MinContrast`.

4.3 Pose Estimation Using Surface-Based 3D Matching

For the 3D pose estimation with surface-based matching, a surface model is generated from a 3D object model that was obtained either from a 3D computer aided design (CAD) model or from a 3D reconstruction approach, e.g., stereo vision (chapter 5 on page 117) or sheet of light (chapter 6 on page 147). The surface model consists of a set of 3D points and the points' normal vectors. That is, the corresponding information must be provided (at least implicitly) by the 3D object model. In contrast to the shape-based 3D matching, the instances of the object are not located in images but in a 3D scene, i.e., in a set of 3D points that is provided as another 3D object model and which can be obtained by a 3D reconstruction approach, too.

In the following, the general proceeding for surface-based 3D matching is introduced (see section 4.3.1).

4.3.1 General Proceeding for Surface-Based 3D Matching

Surface-based 3D matching consists of the following basic steps:

- access the 3D object model needed for the creation of the surface model,
- create the surface model from it,
- access the 3D object model that represents the search data, and
- use the surface model to search the object in the search data.

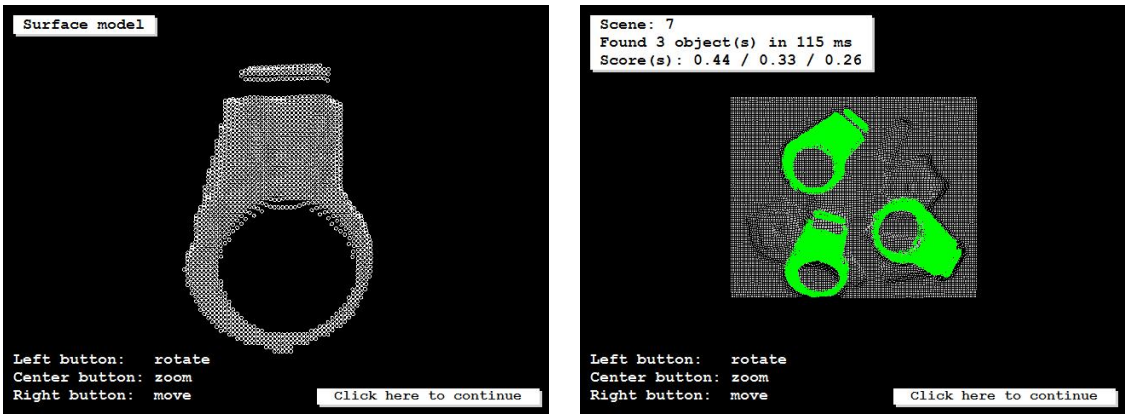


Figure 4.10: Surface-based matching: (left) 3D model of engine cover, (right) model instances found in a 3D scene.

An example for the surface-based 3D matching of engine covers (see figure 4.10) is the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Matching\Surface-Based\find_surface_model.hdev`.

Step 1: Access the 3D object model needed for the creation of the surface model

In contrast to shape-based 3D matching, for surface-based matching the 3D object model does not need to be available as CAD model but can also be derived by a 3D reconstruction. If a 3D object model is available as a CAD model or if the 3D object model was saved to file after an offline 3D reconstruction, it can be accessed using `read_object_model_3d`. Suitable file formats are, e.g., DXF, OFF, PLY, or OM3. The format OM3 is a HALCON-specific format for 3D object models that can be derived from the results of HALCON's 3D reconstruction approaches (see figure 2.29 on page 42 in chapter 1). Tips on how to obtain a suitable model, including the specification of the requirements the CAD models must fulfill, are provided with the description of the operator `read_object_model_3d` in the Reference Manual.

In the example program, the 3D object model is not read from file but is derived from X, Y, and Z images that were obtained by a specific 3D sensor: a "time-of-flight" (TOF) camera. To get the 3D object model of a single engine cover, a region containing a single engine cover is extracted by a blob analysis from the Z image (see figure 4.11) and the Z image is reduced to the corresponding ROI.

```
read_image (Image, ImagePath + 'engine_cover_xyz_01')
decompose3 (Image, Xm, Ym, Zm)
threshold (Zm, ModelZ, 0, 650)
connection (ModelZ, ConnectedModel)
select_obj (ConnectedModel, ModelROI, [10, 9])
union1 (ModelROI, ModelROI)
reduce_domain (Xm, ModelROI, Xm)
```

The X, Y, and Z images are now transformed into a 3D object model using the operator `xyz_to_object_model_3d`. This 3D object model is needed to create the surface model that will serve as model for the matching.

```
xyz_to_object_model_3d (Xm, Ym, Zm, ObjectModel3DModel)
```

Note that for surface-based matching information about the coordinates of the 3D points and their normals is needed. Thus, if a 3D object model is obtained from a CAD model or from multi-view stereo, the normals or alternatively a triangular or polygon mesh should be contained in the 3D object model. If the 3D object model does not already contain normals, e.g., because it has been created from X, Y, and Z images like in the example program, the normals are automatically determined by `create_surface_model`. Note that in this case, the orientation of the normals is ambiguous (see next step).

Step 2: Create the surface model

The operator `create_surface_model` creates a surface model by sampling the 3D object model with a certain distance. The sampling distance can be adjusted with the parameter `RelSamplingDistance`. Note that a smaller

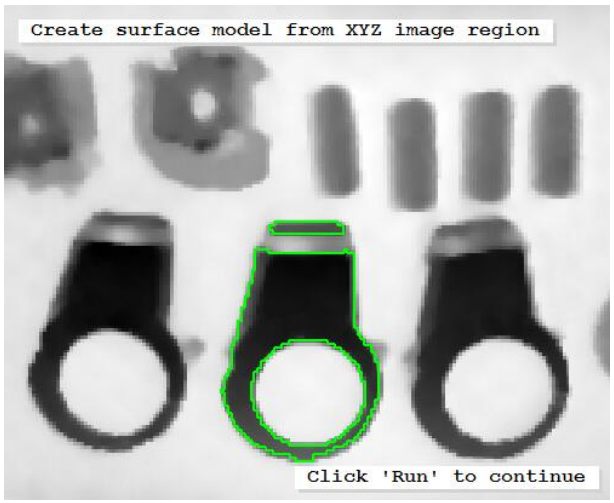


Figure 4.11: A single engine cover extracted from the Z image.

value leads to a slower but more robust matching, whereas a larger value speeds up the matching but at the same time decreases the robustness.

```
create_surface_model (ObjectModel3DModel, 0.03, [], [], SFM)
```

The correct match will only be found if the normals of the model and of the search object point in the same direction. If the normals must be derived by `create_surface_model` because they are not available in the 3D object model, the orientation of the normals is ambiguous and consequently, a suitable orientation of the normals cannot be ensured. To cope with this, especially if the orientation of the search object may vary considerably, two models should be created with the generic parameter *'model_invert_normals'* set to *'true'* and *'false'*, respectively. Both models should then be used for matching and the match with the higher score should be selected.

Step 3: Access the 3D object model that represents the search data

Similar to the 3D object model that was needed for the creation of the surface model, the 3D object models in which the object of interest are searched must be accessed, i.e., they are read from file using `read_object_model_3d` or they are derived online by a 3D reconstruction. In the example program, the 3D object models are derived again from X, Y, and Z images using the operator `xyz_to_object_model_3d`. Note that a blob analysis is applied to remove the background plane from the search data.

```
NumImages := 10
for Index := 2 to NumImages by 1
    read_image (Image, ImagePath + 'engine_cover_xyz_' + Index$'02')
    decompose3 (Image, X, Y, Z)
    threshold (Z, SceneGood, 0, 666)
    reduce_domain (X, SceneGood, XReduced)
    xyz_to_object_model_3d (XReduced, Y, Z, ObjectModel3DSceneReduced)
```

Step 4: Find the surface model in the search data

With the surface model that was created by `create_surface_model` or read from file by `read_surface_model`, the object can be searched for in the search data using the operator `find_surface_model`.

```
    find_surface_model (SFM, ObjectModel3DSceneReduced, 0.05, 0.3, 0.2, \
                        'true', 'num_matches', 10, Pose, Score, \
                        SurfaceMatchingResultID)
```

Several parameters can be set to control the search process. For detailed information, we recommend to read the description of the operator in the Reference Manual. For each model instance that could be located by the matching, the operator returns the pose and a score that describes the quality of the match.

After the matching, the result can be visualized. Here, for each match that exceeds a certain score, the 3D object model is transformed with the corresponding pose and the transformed 3D object models of a specific search scene are stored in the tuple `ObjectModel3DResult`.

```
ObjectModel3DResult := []
for Index2 := 0 to |Score| - 1 by 1
    if (Score[Index2] < 0.11)
        continue
    endif
    CPose := Pose[Index2 * 7:Index2 * 7 + 6]
    rigid_trans_object_model_3d (ObjectModel3DModel, CPose, \
                                 ObjectModel3DRigidTrans)
    ObjectModel3DResult := [ObjectModel3DResult, \
                            ObjectModel3DRigidTrans]
endfor
```

The visualization is then applied using `visualize_object_model_3d`. Actually, the transformed models are displayed together with the original 3D scene (see figure 4.10 on page 105), i.e., with the 3D object model that is obtained from the X, Y, and Z images from which also the search data was obtained, but this time no blob analysis is applied to remove the background plane.

```
xyz_to_object_model_3d (X, Y, Z, ObjectModel3DScene)
NumResult := |ObjectModel3DResult|
tuple_gen_const (NumResult, 'green', Colors)
tuple_gen_const (NumResult, 'circle', Shapes)
tuple_gen_const (NumResult, 3, Radii)
visualize_object_model_3d (WindowHandle, [ObjectModel3DScene, \
                           ObjectModel3DResult], [], PoseOut, \
                            ['color_' + [0,Indices],'point_size_0'], \
                            ['gray',Colors,1.0], Message, [], \
                            Instructions, PoseOut)
```

Besides the basic steps, it is often required to inspect the 3D object model, to re-use the surface model, or to visualize the result of the matching. These steps are described in the Solution Guide I, chapter 11 on page 101.

The procedure `debug_find_surface_model` can be used to further visualize and debug the matching and the used parameters.

4.4 Pose Estimation Using Deformable Surface-Based 3D Matching

Deformable surface-based 3D matching allows to find 3D objects in 3D scenes even if the 3D objects are deformed to a certain degree (see figure 4.12).

For the 3D pose estimation with deformable surface-based 3D matching, a deformable surface model is generated from a 3D object model that was obtained either from a 3D computer aided design (CAD) model, from a 3D reconstruction approach, e.g., stereo vision (chapter 5 on page 117) or sheet of light (chapter 6 on page 147), or directly by a 3D sensor. The deformable surface model consists of a set of 3D points and the points' normal vectors. That is, the corresponding information must be provided (at least implicitly) by the 3D object model. In contrast to the shape-based 3D matching, the instances of the object are not located in images but in a 3D scene, i.e., in a set of 3D points that is provided as another 3D object model and which can be obtained by one of the above mentioned approaches or sensors.

4.4.1 General Proceeding for Deformable Surface-Based 3D Matching

Deformable surface-based 3D matching consists of the following basic steps:

- provide the 3D object model for the creation of the deformable surface model,

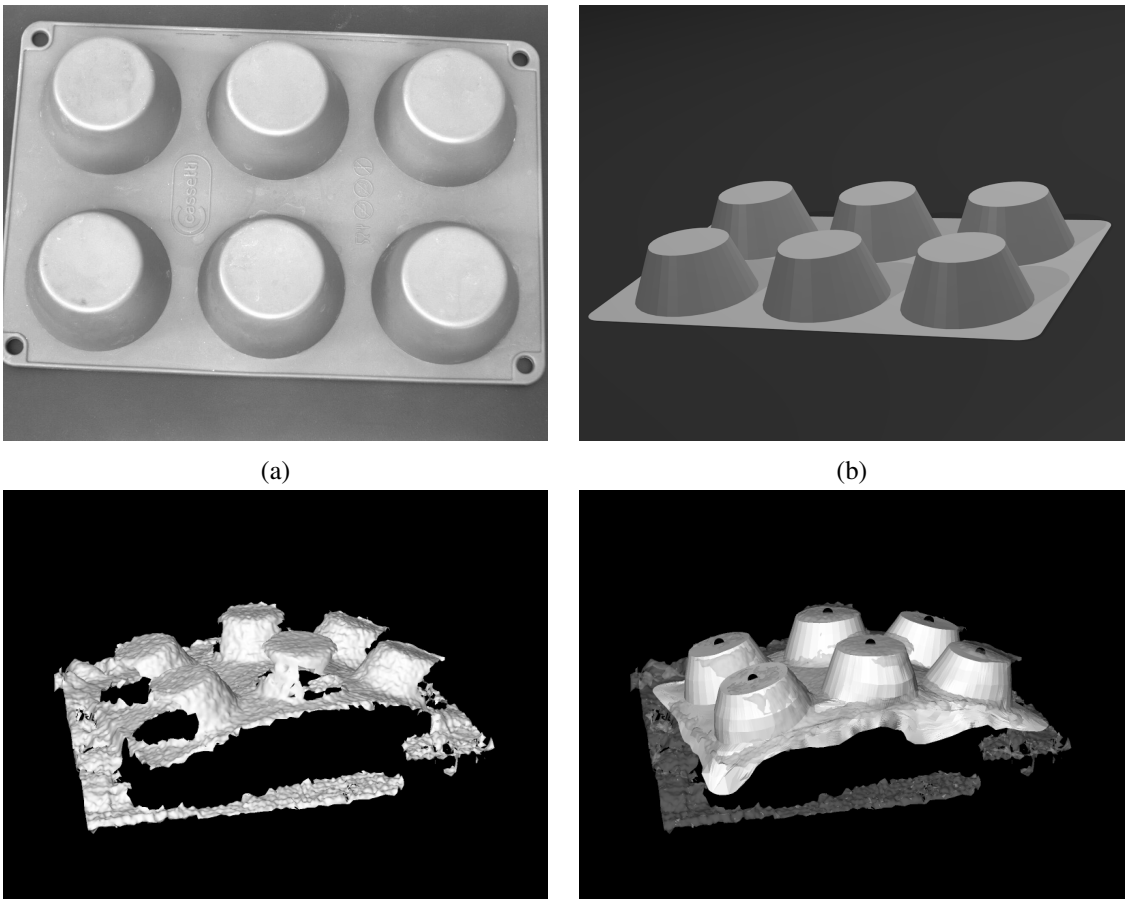


Figure 4.12: Deformable Surface-based matching: (a) The silicone baking mold used as test object, (b) a 3D object model of the silicone baking mold, (c) a 3D scene that shows a deformed baking mold, and (d) matching result (deformed object and reference points).

- create the deformable surface model with `create_deformable_surface_model`,
- optionally define some reference points with `add_deformable_surface_model_reference_point`,
- optionally extend the range of deformations that the deformable surface model is able to cope with,
- provide a 3D scene that represents the search data,
- search for the (possibly) deformed 3D object in the search data with `find_deformable_surface_model`, and

The HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Matching\Deformable-Surface-Based\find_deformable_surface_model.hdev` shows how to use the deformable surface-based 3D matching to find even deformed silicone baking molds (see figure 4.12a).

Step 1: Provide the 3D object model needed for the creation of the deformable surface model

In contrast to shape-based 3D matching, for deformable surface-based matching, the 3D object model does not need to be available as CAD model but can also be provided by a 3D sensor or be derived by a 3D reconstruction. For more information on the possible sources for 3D data, see section 4.3.1 on page 104.

In the example program, the 3D object model is provided as CAD model (see figure 4.12b). This kind of data source was mainly chosen for visualization reasons. With this, it is easier to distinguish the deformed model from the search scene.

Note that for deformable surface-based matching information about the coordinates of the 3D points and their normals is needed. Thus, if a 3D object model is obtained from a CAD model or from multi-view stereo, the normals or a triangular or polygon mesh should be contained in the 3D object model.

Step 2: Create the deformable surface model with `create_deformable_surfabe_model`

The operator `create_deformable_surface_model` creates a deformable surface model by sampling the 3D object model with a certain distance. The sampling distance can be adjusted with the parameter `RelSamplingDistance`. Note that a smaller value leads to a slower but more robust matching, whereas a larger value speeds up the matching but at the same time decreases the robustness.

```
create_deformable_surface_model (ObjectModel3DReference, 0.03, 'stiffness', \
                                 0.85, DeformableSurfaceModel)
```

If the 3D object model does not contain normals, they are determined by `create_deformable_surface_model` automatically. Note that in this case, the orientation of the normals is ambiguous. Internally, they are automatically oriented such that all normals have a positive z component.

The correct match will only be found if the normals of the model and of the search object point in the same direction. If the orientation of the search object may vary considerably, some effort should be made to orient the normals of the search object consistently with those of the model. E.g., it might be helpful to check whether all normals point away from the center of gravity of the object and to flip those normals that do not. Of course, this correction of the normals must be performed for both the model and the search object. Another possibility is to create two models with the generic parameter *'model_invert_normals'* set to *'true'* and *'false'*, respectively. Both models can then be used for matching and the match with the higher score should be selected.

Step 3: Optionally define reference points with
`add_deformable_surface_model_reference_point`

It is possible to define a set of reference points with
`add_deformable_surface_model_reference_point`.

```
add_deformable_surface_model_reference_point (DeformableSurfaceModel, \
                                              ReferencePointX, \
                                              ReferencePointY, \
                                              ReferencePointZ, \
                                              ReferencePointIndex)
```

Reference points are 3D points that can be placed at any position, i.e., it is not necessary that they lie on the object's surface. After a — possibly deformed — object has been found with `find_deformable_surface_model`, the respective 3D position of the reference points in the deformed scene can be determined with `get_deformable_surface_matching_result`.

```
find_deformable_surface_model (DeformableSurfaceModel, \
                               ObjectModel3DSearchSceneForDef, 0.03, 0, [], \
                               [], Score, DeformableSurfaceMatchingResult)
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'reference_point_x', 'all', \
                                        ReferencePointXDeformed)
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'reference_point_y', 'all', \
                                        ReferencePointYDeformed)
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'reference_point_z', 'all', \
                                        ReferencePointZDeformed)
```

Reference points are, e.g., helpful to determine the grasping points for the deformed 3D object.

Step 4: Optionally extend the range of deformations that the deformable surface model is able to cope with

The supported range of deformations can be extended if information about the expected deformations is added to the deformable surface model.

First, provide 3D scenes that contain deformed instances of the 3D object, then, detect those instances with find_deformable_surface_model, and finally, add the found deformed objects to the deformable surface model with add_deformable_surface_model_sample to extend the range of deformations that the deformable surface model is able to cope with.

To make the matching in the 3D scenes used for the model extension faster and more robust, in the example program, the background is eliminated from the reconstructed 3D scene by eliminating all points that are close to the (known) background plane.

```
gen_plane_object_model_3d (PlanePose, [], [], ObjectModel3DPlane)
distance_object_model_3d (ObjectModel3D, ObjectModel3DPlane, [], 0, [], [])
get_object_model_3d_params (ObjectModel3D, '&distance', Distances)
select_points_object_model_3d (ObjectModel3D, '&distance', \
                               MinDistanceToPlane, max(Distances), \
                               ObjectModel3DThresholded)
```

In the resulting 3D scene, the deformed object is detected with find_deformable_surface_model and the found deformed object is added to the deformable surface model with add_deformable_surface_model_sample:

```
find_deformable_surface_model (DeformableSurfaceModel, \
                               ObjectModel3DSearchSceneForDef, 0.03, 0, [], \
                               [], Score, DeformableSurfaceMatchingResult)
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'deformed_sampled_model', 0, \
                                        ObjectModel3DDeformedSampled)
add_deformable_surface_model_sample (DeformableSurfaceModel, \
                                     ObjectModel3DDeformedSampled)
```

Step 5: Provide a 3D scene that represents the search data

The 3D scene in which the object of interest is searched for must be provided as a 3D object model (see figure 4.12c), similar to the 3D object model that was used for the creation of the deformable surface model or for the extension of the supported deformation range. Typically, the 3D object models used for the search are acquired by 3D sensors or reconstructed from stereo image data.

Step 6: Use the deformable surface model to search for the object in the search data

With the deformable surface model that was created with create_deformable_surface_model or read from file with read_deformable_surface_model, the deformed object can be searched for in the search data using the operator find_deformable_surface_model.

```
find_deformable_surface_model (DeformableSurfaceModel, \
                               ObjectModel3DSearchSceneForDef, 0.03, 0, [], \
                               [], Score, DeformableSurfaceMatchingResult)
```

Several parameters can be set to control the search process. For detailed information, we recommend to read the description of the operator find_deformable_surface_modelin the Reference Manual. If a model instance was found, the operator returns a score that describes the quality of the match and a handle that contains the results, like the deformed position of the reference points.

After the matching, the result can be visualized. For this, the deformed reference model can be determined with get_deformable_surface_matching_result:

```
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'deformed_model', 0, \
                                        ObjectModel3DDeformed)
```

The position of the reference points — transformed to the deformed 3D object that was found — can be determined with get_deformable_surface_matching_result. It can, e.g., be used to grasp the deformed object.

```
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'reference_point_x', 'all', \
                                        ReferencePointXDeformed)
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'reference_point_y', 'all', \
                                        ReferencePointYDeformed)
get_deformable_surface_matching_result (DeformableSurfaceMatchingResult, \
                                        'reference_point_z', 'all', \
                                        ReferencePointZDeformed)
```

In the example, the deformed 3D object model is visualized together with the reference points on top of the search scene (see figure 4.12d on page 108).

Note that, especially, if the object does not have a distinct 3D shape, e.g., because most of the points have a similar height, it might be necessary to manually eliminate the background and other objects from the 3D scene to ensure a proper matching result. Otherwise, the object might be found in the background because a relatively flat object fits perfectly to a relatively flat background, especially if deformations of the object are allowed.

4.5 Pose Estimation Using 3D Primitives Fitting

3D primitives fitting determines amongst others the positions or 3D poses of simple 3D shapes, so-called "3D primitives", that are fitted into segmented parts of a 3D scene. The 3D scene is a set of 3D points that is available as 3D object model. It can be obtained, e.g., by stereo vision (chapter 5 on page 117) or sheet of light (section 6.1 on page 147). The available types of 3D primitives comprise a sphere, a cylinder, and a plane. When fitting 3D primitives into the segmented 3D data, the results comprise a radius and position for a sphere, a radius and 3D pose for a cylinder, and a 3D pose for a plane.

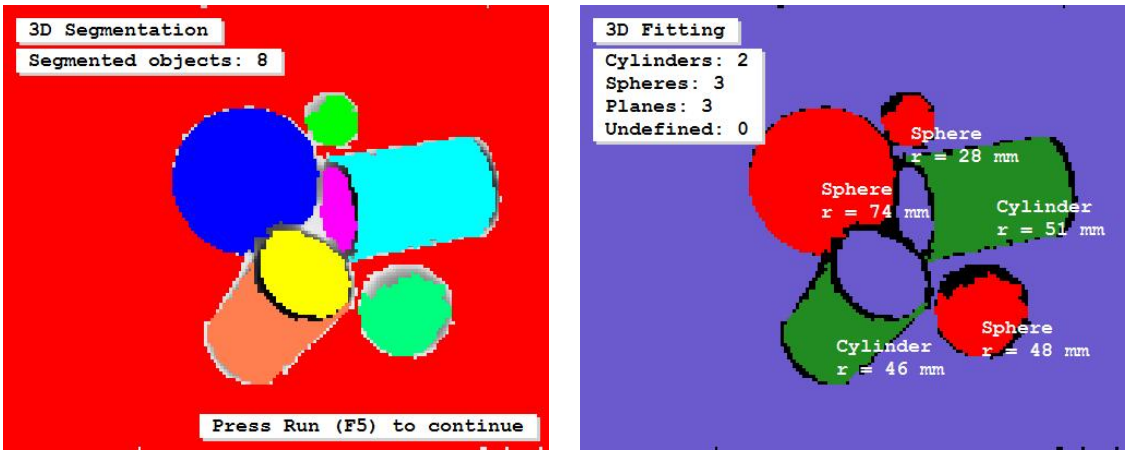


Figure 4.13: 3D primitives fitting: (left) segmentation of 3D object model, (right) result of the fitting.

The fitting typically consists of two steps. The first step is the **segmentation** of the 3D scene into sub-sets of neighbored 3D points that may correspond to selected types of 3D primitives (see figure 4.13, left). The segmentation can be applied by different means, depending on the input data. In any case, the resulting sub-sets of 3D points must be available as 3D object models that contain, at least implicitly, the coordinates of the 3D points and their meshing. The second step is the actual **fitting** (see figure 4.13, right). There, the operator `fit_primitives_object_model_3d` tries to find the best fitting 3D primitive for an individual 3D object model. The result is another 3D object model from which the parameters of the successfully fitted primitive can be queried.

An example for a 3D primitives fitting that uses a simple 2D segmentation to derive the 3D object model of a single cylinder from a 3D scene that is provided by X, Y, and Z images is the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Tools\3D-Segmentation\fit_primitives_object_model_3d.hdev`.

The individual channels of the image are accessed with `access_channel` (see figure 4.14). Then, a threshold is applied to the Z image to separate the cylinder from the background (see figure 4.15). The corresponding

Figure 4.14: Height data used for 3D primitives fitting: (from left to right) X, Y, and Z image.

ROI is created with reduce_domain. From the reduced image channels, a 3D object model is created using xyz_to_object_model_3d.

```
read_image (XYZ, '3d_machine_vision/segmentation/3d_primitives_xyz_02.tif')
access_channel (XYZ, X, 1)
access_channel (XYZ, Y, 2)
access_channel (XYZ, Z, 3)
threshold (Z, Region, 0.0, 0.83)
reduce_domain (X, Region, XTmp)
xyz_to_object_model_3d (XTmp, Y, Z, ObjectModel3DID)
```

A 3D primitive of the 'primitive_type' 'cylinder' is fitted into the 3D object model using the operator fit_primitives_object_model_3d.

```
ParFitting := ['primitive_type', 'fitting_algorithm', 'output_xyz_mapping']
ValFitting := ['cylinder', 'least_squares_huber', 'true']
fit_primitives_object_model_3d (ObjectModel3DID, ParFitting, ValFitting, \
                                ObjectModel3DOutID)
```

Figure 4.15: 3D primitives fitting: (left) simple threshold for 2D segmentation, (right) result returned by the fitting.

The result of the fitting is a handle for a 3D object model from which information like the primitive's parameters can be queried with get_object_model_3d_params.

Often, a simple 2D segmentation is not suitable, e.g., because the 3D data is not derived from Y, X, and Z images or because the 3D scene consists of several 3D structures that touch or overlap. Then, a 3D segmentation using the operator segment_object_model_3d has to be applied as is shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Object-Model\Segmentation\segment_object_model_3d.hdev. There, the individual objects can not be separated by a simple 2D segmentation, because they overlap (see figure 4.16). The 3D data is accessed again from X, Y, and Z images, but this time, the 3D object model is created without a preceding 2D segmentation.

```
read_image (XYZ, '3d_machine_vision/segmentation/3d_primitives_xyz_01.tif')
access_channel (XYZ, X, 1)
access_channel (XYZ, Y, 2)
access_channel (XYZ, Z, 3)
xyz_to_object_model_3d (X, Y, Z, ObjectModel3DID)
```

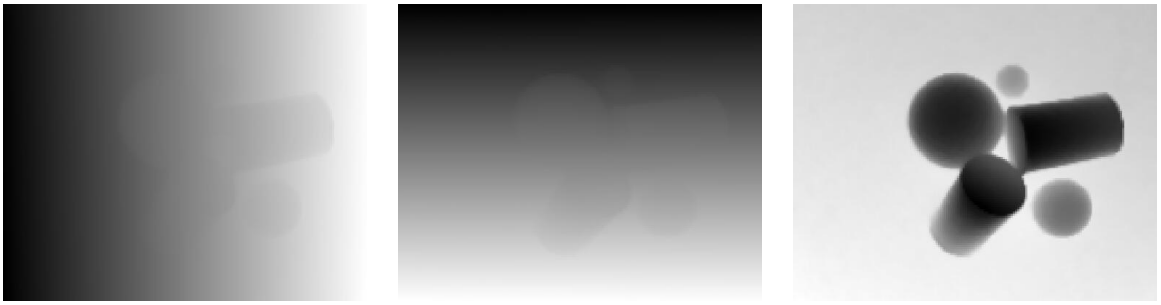


Figure 4.16: Height data used for 3D primitives fitting: (from left to right) X, Y, and Z image.

Instead, the derived 3D object model is prepared for a 3D segmentation by calling the operator prepare_object_model_3d. Note that, if you do not explicitly prepare the 3D object model, the operator is called internally during the segmentation with segment_object_model_3d, which may slow down the application if the same 3D object model is used several times.

```
prepare_object_model_3d (ObjectModel3DID, 'segmentation', 'false', \
                         'max_area_holes', 100)
```

The operator segment_object_model_3d segments the 3D object model into different sub-sets of 3D points that have similar characteristics like the same orientation of the normals or the same curvature of the underlying surface. By default, the operator not only segments the 3D scene but already tries to fit primitives of the selected types into the segments (see figure 4.13 on page 111). Thus, no separate calls to fit_primitives_object_model_3d are needed. Here, 'primitive_type' is set to 'all'. That is, for each sub-set of 3D points the best fitting type of primitive is returned (see figure 4.13 on page 111).

```
ParSegmentation := ['max_orientation_diff', 'max_curvature_diff', \
                    'output_xyz_mapping', 'min_area']
ValSegmentation := [0.13, 0.11, 'true', 150]
ParFitting := ['primitive_type', 'fitting_algorithm']
ValFitting := ['all', 'least_squares_huber']
segment_object_model_3d (ObjectModel3DID, [ParSegmentation,ParFitting], \
                         [ValSegmentation,ValFitting], ObjectModel3DOutID)
```

The result of the combined segmentation and fitting is a tuple of handles for the 3D object models that represent the different sub-sets of 3D points. From each of these 3D object models, information like the success of the fitting, the primitives' types, and the primitives' parameters can be queried with get_object_model_3d_params. Querying the primitive's parameters, a tuple is returned. The size and content of this tuple depends on the type of the primitive. In particular, for a cylinder the tuple contains seven values (three for the position, three for the orientation, and one for the radius), for a sphere it contains four values (three for the position and one for the radius), and for a plane it contains again four values (three for the unit normal vector and one for the orthogonal distance of the plane from the origin of the coordinate system). How to access a single parameter from such a tuple is shown exemplarily for the radius of a cylinder.

```
for Index := 0 to |ObjectModel3DOutID| - 1 by 1
    get_object_model_3d_params (ObjectModel3DOutID[Index], \
                                'has_primitive_data', HasPrimitiveData)
    if (HasPrimitiveData == 'true')
        get_object_model_3d_params (ObjectModel3DOutID[Index], \
                                    'primitive_parameter', \
                                    PrimitiveParameter)
        get_object_model_3d_params (ObjectModel3DOutID[Index], \
                                    'primitive_type', PrimitiveType)
        if (PrimitiveType == 'cylinder')
                RadiusCylinder := PrimitiveParameter[6]
        endif
    endif
endfor
```

4.6 Pose Estimation Using Calibrated Perspective Deformable Matching

The perspective deformable matching finds and locates objects that are similar to a template model in an image. This matching approach uses the contours of the object in the images and is independent of the perspective view on the object, i.e., perspective deformations are considered when searching the model in unknown images. The perspective deformable matching can be applied either for a calibrated camera, then the 3D pose of the object is returned, or for an uncalibrated camera, then only the 2D projective transformation matrix (homography) is returned.

The perspective deformable matching is suitable for all planar objects or planar object parts that are clearly distinguishable by their contours. Compared to the shape-based 3D matching (see section 4.2 on page 95 or the Solution Guide I, chapter 11 on page 101), there is no need to pregenerate different views of an object. Thus, it is significantly faster. Hence, if you search for planar perspectively deformed objects, we recommend the perspective deformable matching.

The Solution Guide II-B, section 3.4 on page 86 shows in detail how to apply the approach. Figure 4.17 shows the poses of engine parts obtained by the HDevelop example `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\locate_engine_parts.hdev`.

4.7 Pose Estimation Using Calibrated Descriptor-Based Matching

Similar to the perspective deformable matching, the descriptor-based matching finds and locates objects that are similar to a template model in an image. Again, the matching can be applied either for a calibrated camera, then the 3D pose of an object is returned, or for an uncalibrated camera, then only the 2D projective transformation matrix (homography) is returned.

The essential difference between the descriptor-based and the perspective deformable matching is that the descriptor-based matching is not based on contours but uses distinctive object points, so-called interest points, to describe the template and to find the model in the image.

Note that the calibrated descriptor-based matching is suitable mainly to determine the 3D pose of planar objects with characteristic texture and distinctive object points. For low-textured objects with rounded edges you should select one of the other pose estimation approaches. Further, the descriptor-based matching is less accurate than the perspective deformable matching. But on the other hand, it is significantly faster if a large search space, e.g., caused by a large scale range, is used.

The Solution Guide II-B, section 3.5 on page 94 shows in detail how to apply the approach. Figure 4.18 shows the 3D pose of a cookie box obtained by the HDevelop example `%HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\locate_cookie_box.hdev`. A comparison between the calibrated descriptor-based matching and pose estimation methods that use manually extracted correspondences (see section 4.1 on page 92) is provided by the HDevelop example `%HALCONEXAMPLES%\hdevelop\Matching\Descriptor-Based\pose_from_point_correspondences.hdev`.

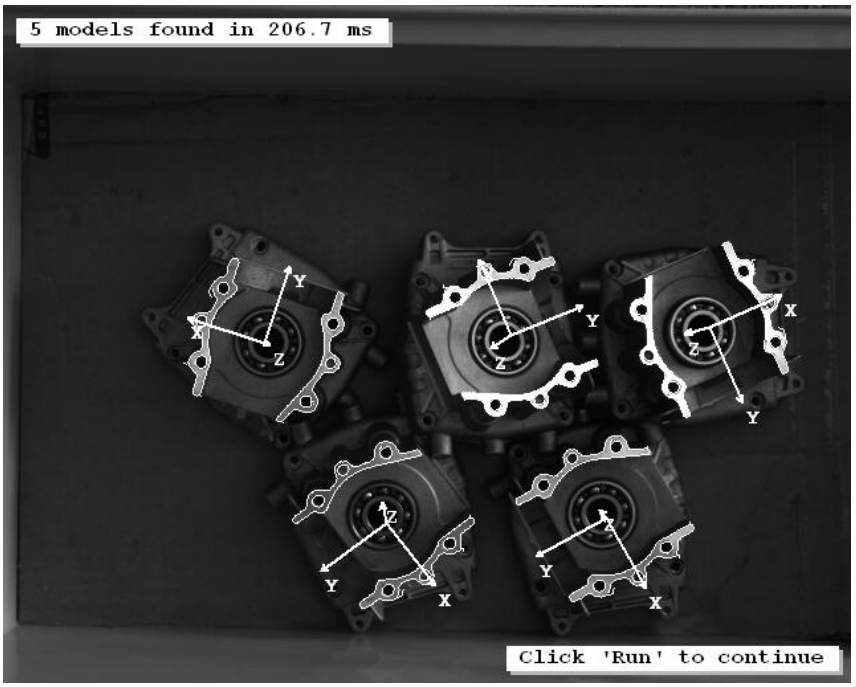


Figure 4.17: 3D poses of engine parts obtained by a calibrated perspective deformable matching.

Figure 4.18: 3D pose of a cookie box label obtained by a calibrated descriptor-based matching.

4.8 Pose Estimation for Circles

HALCON offers an alternative approach to estimate the pose of 3D circles, which can be applied with less effort than the previously described approaches. It is based on the known geometrical behavior of perspectively distorted circles. In particular, 3D circles are represented as ellipses in the image. Using the extracted 2D ellipse of a 3D circle together with the internal camera parameters and the known radius of the circle, the two possible 3D poses of the circle (having the same position but opposite orientations) can be obtained easily using the operator `get_circle_pose`. The HDevelop examples `%HALCONEXAMPLES%\hdevelop\Transformations\Poses\get_circle_pose.hdev` and `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\3d_position_of_circles.hdev` show in detail how to apply the approach.

4.9 Pose Estimation for Rectangles

Additionally to the pose estimation for 3D circles, also the poses of 3D rectangles can be estimated with an approach that can be applied with less effort than the general approach. It is based on the known geometrical behavior of perspectively distorted rectangles. In particular, a contour is segmented into four line segments and their intersections are considered as the corners of a quadrangular contour. Using the extracted 2D quadrangle of the 3D rectangle together with the internal camera parameters and the known size of the rectangle, the four (or eight in case of a square) possible 3D poses of the rectangle can be obtained easily using the operator `get_rectangle_pose`. The HDevelop examples `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\get_rectangle_pose_barcode.hdev` and `%HALCONEXAMPLES%\hdevelop\Applications\Position-Recognition-3D\3d_position_of_rectangle.hdev` show in detail how to apply the approach.

Chapter 5

3D Vision With a Stereo System

With a stereo system, i.e., with two or more cameras, you can derive 3D information of the surface of arbitrarily shaped objects. Possible results are distance images, 3D coordinates, or 3D surfaces.

Typical applications of stereo vision comprise, but are not limited to, completeness checks, inspection of ball grid arrays, etc. Reconstructing surfaces can also serve as a preprocessing step for surface-based 3D matching (see section 4.3 on page 104) or 3D primitives fitting (see section 4.5 on page 111).

Figure 5.1 shows a surface reconstructed from four stereo images. Figure 5.2 shows an image of a binocular stereo camera system, the resulting stereo image pair, and the height map that has been derived from the reconstructed distance image.

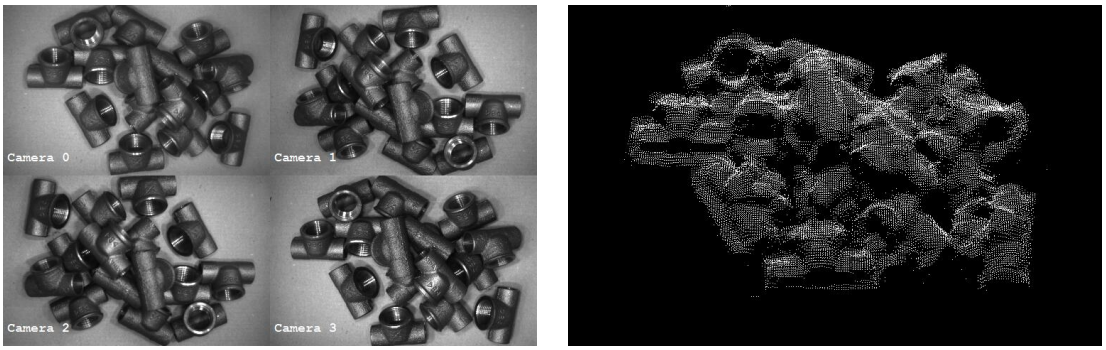


Figure 5.1: Left: images of a 4-camera system; right: reconstructed surface (3D object model).

HALCON actually provides two stereo methods:

- **Binocular stereo** (section 5.3 on page 124) uses exactly two cameras. The result is a disparity image, a distance image, or 3D coordinates. The latter are returned either for selected points or for the complete view.
- **Multi-view stereo** (section 5.4 on page 139) can use more than two cameras. Thus, it is able to image the whole 3D object and not just the surface of a specific view. It returns results in form of 3D surfaces (given as 3D object models, see page 38) or 3D coordinates of selected points.

Before describing the two methods, we first take a look at some general topics. In particular, we

- introduce you to the principle of stereo vision (section 5.1) and
- show how to calibrate a stereo system (section 5.2 on page 122).

5.1 The Principle of Stereo Vision

Assume the simplified configuration of two parallel looking 1D cameras with identical internal parameters as shown in figure 5.3. Furthermore, the basis, i.e., the straight line connecting the two optical centers of the two cameras, is assumed to coincide with the x-axis of the first camera.

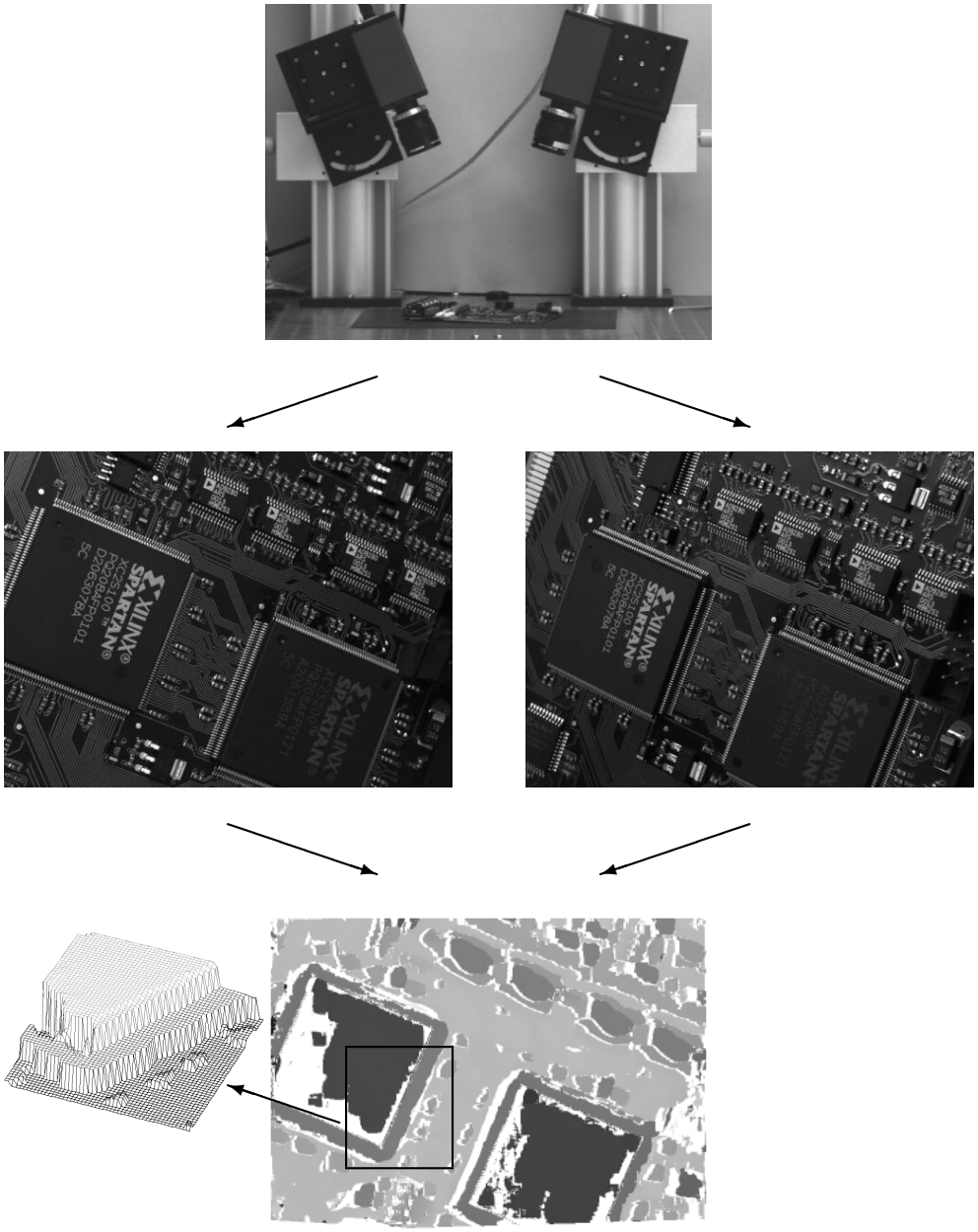


Figure 5.2: Top: stereo camera system; center: stereo image pair; bottom: height map.

Then, the image plane coordinates of the projections of the point $P(x^c, z^c)$ into the two images can be expressed by

$$u_1 = f\frac{x^c}{z^c} \tag{5.1}$$

$$u_2 = f\frac{x^c - b}{z^c} \tag{5.2}$$

where f is the focal length and b the length of the basis.

The pair of image points that results from the projection of one object point into the two images is often referred to as *conjugate points* or *homologous points*.

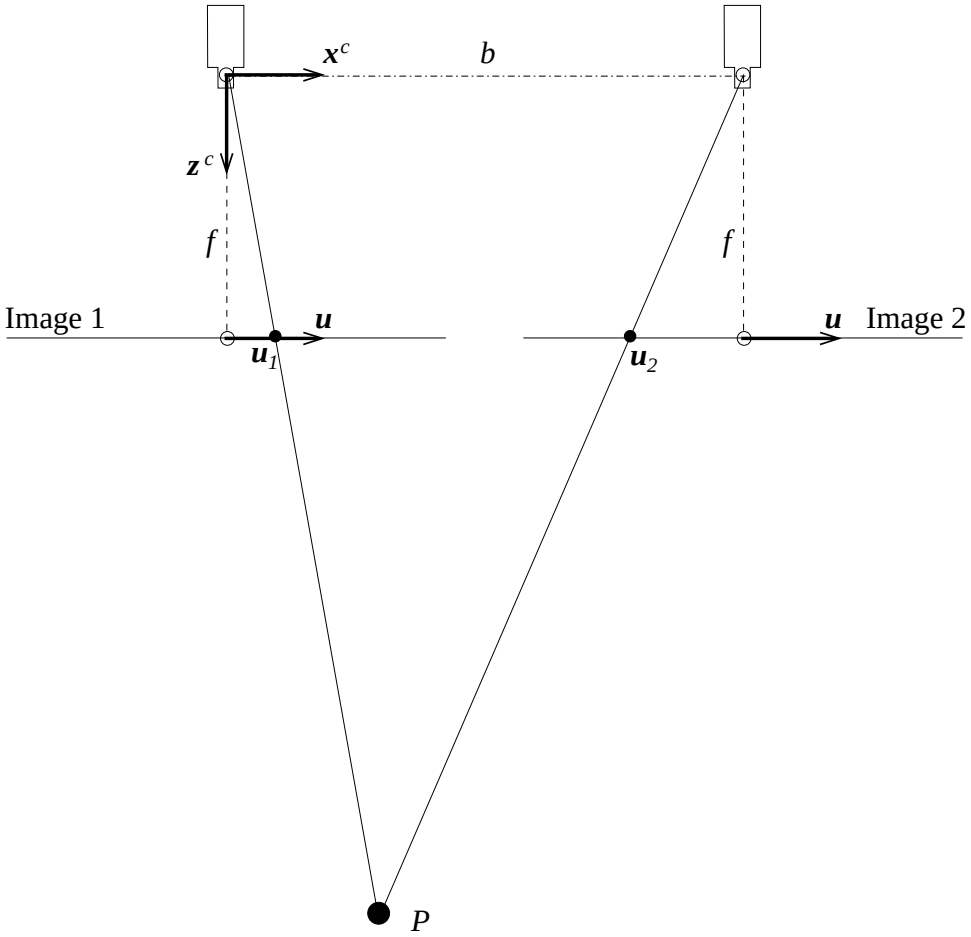


Figure 5.3: Vertical section of a binocular stereo camera system.

The difference between the two image locations of the *conjugate points* is called the *disparity* d:

$$d = (u_2 - u_1) = -\frac{f \cdot b}{z^c} \tag{5.3}$$

Given the camera parameters and the image coordinates of two conjugate points, the z^c coordinate of the corresponding object point P, i.e., its distance from the stereo camera system, can be computed by

$$z^c = -\frac{f \cdot b}{d} \tag{5.4}$$

Note that the internal camera parameters of both cameras and the relative pose of the second camera in relation to the first camera are necessary to determine the distance of P from the stereo camera system.

Thus, the tasks to be solved for stereo vision are

1. to determine the camera parameters and
2. to determine conjugate points.

The first task is achieved by the calibration of the stereo camera system, which is described in section 5.2. This calibration is quite similar to the calibration of a single camera, described in section 3.2 on page 61, in fact, it even uses the same operators.

The second task is the so-called stereo matching process, which in HALCON is just a call of the operator `binocular_disparity` (or `binocular_distance`, respectively) for the correlation-based stereo and a call of

the operator `binocular_disparity_mg` (or `binocular_distance_mg`, respectively) for multigrid stereo. These operators are described in section 5.3.5, together with the operators doing all the necessary calculations to obtain world coordinates from the stereo images.

The multi-view surface reconstruction described in section 5.4.2.1 on page 141 extends the basic stereo vision principle to more than one image pair. There, the matching step is "hidden" in the reconstruction operators, which use the operator `binocular_disparity` internally to compute disparity images of the individual stereo image pairs.

5.1.1 The Setup of a Stereo Camera System

The basic stereo camera system consists of two cameras looking at the same object from different positions (see figure 5.4). Multi-view stereo systems have additional cameras, but the principle stays the same.

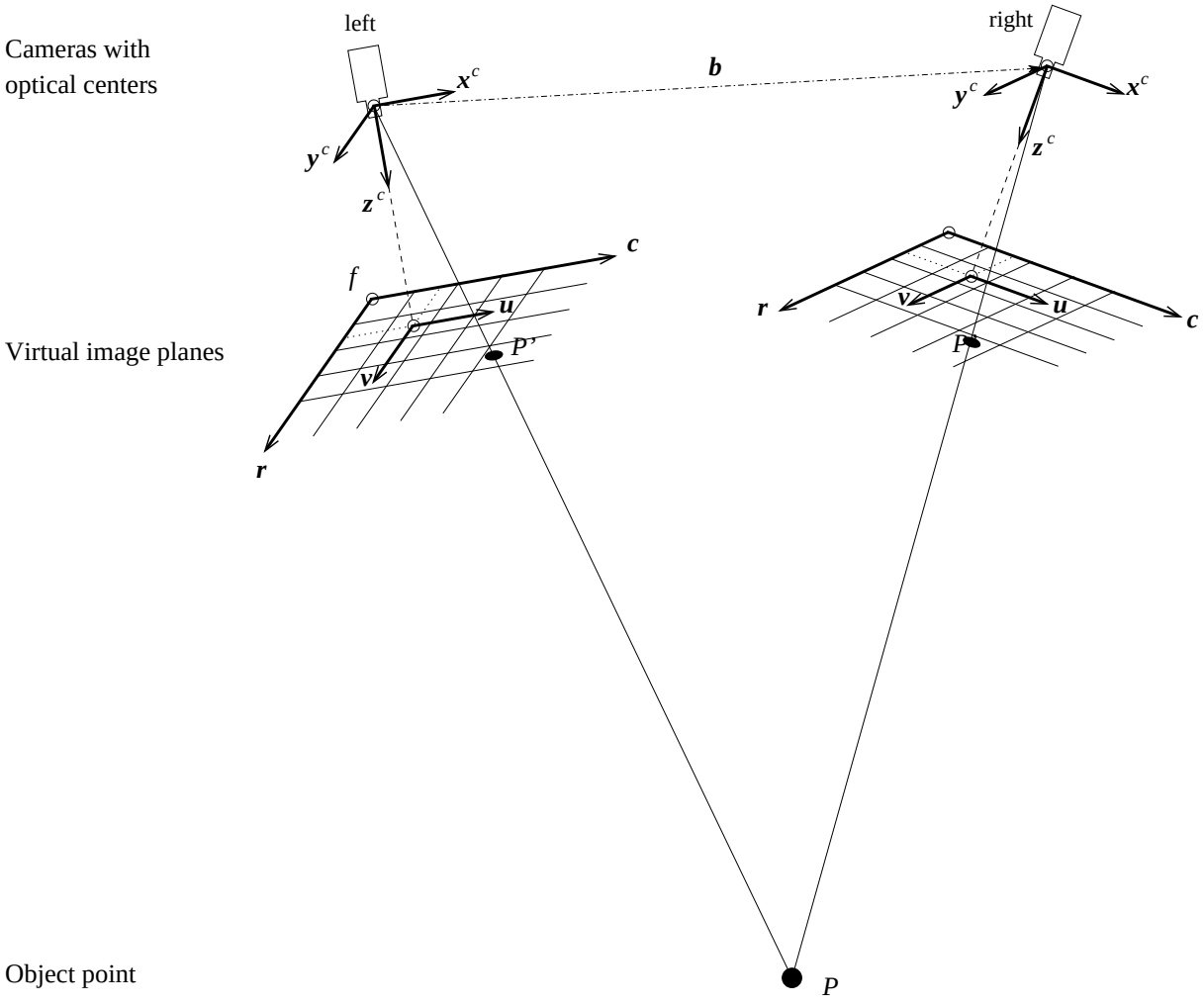


Figure 5.4: Stereo camera system (r and y axes point towards the reader).

It is very important to ensure that neither the internal camera parameters (e.g., the focal length) nor the relative pose between the cameras changes during the calibration process or between the calibration process and the ensuing application of the calibrated stereo camera system, because the calibration remains valid only as long as the cameras preserve their relative pose. Therefore, it is advisable to mount the cameras on a *stable* platform.

5.1.2 Resolution of a Stereo Camera System

The manner in which the cameras are placed influences the accuracy of the results that is achievable with the stereo camera system.

The distance resolution Δz, i.e., the accuracy with which the distance z of the object surface from the stereo camera system can be determined, can be expressed by

$$\Delta z = \frac{z^2}{f \cdot b} \cdot \Delta d \tag{5.5}$$

To achieve a high distance resolution, the setup should be chosen such that the length b of the basis as well as the focal length f are large, and that the stereo camera system is placed as close as possible to the object. In addition, the distance resolution depends directly on the accuracy Δd with which the disparities can be determined. If the calibration has been performed accurately and the corresponding calibration error is in the order of 0.1 pixels, the disparities typically can be determined with an accuracy of $1/5$ up to $1/10$ pixel, which corresponds to approximately 1 μm for a camera with 7.4 μm pixel size. Note that generally the disparities cannot be determined more accurately than the calibration error of the calibration of a stereo camera system.

In figure 5.5, the distance resolutions that are achievable in the ideal case are plotted as a function of the distance for four different configurations of focal lengths and base lines, assuming Δd to be 1 μm.

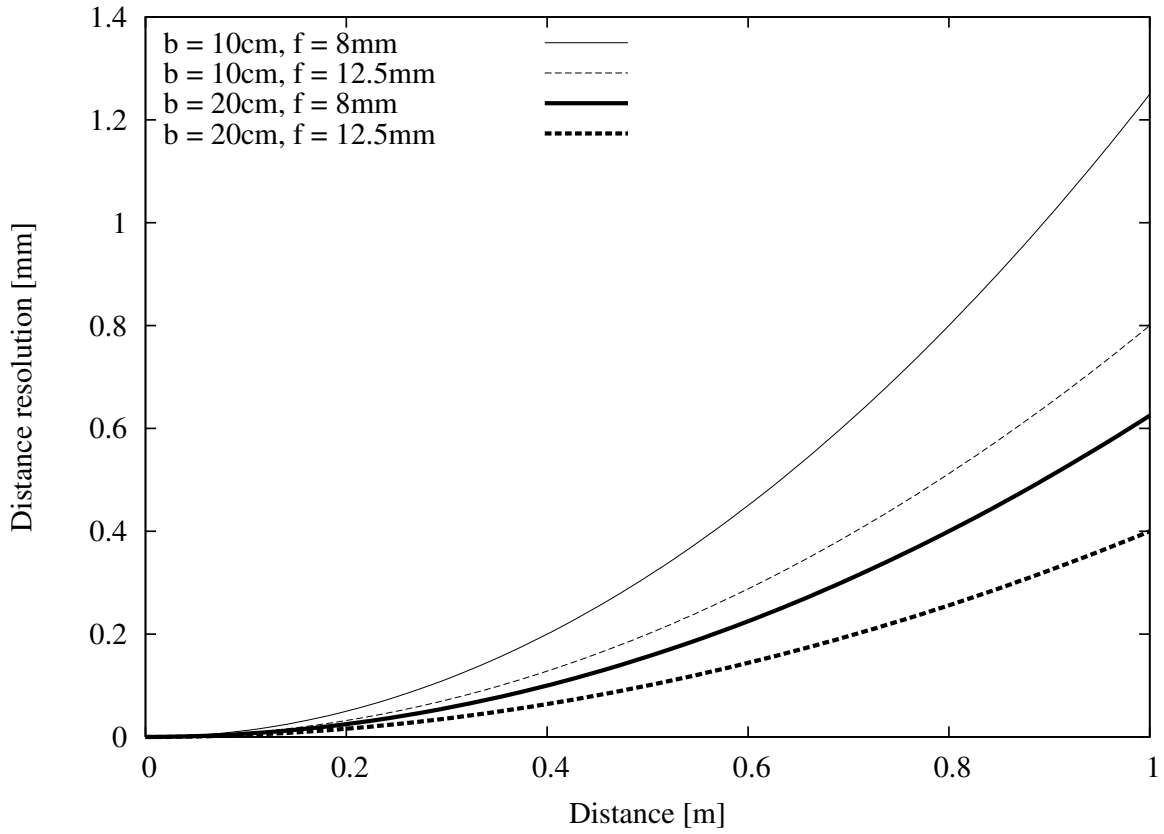


Figure 5.5: Distance resolution plotted over the distance ($\Delta d = 1\,\mu m$).

Note that if the ratio between b and z is very large, problems during the stereo matching process may occur, because the two images of the stereo pair differ too much. The maximum reasonable ratio b/z depends on the surface characteristics of the object. In general, objects with little height differences can be imaged with a higher ratio b/z, whereas objects with larger height differences should be imaged with a smaller ratio b/z.

If this is difficult to ensure in your application when using two cameras, you should consider using a multi-view stereo system, i.e., more than two cameras.

5.1.3 Optimizing Focus with Tilt Lenses

For a successful stereo reconstruction, it is necessary that corresponding points of the object appear sharp in both cameras. In some setups, e.g., when using a large magnification or telecentric lenses, this requirement can be

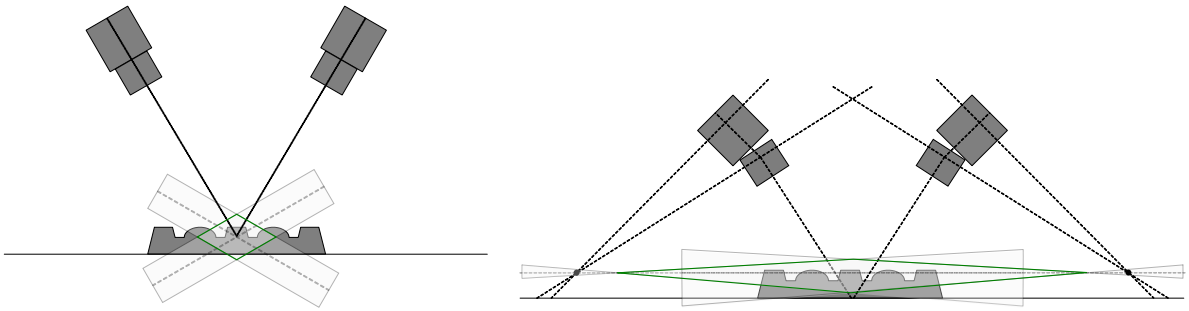


Figure 5.6: If the depth of field is small while the angle between the two cameras is relatively large, only a small part of the object plane is in focus (left). With tilt lenses, the focus planes of the cameras can be aligned with the object plane. This is done, when the condition of the Scheimpflug principle is met. After that, the whole object surface is in focus (right).

difficult to fulfill. This is because the depth of field is small and the angle between the two cameras is relatively large. Therefore, the overlapping area, where the object is in focus from both cameras, may not cover the whole object (see figure 5.6).

In such cases, tilt lenses can be used to adjust the focus planes of each camera to better fit the object plane. Section 2.2.3 describes the effect of tilt lenses in more detail.

5.2 Calibrating the Stereo Camera System

As mentioned above, the calibration of the stereo camera system is very similar to the calibration of a single camera (section 3.2 on page 61). The major difference is that it is not sufficient to view the calibration plate from a single camera, but it must be visible in at least some of the overlapping parts of the images that are taken by each pair of neighbored cameras (see section 5.2.2 for details). There, it must be completely visible for calibration plates with rectangularly arranged marks whereas for calibration plates with hexagonally arranged marks at least one finder pattern must be visible.

In this section only a brief description of the calibration process is given. More details can be found in section 3.2 on page 61. Here, the stereo-specific parts of the calibration process are described in depth. In particular, it is shown

- how to create and configure the calibration data model for multiple cameras (section 5.2.1),
- how to acquire suitable calibration images (section 5.2.2),
- how to add the observation data of multiple cameras to the calibration data model (section 5.2.3), and
- how to perform the calibration (section 5.2.4 on page 124).

The code fragments used for the following descriptions belong to the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Calibration\Multi-View\calibrate_cameras_multiple_camera_setup.hdev`.

5.2.1 Creating and Configuring the Calibration Data Model

As described in section 3.2.1 on page 62, the first step of preparing for a calibration is to create the calibration data model with the operator `create_calib_data`. In the example, four cameras are used with one calibration object.

```
NumCameras := 4
NumCalibObjects := 1
create_calib_data ('calibration_object', NumCameras, NumCalibObjects, \
                   CalibDataID)
```

Then, the initial camera parameters are set with the operator set_calib_data_cam_param (see also section 3.2.2 on page 62). The parameter CameraIdx is set to 'all', so that the values are set for all cameras.

```
gen_cam_par_area_scan_polynomial (0.0085, 0.0, 0.0, 0.0, 0.0, 0.0, 6e-6, \
                                  6e-6, Width * .5, Height * .5, Width, \
                                  Height, StartCamPar)
set_calib_data_cam_param (CalibDataID, 'all', [], StartCamPar)
```

Finally, the calibration object is described with the operator set_calib_data_calib_object (see section 3.2.3 on page 67).

```
CaltabDescr := 'caltab_100mm.descr'
set_calib_data_calib_object (CalibDataID, 0, CaltabDescr)
```

5.2.2 Acquiring Calibration Images

For the calibration of the stereo camera system, each camera acquires multiple images of one or more calibration objects in different poses. Note that it is not necessary that the calibration object is always visible in all poses for each camera. The only requirement is that the cameras can be "connected" in a chain by the calibration object poses, e.g., that camera 0 and 1 observe the pose 0 of the calibration object, camera 1 and 2 observe pose 3, and camera 2 and 3 observe pose 8 (see figure 5.7).

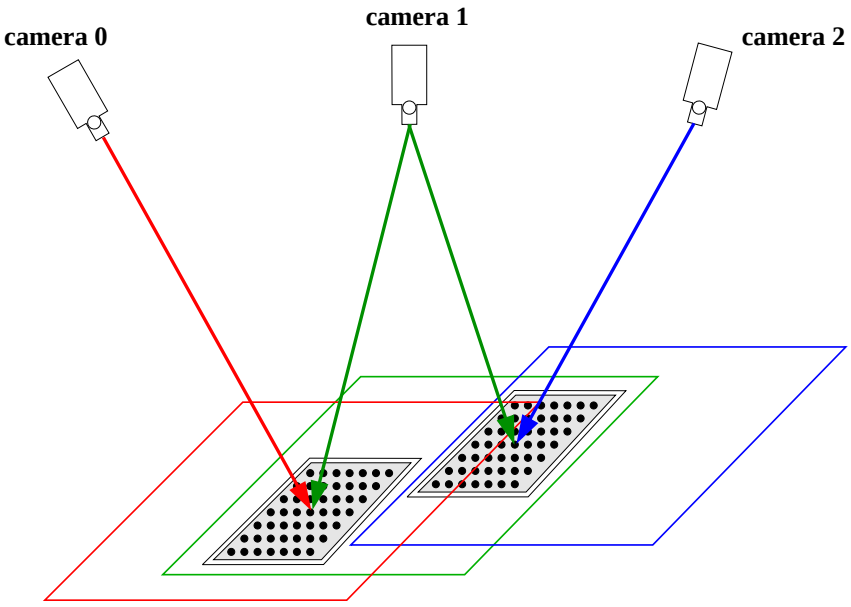


Figure 5.7: Cameras are connected in a chain.

Note that when taking the images for the calibration, the recommendations for taking the calibration images for the single camera calibration (see the section "How to take a set of suitable images?" in the chapter reference "Calibration") apply accordingly. More information on the calibration of multi-view camera setups, especially with mixed (telecentric and perspective) camera setups, can be found in the chapter "Calibration ▷ Multi-View".

Note that you must not change the camera setup between the acquisition of the calibration images and the acquisition of the stereo images of the object to be investigated. How to obtain suitable stereo image pairs for the object to investigate is shown in section 5.3.3 on page 126.

5.2.3 Observing the Calibration Object

As in the case of the single camera calibration, the main input data for the camera calibration are the observed points of the calibration objects in the camera images (see section 3.2.4 on page 71). Generally, you extract the observations and then add them to the calibration data model with the operator set_calib_data_observ_points.

If you use a standard HALCON calibration plate as calibration object, you can extract and store the observations in a single step using `find_calib_object`. In the example, this operator is called within a double loop over all poses and all cameras. The code raises a warning if the calibration plate could not be found in an image.

```
for PoseIndex := 0 to NumPoses - 1 by 1
    for CameraIndex := 0 to NumCameras - 1 by 1
        read_image (Image, FileName)
        Message := ['Camera ' + CameraIndex,'Pose # ' + PoseIndex]
        try
            find_calib_object (Image, CalibDataID, CameraIndex, 0, \
                               PoseIndex, [], [])
        catch (Exception)
            if (Exception[0] == 8402)
                Message := [Message,'No calibration tab found!']
            elseif (Exception[0] == 8404)
                Message := [Message,'Marks were not identified!']
            else
                Message := [Message,'Unknown Exception!.']
            endif
            Message := [Message,'This image will be ignored.']
        endtry
        disp_message (WindowHandles[CameraIndex], Message, 'window', 12, 12, \
                      Color, 'true')
    endfor
endfor
```

5.2.4 Calibrating the Cameras

The actual calibration of the stereo camera system is carried out with the operator `calibrate_cameras` (see section 3.2.6 on page 72).

```
calibrate_cameras (CalibDataID, Error)
```

The calibration stores its results in the calibration data model. How to access them is described in separate sections for binocular stereo (section 5.3.2 on page 126) and multi-view stereo (section 5.4.1.1 on page 140) because the methods use the results in different forms.

5.3 Binocular Stereo Vision

With the operators for binocular stereo vision, you can compute disparity and distance images and 3D coordinates using two cameras. In fact, HALCON provides three methods for binocular stereo: correlation-based stereo, multigrid stereo, and multi-scanline stereo (see section 5.3.1 for the differences).

The following sections show how to

- access the results of the calibration (section 5.3.2 on page 126),
- rectify the stereo images (section 5.3.4 on page 127), and
- reconstruct 3D information (section 5.3.5 on page 130).

If not stated otherwise, the example programs used in this section are

- `stereo_calibration.hdev`,
- `height_above_reference_plane_from_stereo.hdev`, and
- `3d_information_for_selected_points.hdev`.

They can be found in the directory `solution_guide\3d_vision`.

As an alternative to fully calibrated stereo, HALCON offers the so-called uncalibrated stereo vision. Here, the relation between the cameras is determined from the scene itself, i.e., without needing a special calibration object. Please refer to section 5.3.6 on page 138 for more information about this method.

5.3.1 Comparison of the Stereo Matching Approaches Correlation-Based, Multigrid, and Multi-Scanline Stereo

In HALCON, three approaches for stereo matching are available: the traditionally used correlation-based stereo matching, the multigrid stereo matching, and the multi-scanline stereo matching.

The **correlation-based stereo** matching uses correlation techniques to find corresponding points and thus to determine the disparities or distances for the observed image points. The disparities or distances are calculated with the operators `binocular_disparity` or `binocular_distance`, respectively. The correlation-based stereo is characterized by the following advantages and disadvantages:

Most important advantages of correlation-based stereo:

- fast,
- can be automatically parallelized on multi-core or multi-processor hardware, and
- is invariant against gray-value changes.

Most important disadvantage of correlation-based stereo:

- works good only for significantly textured areas. Areas without enough texture cannot be reconstructed.

The **multigrid stereo matching** uses a variational approach based on multigrid methods. This approach returns disparity and distance values also for image parts that contain no texture (as long as these parts are surrounded by significant structures between which an interpolation of values is possible). The disparities or distances are calculated with the operators `binocular_disparity_mg` or `binocular_distance_mg`, respectively. The multigrid stereo is characterized by the following advantages and disadvantages:

Most important advantages of multigrid stereo:

- interpolates 3D information for areas without texture based on the surrounding areas,
- in particular for edges, the accuracy in general is higher than for correlation-based stereo, and
- the resolution is higher than for correlation-based stereo, i.e., smaller objects can be reconstructed.

Most important disadvantages of multigrid stereo:

- only partially invariant against gray value changes,
- edges are somewhat blurred, and
- it cannot be parallelized automatically.

The **multi-scanline stereo matching** uses a multi-scanline optimization. Similar to the multigrid approach, it returns disparity and distance values also for image parts that contain only little texture but tries to preserve discontinuities, which are blurred by the multigrid approach. The disparities or distances are calculated with the operators `binocular_disparity_ms` or `binocular_distance_ms`, respectively. The multi-scanline stereo is characterized by the following advantages and disadvantages:

Most important advantages of multi-scanline stereo:

- determines 3D information for areas with little texture and
- preserves discontinuities.

Most important disadvantages of multi-scanline stereo:

- runtime increases significantly with image size and disparity search range and
- high memory consumption.

5.3.2 Accessing the Calibration Results

As described in section 3.2.7 on page 72, you access the results of the calibration with the operator get_calib_data. In addition to the internal camera parameters, we now use the operator also to get the relative pose between the two cameras.

```
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParamL)
get_calib_data (CalibDataID, 'camera', 1, 'params', CamParamR)
get_calib_data (CalibDataID, 'camera', 1, 'pose', cLPcR)
```

If you want to perform the calibration in an offline step, you can save the camera setup model with write_camera_setup_model.

```
write_camera_setup_model (CameraSetupModelID, 'stereo_camera_setup.csm')
```

5.3.3 Acquiring Stereo Images

The following rules help you to acquire suitable stereo image pairs. Section 5.3.3.1 and section 5.3.3.2 provide you with background information to understand the rules.

- Do not change the camera setup between the acquisition of the calibration images and the acquisition of the stereo images of the object to be investigated. How to obtain suitable images for the calibration of a stereo camera system is shown in section 5.2.2 on page 123.
- Ensure a proper illumination of the object, e.g., avoid reflections.
- If the object shows no texture, consider to project texture onto it or use multigrid stereo (see section 5.3.1 on page 125).
- Place the object such that repetitive patterns are not aligned with the rows of the rectified images.

5.3.3.1 Image Texture

The 3D coordinates of each object point are derived by intersecting the lines of sight of the respective conjugate image points. The conjugate points are determined by an automatic matching process. This matching process has some properties that should be accounted for during the image acquisition.

For each point of the first image, the conjugate point in the second image must be determined. This point matching relies on the availability of texture. The conjugate points cannot be determined correctly in areas without sufficient texture (figure 5.8).

This applies in particular for the correlation-based (binocular) stereo (which is currently the base for multi-view stereo). The multigrid (binocular) stereo, however, can interpolate values in areas without texture. But note that even then, a certain amount of texture must be available to enable the interpolation. In section 5.3.1 on page 125 the differences between both stereo matching approaches are described in more detail.

5.3.3.2 Repetitive Patterns

If the images contain repetitive patterns, the matching process may be confused, since in this case many points look alike. In order to make the matching process fast and reliable, the stereo images are rectified such that pairs of conjugate points always have identical row coordinates in the rectified images, i.e., that the search space in the second rectified image is reduced to a line. With this, repetitive patterns can disturb the matching process only if they are parallel to the rows of the rectified images (figure 5.9).

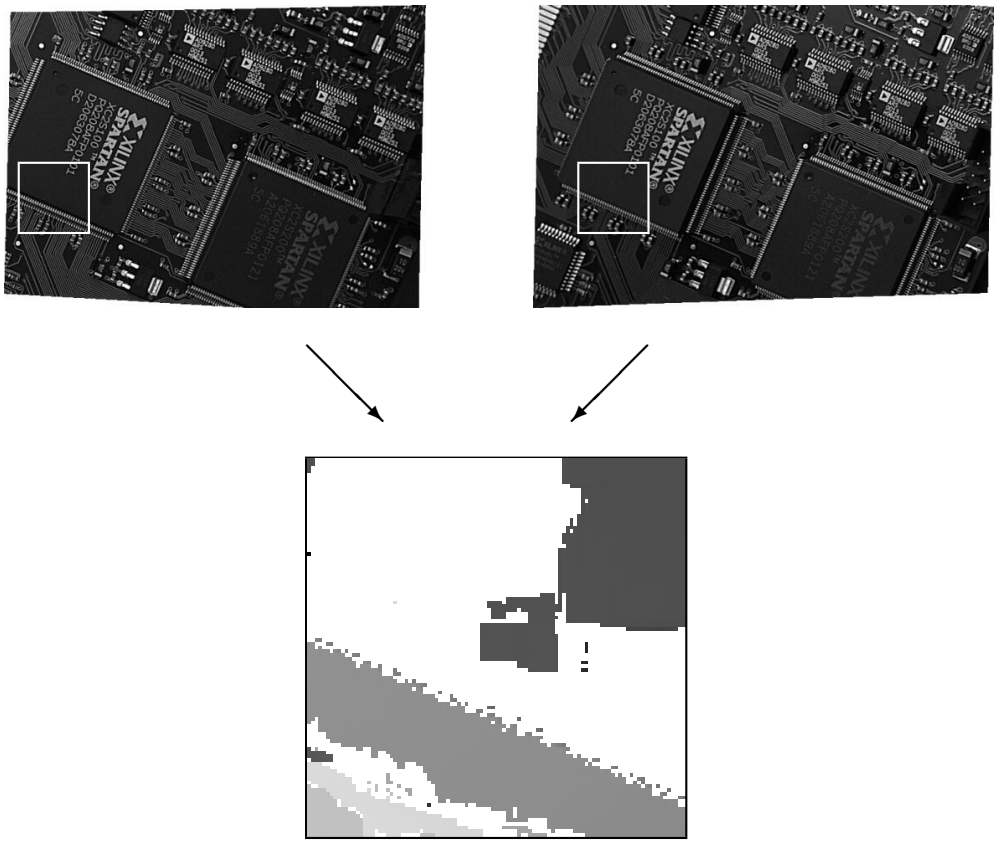


Figure 5.8: Rectified stereo images and matching result of the correlation-based stereo in a poorly textured area (regions where the matching process failed are displayed white).

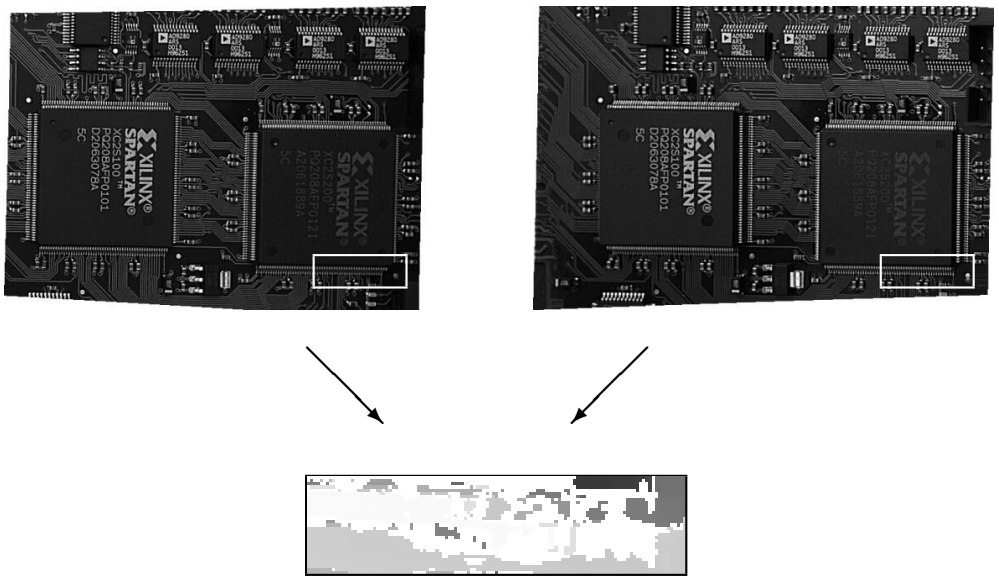


Figure 5.9: Rectified stereo images with repetitive patterns aligned to the image rows and cutout of the matching result (regions where the matching process failed are displayed white).

5.3.4 Rectifying the Stereo Images

With the internal camera parameters and the relative pose, the stereo images can be *rectified*, so that conjugate points lie on the same row in both rectified images.

Note that it is assumed that the parameters of the *first* image of a pair stem from the *left* image and the parameters of the *second* image stem from the *right* image, whereas the notations 'left' and 'right' refer to the line of sight of

the two cameras (see figure 5.4 on page 120). If the images are used in the reverse order, they will appear upside down after the rectification.

In figure 5.10 the original images of a stereo pair are shown, where the two cameras are rotated heavily with respect to each other. The corresponding rectified images are displayed in figure 5.11.

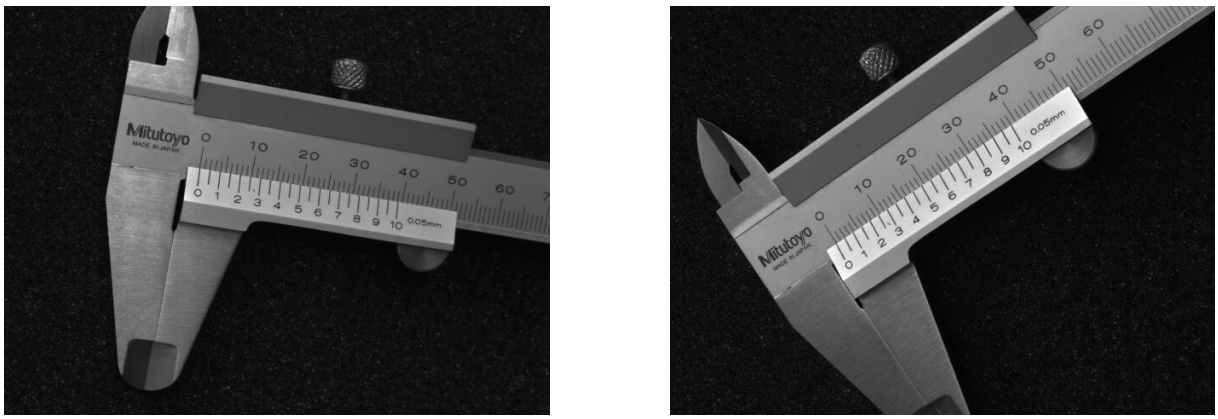


Figure 5.10: Original stereo images.

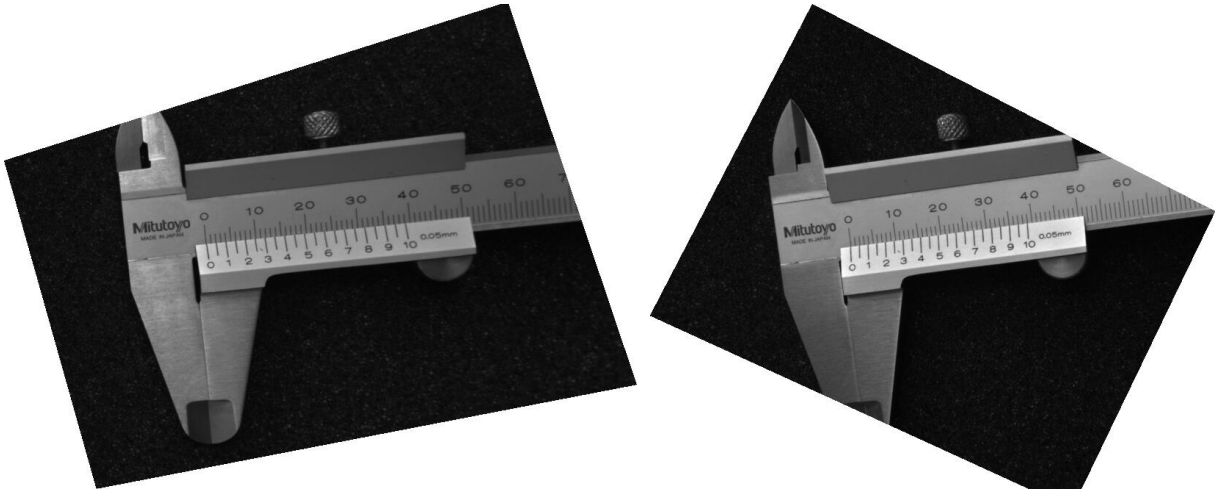


Figure 5.11: Rectified stereo images.

The rectification itself is carried out using the operators `gen_binocular_rectification_map` and `map_image`. The operator `gen_binocular_rectification_map` requires the internal camera parameters of both cameras and the relative pose of the second camera in relation to the first camera.

```
gen_binocular_rectification_map (MapL, MapR, CamParamL, CamParamR, cLPcR, 1, \
                                 'viewing_direction', 'bilinear', \
                                 RectCamParL, RectCamParR, CamPoseRectL, \
                                 CamPoseRectR, RectLPosRectR)
```

The parameter `SubSampling` can be used to change the size and resolution of the rectified images with respect to the original images. A value of *1* indicates that the rectified images will have the same size as the original images. Larger values lead to smaller images with a resolution reduced by the given factor, smaller values lead to larger images.

Reducing the image size has the effect that the following stereo matching process runs faster, but also that less details are visible in the result. In general, it is proposed not to use values below *0.5* or above *2*. Otherwise, smoothing or aliasing effects occur, which may disturb the matching process.

The rectification process can be described as projecting the original images onto a common rectified image plane. The method to define this plane can be selected by the parameter `Method`. So far, two methods are implemented.

For the method *'geometric'* the orientation of the common rectified image plane is defined by the cross product of the base line and the line of intersection of the two original image planes. The default method, *'viewing_direction'*, uses the base line as x-axis of the common image plane. The mean of the viewing directions (z-axes) of the two cameras is used to span the x-z plane of the rectified system. The resulting rectified z-axis is the orientation of the common image plane and as such located in this plane and orthogonal to the base line.

For perspective cameras, the rectified images can be thought of as being acquired by a virtual stereo camera system, called rectified stereo camera system, as displayed in figure 5.12. The optical centers of the rectified cameras are the same as for the real cameras, but the rectified cameras are rotated such that they are looking parallel and that their x-axes are collinear. In addition, both rectified cameras have the same focal length. Therefore, the two image planes coincide. Note that the principal point of the rectified images, which is the origin of the image plane coordinate system, may lie outside the image.

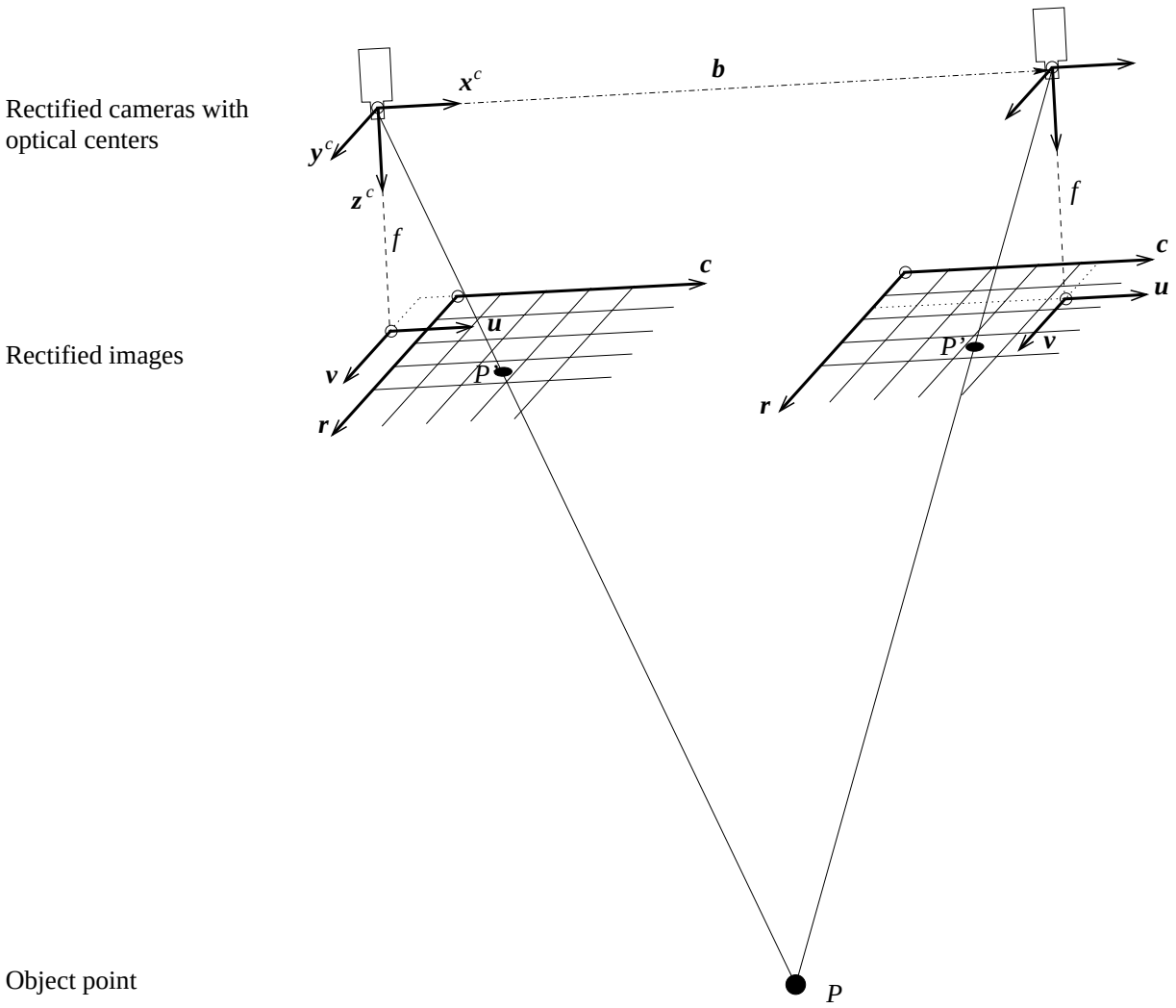


Figure 5.12: Rectified stereo camera system.

The parameter `Interpolation` specifies whether bilinear interpolation (*'bilinear'*) should be applied between the pixels of the input images or whether the gray value of the nearest pixel (*'none'*) should be used. Bilinear interpolation yields smoother rectified images, whereas the use of the nearest neighbor is faster.

The operator returns the rectification maps and the camera parameters of the virtual, rectified cameras.

Finally, the operator `map_image` can be applied to both stereo images using the respective rectification map generated by the operator `gen_binocular_rectification_map`.

```
map_image (ImageL, MapL, ImageRectifiedL)
map_image (ImageR, MapR, ImageRectifiedR)
```

If the calibration was erroneous, the rectification will produce wrong results. This can be checked very easily by comparing the row coordinates of conjugate points selected from the two rectified images. If the row coordinates of conjugate points are different within the two rectified images, they are not correctly rectified. In this case, you should check the calibration process carefully.

An incorrectly rectified image pair may look like the one displayed in figure 5.13.

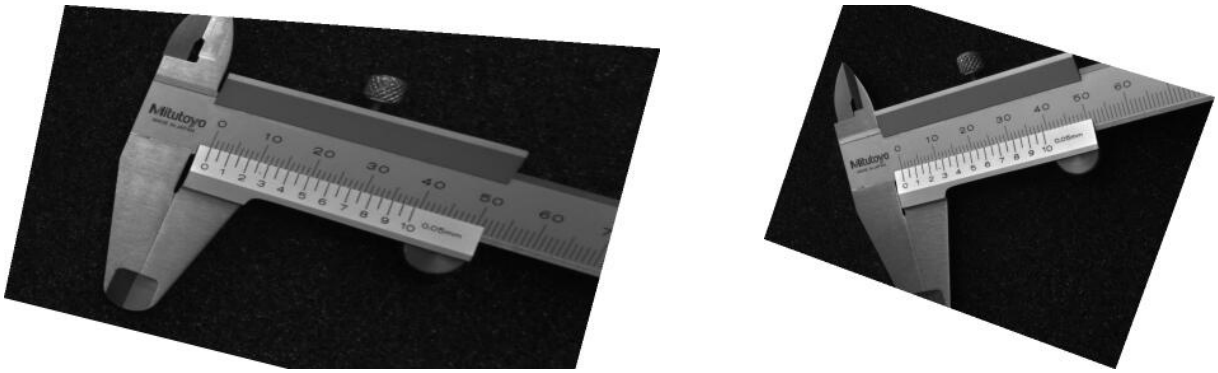


Figure 5.13: Incorrectly rectified stereo images.

5.3.5 Reconstructing 3D Information

There are many possibilities to derive 3D information from rectified stereo images.

Non-metrical information: If only non-metrical information about the surface of an object is needed, it may be sufficient to determine the disparities within the overlapping area of the stereo image pair by using the operator `binocular_disparity` for correlation-based stereo (section 5.3.5.1) or `binocular_disparity_mg` for multigrid stereo (section 5.3.5.2 on page 132). The differences between both stereo matching approaches are described in more detail in section 5.3.1 on page 125.

Distance of the object surface: If metrical information is required, the operator `binocular_distance` (section 5.3.5.4 on page 134) or `binocular_distance_mg` (section 5.3.5.5 on page 136), respectively, can be used to extract the distance of the object surface from the stereo camera system.

3D coordinate images: Having a disparity image of a rectified binocular stereo system, you can additionally derive the corresponding x, y, and z coordinates using `disparity_image_to_xyz` (see section 5.3.5.8 on page 137).

3D coordinates for selected points: To derive metrical information for selected points only, the operators `disparity_to_distance` or `disparity_to_point_3d` can be used. The first of these two operators calculates the distance z of points from the stereo camera system based on their disparity (see section 5.3.5.7 on page 137). The second operator calculates the x, y, and z coordinates from the row and column position of a point in the first rectified image and from its disparity (see section 5.3.5.8 on page 137).

Alternatively, the operator `intersect_lines_of_sight` can be used to calculate the x, y, and z coordinates of selected points (see section 5.3.5.9 on page 137). Then, there is no need to determine the disparities in advance. Only the image coordinates of the conjugate points and the camera parameters are needed. This operator can also handle image coordinates of the original stereo images. Thus, the rectification can be omitted. In exchange, you must determine the conjugate points by yourself.

Note that all operators that deal with disparities or distances require all input to be based on the rectified images. This applies to the image coordinates as well as to the camera parameters.

5.3.5.1 Determining Disparities Using Correlation-Based Stereo

Disparities are an indicator for the distance z of object points from the stereo camera system, since points with equal disparities also have equal distances z (equation 5.4 on page 119).

Therefore, if it is only necessary to know whether there are locally high objects, it is sufficient to derive the disparities. For correlation-based stereo, this is done by using the operator binocular_disparity.

```
binocular_disparity (ImageRectifiedL, ImageRectifiedR, DisparityImage, \
                     ScoreImageDisparity, 'ncc', MaskWidth, MaskHeight, \
                     TextureThresh, MinDisparity, MaxDisparity, NumLevels, \
                     ScoreThresh, 'left_right_check', 'interpolation')
```

The operator requires the two *rectified* images as input. The disparities are derived only for those conjugate points that lie within the respective image domain in both images. With this, it is possible to speed up the calculation of the disparities if the image domain of at least one of the two rectified images is reduced to a region of interest, e.g., by using the operator reduce_domain.

Several parameters can be used to control the behavior of the matching process that is performed by the operator binocular_disparity to determine the conjugate points:

With the parameter Method, the matching function is selected. The methods *'sad'* (summed absolute differences) and *'ssd'* (summed squared differences) compare the gray values of the pixels within a matching window directly, whereas the method *'ncc'* (normalized cross correlation) compensates for the mean gray value and its variance within the matching window. Therefore, if the two images differ in brightness and contrast, the method *'ncc'* should be preferred. However, since the internal computations are less complex for the methods *'sad'* and *'ssd'*, they are faster than the method *'ncc'*.

The width and height of the matching window can be set independently with the parameters MaskWidth and MaskHeight. The values should be odd numbers. Otherwise they will be increased by one. A larger matching window will lead to a smoother disparity image, but may result in the loss of small details. In contrast, the results of a smaller matching window tend to be noisy but they show more spatial details.

Because the matching process relies on the availability of texture, low-textured areas can be excluded from the matching process. The parameter TextureThresh defines the minimum allowed variance within the matching window. For areas where the texture is too low, no disparities will be determined.

The parameters MinDisparity and MaxDisparity define the minimum and maximum disparity values. They are used to restrict the search space for the matching process. If the specified disparity range does not contain the actual range of the disparities, the conjugate points cannot be found correctly. Therefore, the disparities will be incomplete and erroneous. On the other hand, if the disparity range is specified too large, the matching process will be slower and the probability of mismatches increases.

Therefore, it is important to set the parameters MinDisparity and MaxDisparity carefully. There are several possibilities to determine the appropriate values:

- If you know the minimum and maximum distance of the object from the stereo camera system (section 5.3.5.4 on page 134), you can use the operator distance_to_disparity to determine the respective disparity values.
- You can also determine these values directly from the rectified images. For this, you should display the two rectified images and measure the approximate column coordinates of the point N, which is nearest to the stereo camera system (N_{col}^{image1} and N_{col}^{image2}) and of the point F, which is the farthest away (F_{col}^{image1} and F_{col}^{image2}), each in both rectified images.
  Now, the values for the definition of the disparity range can be calculated as follows:

$$MinDisparity = N_{col}^{image2} - N_{col}^{image1} \quad (5.6)$$

$$MaxDisparity = F_{col}^{image2} - F_{col}^{image1} \quad (5.7)$$

The operator binocular_disparity uses image pyramids to improve the matching speed. The disparity range specified by the parameters MinDisparity and MaxDisparity is only used on the uppermost pyramid level, indicated by the parameter NumLevels. Based on the matching results on that level, the disparity range for the matching on the next lower pyramid levels is adapted automatically.

The benefits with respect to the execution time are greatest if the objects have different regions between which the appropriate disparity range varies strongly. However, take care that the value for NumLevels is not set too large, as otherwise the matching process may fail because of lack of texture on the uppermost pyramid level.

The parameter ScoreThresh specifies which matching scores are acceptable. Points for which the matching score is not acceptable are excluded from the results, i.e., the resulting disparity image has a reduced domain that comprises only the accepted points.

Note that the value for ScoreThresh must be set according to the matching function selected via Method. The two methods *'sad'* ($0 \leq$ score ≤ 255) and *'ssd'* ($0 \leq$ score ≤ 65025) return lower matching scores for better matches. In contrast, the method *'ncc'* ($-1 \leq$ score ≤ 1) returns higher values for better matches, where a score of zero indicates that the two matching windows are totally different and a score of minus one denotes that the second matching window is exactly inverse to the first matching window.

The parameter Filter can be used to activate a downstream filter by which the reliability of the resulting disparities is increased. Currently, it is possible to select the method *'left_right_check'*, which verifies the matching results based on a second matching in the reverse direction. Only if both matching results correspond to each other, the resulting conjugate points are accepted. In some cases, this may lead to gaps in the disparity image, even in well textured areas, as this verification is very strict. If you do not want to verify the matching results based on the *'left_right_check'*, set the parameter Filter to *'none'*.

The subpixel refinement of the disparities is switched on by setting the parameter SubDisparity to *'interpolation'*. It is switched off by setting the parameter to *'none'*.

The results of the operator binocular_disparity are the two images Disparity and Score, which contain the disparities and the matching score, respectively. In figure 5.14, a rectified stereo image pair is displayed, from which the disparity and score images that are displayed in figure 5.15 were derived.

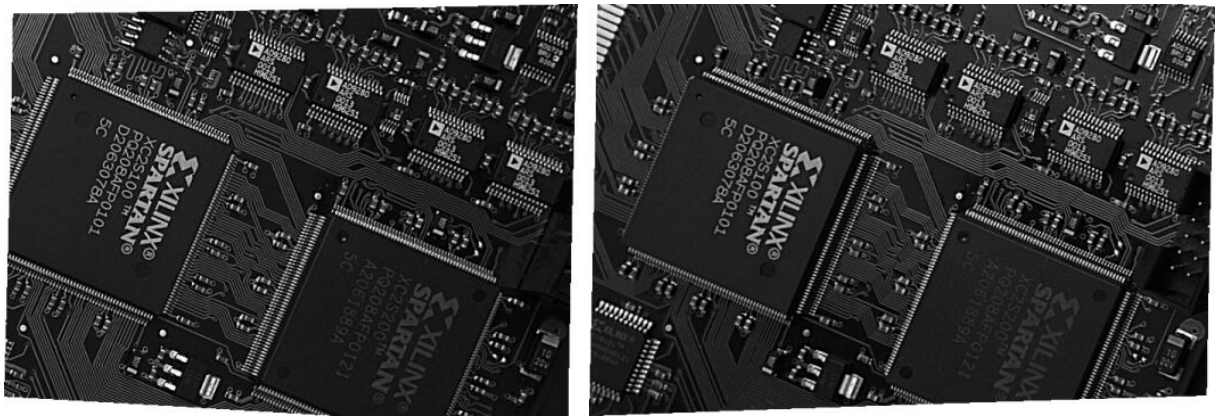


Figure 5.14: Rectified stereo images.

Both resulting images refer to the image geometry of the first rectified image, i.e., the disparity for the point (r,c) of the first rectified image is the gray value at the position (r,c) of the disparity image. The disparity image can, e.g., be used to extract the components of the board, which would be more difficult in the original images, i.e., without the use of 3D information.

In figure 5.15, areas where the matching did not succeed, i.e., undefined regions of the images, are displayed white in the disparity image and black in the score image.

5.3.5.2 Determining Disparities Using Multigrid Stereo

For multigrid stereo, the disparities can be derived with binocular_disparity_mg. Similar to the correlation-based approach, the two rectified images are used as input and the disparity image as well as a score image are returned. But as the disparities are obtained by a different algorithm, the parameters that control the behavior of the multigrid stereo matching process are completely different. In particular, the multigrid-specific control parameters are GrayConstancy, GradientConstancy, Smoothness, InitialGuess, CalculateScore, MGParamName, and MGParamValue. They are explained in detail in the Reference Manual entry for binocular_disparity_mg. The significant differences between the different stereo matching approaches are listed in section 5.3.1 on page 125.

How to determine the disparities of the components on a PCB using binocular_disparity_mg with different levels of accuracy is shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Binocular-Stereo\binocular_disparity_mg.hdev.

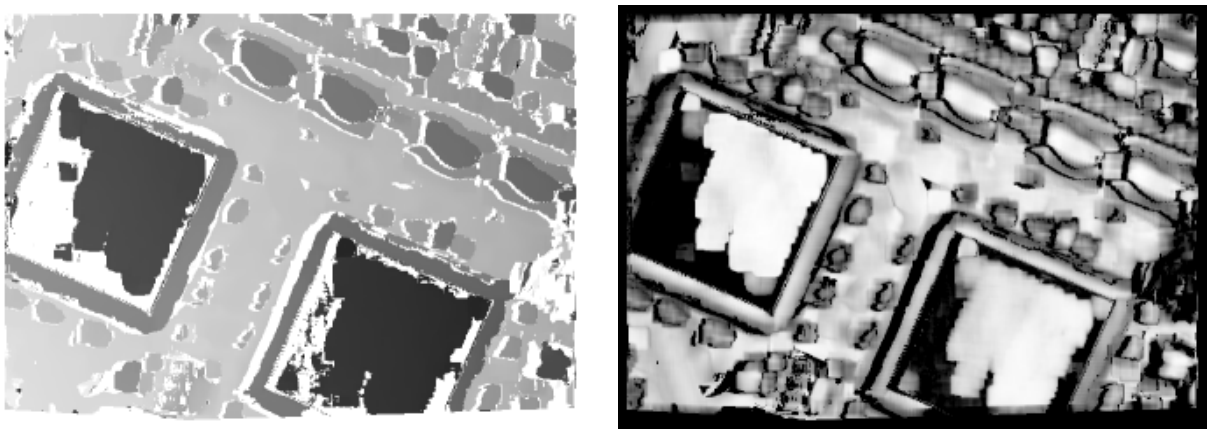


Figure 5.15: Disparity image (left) and score image (right).

There, first the two stereo images are rectified.

```
read_image (ImageL, 'stereo/board/board_l_01')
read_image (ImageR, 'stereo/board/board_r_01')
gen_cam_par_area_scan_division (0.0130507774353, -665.817817207, \
                                1.4803417027e-5, 1.48e-5, 155.89225769, \
                                126.70664978, 320, 240, CamParamL)
gen_cam_par_area_scan_division (0.0131776504517, -731.860636733, \
                                1.47997569293e-5, 1.48e-5, 162.98210144, \
                                119.301040649, 320, 240, CamParamR)
create_pose (0.153573, -0.003734, 0.044735, 0.174289, 319.843388, \
             359.894955, 'Rp+T', 'gba', 'point', RelPose)
gen_binocular_rectification_map (MapL, MapR, CamParamL, CamParamR, RelPose, \
                                 1, 'viewing_direction', 'bilinear', \
                                 RectCamParL, RectCamParR, CamPoseRectL, \
                                 CamPoseRectR, RectLPosRectR)
map_image (ImageL, MapL, ImageRectifiedL)
map_image (ImageR, MapR, ImageRectifiedR)
```

Then, binocular_disparity_mg is called four times, each time with a different parameter for the parameter MGParamValue, which sets the accuracy that should be achieved. Note that an increasing accuracy also leads to an increasing runtime.

```
DefaultParameters := ['fast', 'fast_accurate', 'accurate', 'very_accurate']
for I := 0 to |DefaultParameters| - 1 by 1
    Parameters := DefaultParameters[I]
    binocular_disparity_mg (ImageRectifiedL, ImageRectifiedR, Disparity, \
                            Score, 1, 10, 5, 0, 'false', \
                            'default_parameters', Parameters)
endfor
```

5.3.5.3 Determining Disparities Using Multi-Scanline Stereo

For multi-scanline stereo, the disparities can be derived with binocular_disparity_ms. Similar to the correlation-based approach, the two rectified images are used as input and the disparity image as well as a score image are returned. But as the disparities are obtained by a different algorithm, the parameters that control the behavior of the multi-scanline stereo matching process are completely different. In particular, the control parameters specific for the multi-scanline approach are SurfaceSmoothing and EdgeSmoothing. They are explained in detail in the Reference Manual entry for binocular_disparity_ms. The significant differences between the different stereo matching approaches are listed in section 5.3.1 on page 125.

How to determine the disparities of the components on a PCB using binocular_disparity_ms with different amounts of smoothing is shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\

3D-Reconstruction\Binocular-Stereo\binocular_disparity_ms.hdev.

There, first the two stereo images are rectified.

```
read_image (ImageL, 'stereo/board/board_l_01')
read_image (ImageR, 'stereo/board/board_r_01')
gen_cam_par_area_scan_division (0.0130507774353, -665.817817207, \
                                1.4803417027e-5, 1.48e-5, 155.89225769, \
                                126.70664978, 320, 240, CamParamL)
gen_cam_par_area_scan_division (0.0131776504517, -731.860636733, \
                                1.47997569293e-5, 1.48e-5, 162.98210144, \
                                119.301040649, 320, 240, CamParamR)
create_pose (0.153573, -0.003734, 0.044735, 0.174289, 319.843388, \
             359.894955, 'Rp+T', 'gba', 'point', RelPose)
gen_binocular_rectification_map (MapL, MapR, CamParamL, CamParamR, RelPose, \
                                 1, 'viewing_direction', 'bilinear', \
                                 RectCamParL, RectCamParR, CamPoseRectL, \
                                 CamPoseRectR, RectLPosRectR)
map_image (ImageL, MapL, ImageRectifiedL)
map_image (ImageR, MapR, ImageRectifiedR)
```

Then, binocular_disparity_ms is called four times, each time with a different parameter settings for the parameters SurfaceSmoothing and EdgeSmoothing, which control the amount of smoothing.

```
SurfaceSmoothing := [0, 10, 20, 30, 40, 50, 0, 0, 0, 0, 0, 30]
EdgeSmoothing := [0, 0, 0, 0, 0, 0, 10, 20, 30, 40, 50, 20]
for I := 0 to |SurfaceSmoothing| - 1 by 1
    binocular_disparity_ms (ImageRectifiedL, ImageRectifiedR, Disparity, \
                            Score, 20, 40, SurfaceSmoothing[I], \
                            EdgeSmoothing[I], [], [])
endfor
```

5.3.5.4 Determining Distances Using Correlation-Based Stereo

The distance of an object point from the stereo camera system is defined as its distance from the x-y-plane of the coordinate system of the first rectified camera. For correlation-based stereo, it can be determined by the operator binocular_distance, which is used analogously to the operator binocular_disparity described in section 5.3.5.1 on page 130.

```
binocular_distance (ImageRectifiedL, ImageRectifiedR, DistanceImage, \
                    ScoreImageDistance, RectCamParL, RectCamParR, \
                    RectLPosRectR, 'ncc', MaskWidth, MaskHeight, \
                    TextureThresh, MinDisparity, MaxDisparity, NumLevels, \
                    ScoreThresh, 'left_right_check', 'interpolation')
```

The three additional parameters, namely the camera parameters of the rectified cameras as well as the relative pose of the second rectified camera in relation to the first rectified camera can be taken directly from the output of the operator gen_binocular_rectification_map.

Figure 5.16 shows the distance image and the respective score image for the rectified stereo pair of figure 5.14 on page 132. Because the distance is calculated directly from the disparities and from the camera parameters, the distance image looks similar to the disparity image (figure 5.15). What is more, the score images are identical, since the underlying matching process is identical.

It can be seen from figure 5.16 that the distance of the board changes continuously from left to right. The reason is that, in general, the x-y-plane of the coordinate system of the first rectified camera will be tilted with respect to the object surface (see figure 5.17).

If it is necessary that one reference plane of the object surface has a constant distance value of, e.g., zero, the tilt can be compensated easily: First, at least three points that lie on the reference plane must be defined. These points are used to determine the orientation of the (tilted) reference plane in the distance image. Therefore, they should be selected such that they enclose the region of interest in the distance image. Then, a distance image of

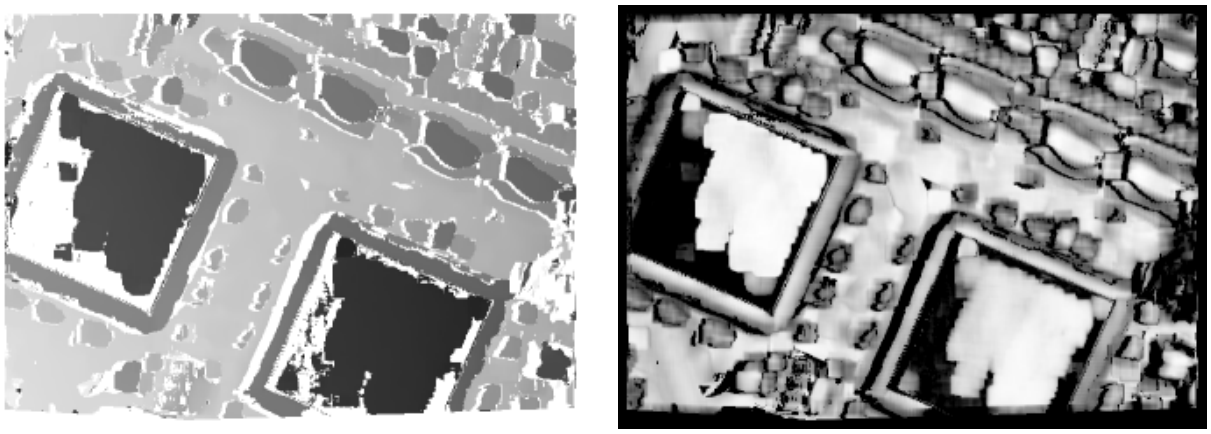


Figure 5.16: Distance image (left) and score image (right).

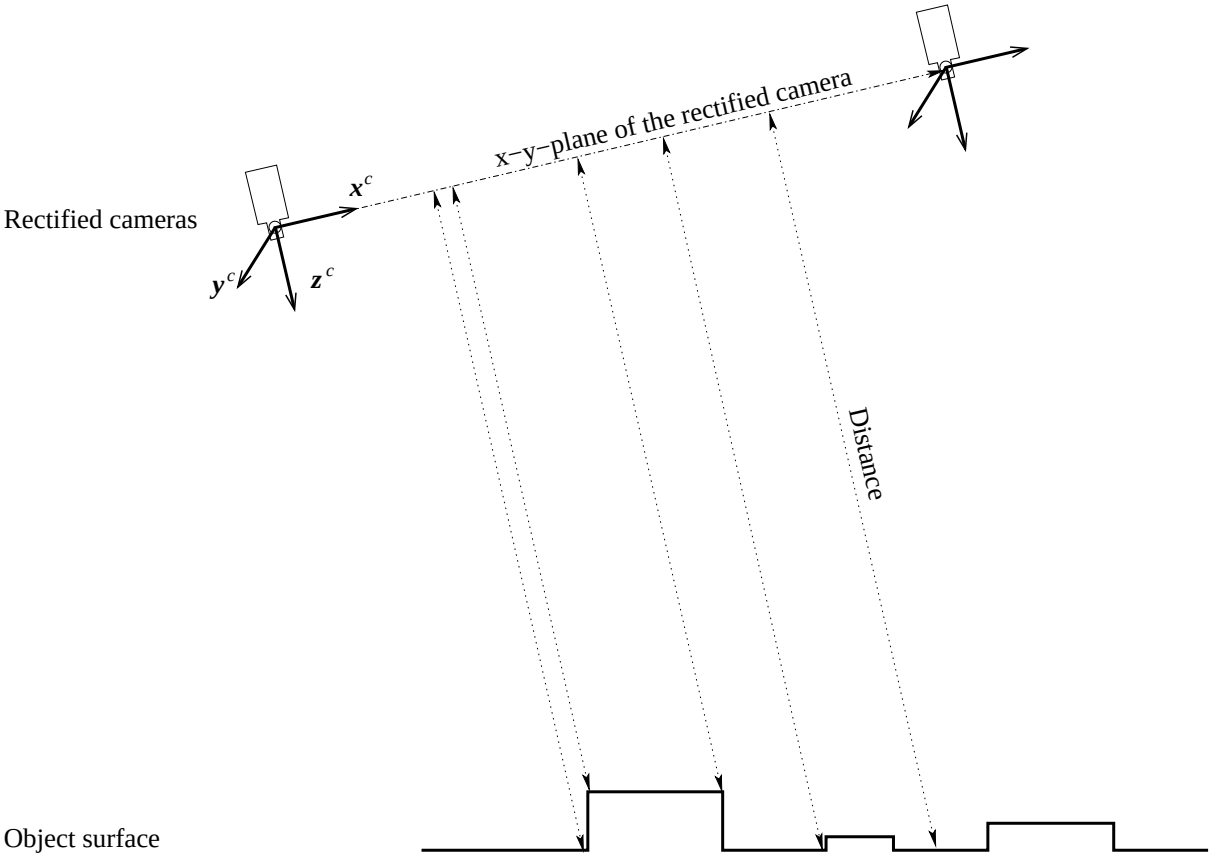


Figure 5.17: Distances of the object surface from the x-y-plane of the coordinate system of the first rectified camera.

the (tilted) reference plane can be simulated and subtracted from the distance image of the object. Finally, the distance values themselves must be adapted by scaling them with the cosine of the angle between the tilted and the corrected reference plane.

These calculations are carried out in the procedure `tilt_correction`, which is part of the example program `%HALCONEXAMPLES%\solution_guide\3d_vision\height_above_reference_plane_from_stereo.hdev` (appendix A.4 on page 233).

```
procedure tilt_correction (DistanceImage, RegionDefiningReferencePlane,
                           DistanceImageCorrected):::
```

In principle, this procedure can also be used to correct the disparity image, but note that you must not use the

corrected disparity values as input to any operators that derive metric information.

If the reference plane is the ground plane of the object, an inversion of the distance image generates an image that encodes the heights above the ground plane. Such an image is displayed on the left hand side in figure 5.18.

Objects of different height above or below the ground plane can be segmented easily using a simple `threshold` with the minimal and maximal values given directly in units of the world coordinate system, e.g., meters. The image on the right hand side of figure 5.18 shows the results of such a segmentation, which can be carried out based on the corrected distance image or the image of the heights above the ground plane.

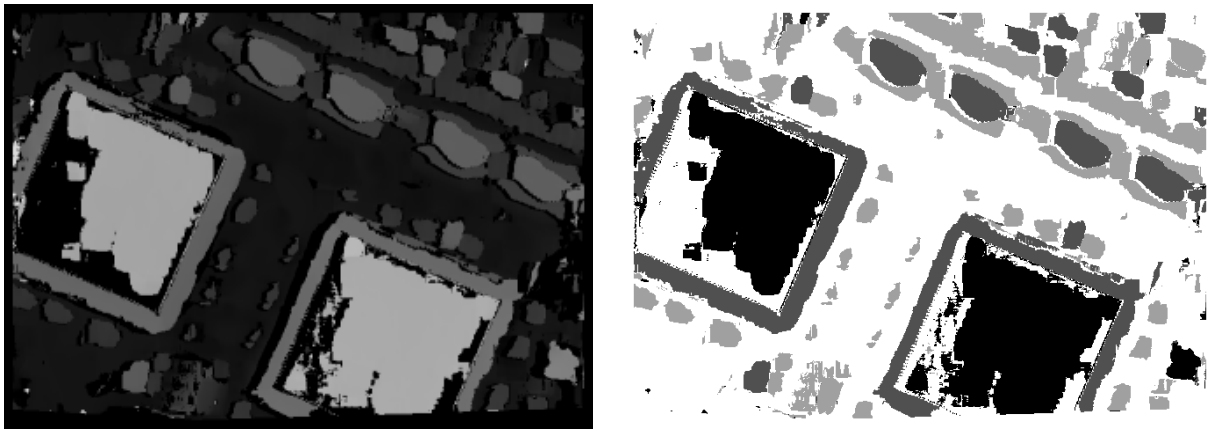


Figure 5.18: Left: Height above the reference plane; Right: Segmentation of high objects (white: 0-0.4 mm, light gray: 0.4-1.5 mm, dark gray: 1.5-2.5 mm, black: 2.5-5 mm).

5.3.5.5 Determining Distances Using Multigrid Stereo

For multigrid stereo, the distance of an object point from the stereo camera system can be determined with the operator `binocular_distance_mg`, which is used analogously to the operator `binocular_disparity_mg` described in section 5.3.5.2 on page 132, but with the three additional parameters described also for `binocular_distance`. These are the camera parameters of the rectified cameras as well as the relative pose of the second rectified camera in relation to the first rectified camera, which can be taken directly from the output of the operator `gen_binocular_rectification_map`.

How to determine the depths of the components on a PCB with high accuracy is shown in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Binocular-Stereo\binocular_distance_mg.hdev`.

```
binocular_distance_mg (ImageRectifiedL, ImageRectifiedR, Distance, Score, \
                       RectCamParL, RectCamParR, RectLPosRectR, 1, 10, 5, \
                       0, 'false', 'default_parameters', 'accurate')
```

5.3.5.6 Determining Distances Using Multi-Scanline Stereo

For multi-scanline stereo, the distance of an object point from the stereo camera system can be determined with the operator `binocular_distance_ms`, which is used analogously to the operator `binocular_disparity_ms` described in section 5.3.5.3 on page 133, but with the three additional parameters described also for `binocular_distance`. These are the camera parameters of the rectified cameras as well as the relative pose of the second rectified camera in relation to the first rectified camera, which can be taken directly from the output of the operator `gen_binocular_rectification_map`.

How to determine the depths of the components on a PCB with high accuracy, especially at discontinuities, is shown in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Binocular-Stereo\binocular_distance_ms.hdev`.

```
binocular_distance_ms (ImageRectifiedL, ImageRectifiedR, Distance, Score, \
                       RectCamParL, RectCamParR, RelPose, 20, 40, 30, 20, \
                       [], [])
```

5.3.5.7 Determining Distances for Selected Points from the Disparity Image

If only the distances of selected points should be determined, the operator disparity_to_distance can be used. It simply transforms given disparity values into the respective distance values. For example, if you want to know the distances of two given points from the stereo camera system you can determine the respective disparities from the disparity image and transform them into distances.

```
get_grayval (Disparity, RowL, ColumnL, DisparityOfSelectedPoints)
disparity_to_distance (RectCamParL, RectCamParR, RectLPosRectR, \
                       DisparityOfSelectedPoints, DistanceOfPoints)
```

This transformation is constant for the entire rectified image, i.e., all points having the same disparity have the same distance from the x-y-plane of the coordinate system of the first rectified camera. Therefore, besides the camera parameters of the rectified cameras, only the disparity values need to be given.

5.3.5.8 Determining 3D Coordinates from the Disparity Image

If the x, y, and z coordinates of points have to be calculated, different operators are available that derive the corresponding information from the disparity image:

disparity_to_point_3d computes the 3D coordinates for specified image coordinates and returns them in three tuples.

```
disparity_to_point_3d (RectCamParL, RectCamParR, RectLPosRectR, RowL, \
                       ColumnL, DisparityOfSelectedPoints, \
                       X_CCS_FromDisparity, Y_CCS_FromDisparity, \
                       Z_CCS_FromDisparity)
```

disparity_image_to_xyz computes 3D coordinates for the complete image and returns them in three images (see the HDevelop example program %HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Binocular-Stereo\disparity_image_to_xyz.hdev).

```
disparity_image_to_xyz (DisparityImage, X, Y, Z, RectCamParL, RectCamParR, \
                        RectLPosRectR)
```

The operators require the camera parameters of the two rectified cameras and the relative pose of the cameras.

The x, y, and z coordinates are returned in the coordinate system of the first rectified camera.

5.3.5.9 Determining 3D Coordinates for Selected Points from Point Correspondences

If only the 3D coordinates of selected points should be determined, you can alternatively use the operator intersect_lines_of_sight to determine the x, y, and z coordinates of points from the image coordinates of the respective conjugate points. Note that you must determine the image coordinates of the conjugate points yourself.

```
intersect_lines_of_sight (RectCamParL, RectCamParR, RectLPosRectR, RowL, \
                          ColumnL, RowR, ColumnR, X_CCS_FromIntersect, \
                          Y_CCS_FromIntersect, Z_CCS_FromIntersect, \
                          Dist)
```

The x, y, and z coordinates are returned in the coordinate system of the first (rectified) camera.

The operator can also handle image coordinates of the original stereo images. Thus, the rectification can be omitted. In this case, the camera parameters of the original stereo cameras have to be given instead of the parameters of the rectified cameras.

It is possible to transform the x, y, and z coordinates determined by the latter two operators from the coordinate system of the first (rectified) camera into a given world coordinate system (WCS), e.g., a coordinate system with

respect to the building plan of a factory building. For this, a homogeneous transformation matrix, which describes the transformation between the two coordinate systems, is needed.

This homogeneous transformation matrix can be determined in various ways. The easiest way is to take an image of a HALCON calibration plate with the first camera only. If the 3D coordinates refer to the *rectified* camera coordinate system, the image must be rectified as well. Then, the pose of the calibration plate in relation to the first (rectified) camera can be determined using the operators `find_calib_object` and `get_calib_data_observ_points` (note that before applying these operators a HALCON calibration data model must be created and initialized).

```
find_calib_object (ImageRectifiedL, CalibDataID, 0, 0, 0, [], [])
get_calib_data_observ_points (CalibDataID, 0, 0, 0, RCoordL, CCoordL, \
                              IndexLnew, PoseL)
```

The resulting pose can be converted into a homogeneous transformation matrix.

```
pose_to_hom_mat3d (PoseL, HomMat3D_WCS_to_RectCCS)
```

If necessary, the transformation matrix can be modified with the operators `hom_mat3d_rotate_local`, `hom_mat3d_translate_local`, and `hom_mat3d_scale_local`.

```
hom_mat3d_translate_local (HomMat3D_WCS_to_RectCCS, 0.01125, -0.01125, 0, \
                           HomMat3DTranslate)
hom_mat3d_rotate_local (HomMat3DTranslate, rad(180), 'y', \
                        HomMat3D_WCS_to_RectCCS)
```

The homogeneous transformation matrix must be inverted in order to represent the transformation from the (rectified) camera coordinate system into the WCS.

```
hom_mat3d_invert (HomMat3D_WCS_to_RectCCS, HomMat3D_RectCCS_to_WCS)
```

Finally, the 3D coordinates can be transformed using the operator `affine_trans_point_3d`.

```
    affine_trans_point_3d (HomMat3D_RectCCS_to_WCS, X_CCS_FromIntersect, \
                           Y_CCS_FromIntersect, Z_CCS_FromIntersect, X_WCS, \
                           Y_WCS, Z_WCS)
```

The homogeneous transformation matrix can also be determined from three specific points. If the origin of the WCS, a point on its x-axis, and a third point that lies in the x-y-plane, e.g., directly on the y-axis, are given, the transformation matrix can be determined using the procedure `gen_hom_mat3d_from_three_points`, which is part of the HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\3d_information_for_selected_points.hdev`.

```
procedure gen_hom_mat3d_from_three_points (Origin, PointOnXAxis,
                                           PointInXYPlane, HomMat3d):::
```

The resulting homogeneous transformation matrix can be used as input for the operator `affine_trans_point_3d`, as shown above.

5.3.6 Uncalibrated Stereo Vision

Similar to uncalibrated mosaicking (see chapter 10 on page 205), HALCON also provides an "uncalibrated" version of stereo vision, which derives information about the cameras from the scene itself by matching characteristic points. The main advantage of this method is that you need no special calibration object. The main disadvantage of this method is that, without a precisely known calibration object, the accuracy of the result is highly dependent on the content of the scene, i.e., the accuracy of the result degrades if the scene does not contain enough 3D information or if the extracted characteristic points in the two images do not precisely correspond to the same world points, e.g., due to occlusion.

In fact, HALCON provides two versions of uncalibrated stereo: **Without any knowledge about the cameras and about the scene**, you can rectify the stereo images and perform a segmentation similar to the method described in section 5.3.5.4 on page 134. For this, you first use the operator `match_fundamental_matrix_ransac`, which determines the so-called *fundamental matrix*. This matrix models the cameras and their relation. But in contrast to the model described in section 5.1 on page 117, internal and external parameters are not available separately. Thus, no metric information can be derived.

The fundamental matrix is then passed on to the operator `gen_binocular_proj_rectification`, which is the "uncalibrated" version of the operator `gen_binocular_rectification_map`. With the output of this operator, i.e., the rectification maps, you can then proceed to rectify the images as described in section 5.3.4 on page 127. Because the relative pose of the cameras is not known, you cannot generate the distance image and segment it as described in section 5.3.5.4 on page 134. The HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Object-Recognition-2D\board_segmentation_uncalib.hdev` shows an alternative approach that can be used if the reference plane appears dominant in the images, i.e., if many correspondences are found on it.

Because no calibration object is needed, this method can be used to perform stereo vision with a single camera. Note, however, that the method assumes that there are no lens distortions in the images. Therefore, the accuracy of the results degrades if the lens has significant distortions.

If the internal parameters of the camera are known, you can determine the relative pose between the cameras using the operator `match_rel_pose_ransac` and then use all the stereo methods described for the fully calibrated case. There is, however, a limitation: The determined pose is relative in a second sense, because it can be determined only up to a scale factor. The reason for this is that without any knowledge about the scene, the algorithm cannot know whether the points in the scene are further away or whether the cameras are further apart because the effect in the image is the same in both cases. If you have additional information about the scene, you can solve this ambiguity and determine the "real" relative pose. This method is shown in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\3D-Reconstruction\Binocular-Stereo\uncalib_stereo_boxes.hdev`.

5.4 Multi-View Stereo Vision

In comparison to binocular stereo, multi-view stereo allows

- to use more than two cameras and thus to reconstruct 3D information from all around an object and
- to reconstruct surfaces and 3D coordinates of selected points in form of 3D object models.

Internally, it is based on binocular stereo. However, by default it does not return the disparity image as a result.

The following sections show

- how to initialize the stereo model (section 5.4.1) and
- how to reconstruct 3D information (section 5.4.2 on page 141).

5.4.1 Initializing the Stereo Model

The operators for multi-view stereo use a so-called *stereo model* to encapsulate all needed data. The following sections show

- how to access the calibration results (section 5.4.1.1),
- how to specify the world coordinate system (section 5.4.1.2), and
- how to create the stereo model (section 5.4.1.3).

5.4.1.1 Accessing the Calibration Results

In contrast to binocular stereo (see section 3.2.7 on page 72), for multi-view stereo you access the results of the calibration not separately but in form of a so-called *camera setup model*, which contains the internal camera parameters as well as the relative poses between the cameras. To derive the camera setup model from the calibration data model, which must have been calibrated before with `calibrate_cameras` as is described in section 5.2 on page 122, you call the operator `get_calib_data` as follows:

```
get_calib_data (CalibDataID, 'model', 'general', 'camera_setup_model', \
                CameraSetupModelID)
```

If you want to perform the calibration in an offline step, you can save the camera setup model with the operator `write_camera_setup_model`.

```
write_camera_setup_model (CameraSetupModelID, 'four_camera_setup_model.csm')
```

5.4.1.2 Specifying the Coordinate System of the Camera Setup

The coordinate system of the stereo camera setup is identical to the coordinate system of the so-called *reference camera* of the setup, which is typically the camera with the index 0 (see the upper part of figure 5.19). The poses of the other cameras and the reconstructed coordinates are computed relative to this camera.

You can change the setup's coordinate system with the operator `set_camera_setup_param` in two ways. In particular, you can

- **select another camera as reference camera** by setting `CameraIdx` to `'general'` and `GenParamName` to `'reference_camera'` and passing the index of the camera in `GenParamValue` or
- **specify the pose of the desired setup coordinate system** (relative to the current one) by setting `CameraIdx` to `'general'` and `GenParamName` to `'coord_transf_pose'` and passing the pose in `GenParamValue`.

The latter case is shown at the bottom of figure 5.19. There, the desired coordinate system is marked by the calibration plate (typically, you would add a rotation to let the z axis point upwards).

How to change the pose of the setup's coordinate system is shown, e.g., in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Calibration\Multi-View\calibrate_cameras_multiple_camera_setup.hdev`. There, it is moved from the reference camera to the calibration plate with pose 0. For that, pose 0 of the calibration plate relative to the reference camera is accessed with `get_calib_data` using the parameter `'calib_obj_pose'` and, to consider the thickness of the calibration plate, the z coordinate of the pose is modified with `set_origin_pose`. Then, the setup's coordinate system is moved into this pose with `set_camera_setup_param`.

```
RefPoseIndex := 0
get_calib_data (CalibDataID, 'calib_obj_pose', [0,RefPoseIndex], 'pose', \
                PoseCam0Indx0)
set_origin_pose (PoseCam0Indx0, 0, 0, CaltabThickness, ReferencePose)
set_camera_setup_param (CameraSetupModelID, 'general', 'coord_transf_pose', \
                        ReferencePose)
```

5.4.1.3 Creating the Stereo Model

After adapting the camera setup model to your requirements, you pass it to the operator `create_stereo_model`, which creates the stereo model. Note that at this point you must already specify whether you want to reconstruct points or surfaces! To reconstruct surfaces, use either the parameter `'surface_pairwise'` or `'surface_fusion'`.

```
create_stereo_model (CameraSetupModelID, 'surface_pairwise', [], [], \
                     StereoModelID)
```

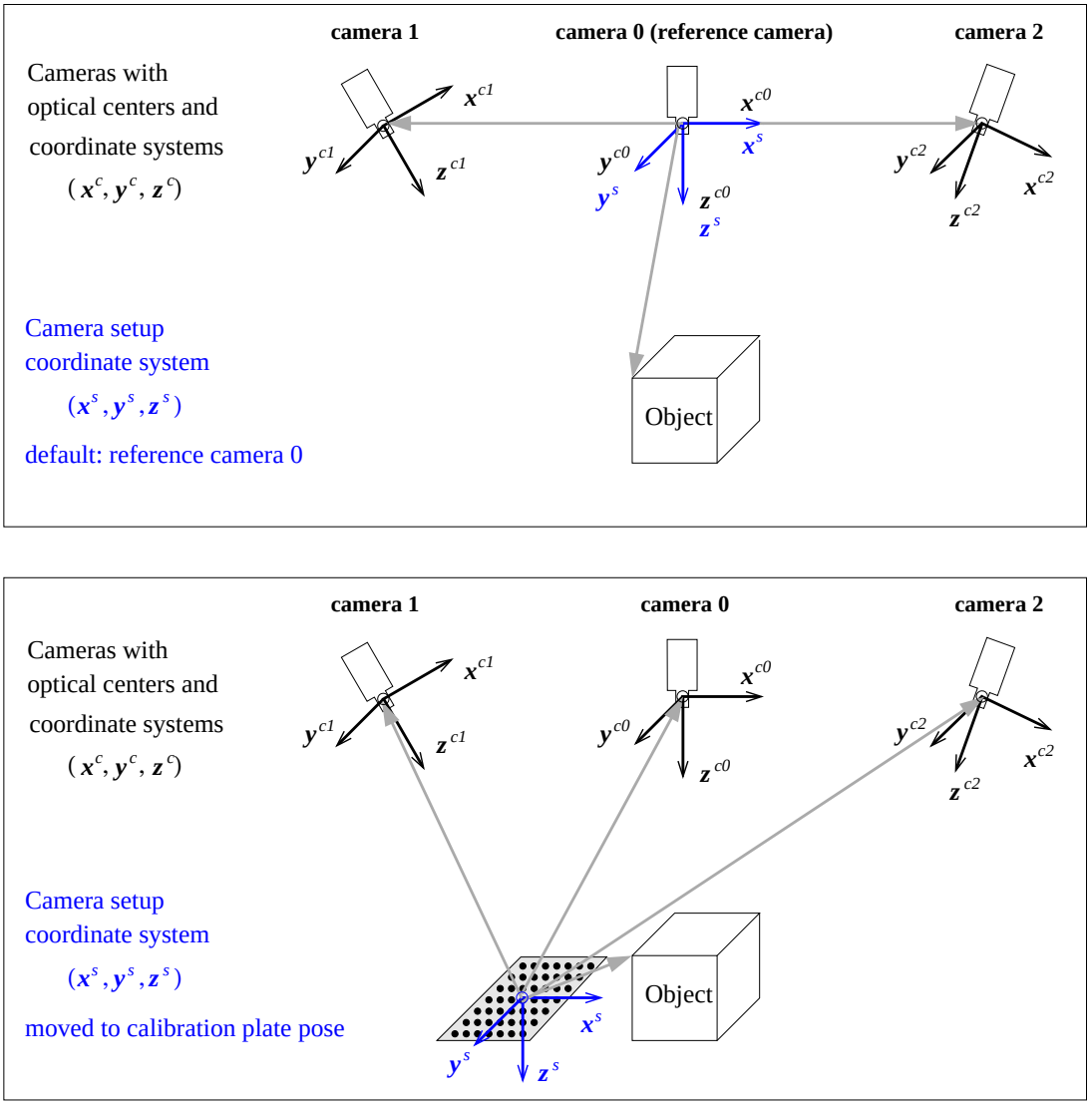


Figure 5.19: Coordinate systems of a multi-view camera setup: Top: default setup coordinate system located in the reference camera 0; bottom: setup coordinate system moved to the coordinate system of the calibration plate.

To reconstruct points, call the operator with the parameter `'points_3d'`.

```
create_stereo_model (CameraSetupModelID, 'points_3d', [], [], StereoModelID)
```

5.4.2 Reconstructing 3D Information

With multi-view stereo, you can reconstruct

- the surface of an object (section 5.4.2.1) or
- 3D coordinates for selected points (section 5.4.2.2 on page 143).

5.4.2.1 Reconstructing Surfaces

The main functionality of multi-view stereo is the reconstruction of surfaces. Two methods are available: `'surface_pairwise'` and `'surface_fusion'`. The pairwise reconstruction is the faster method and may be sufficient for many applications. For example, 3D Matching can often deal quite well with

the results of 'surface_pairwise'. However, the fusion method has many advantages, like an improved noise reduction and outlier suppression. In general, the visual results of the fusion method are much better. Have a look at the example program %HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Multi-View\reconstruct_surface_stereo_fusion.hdev for a comparison of both methods. Additionally, the example programs %HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Multi-View\reconstruct_surface_stereo_pairwise_workflow.hdev and %HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Multi-View\reconstruct_surface_stereo_fusion_workflow.hdev explain the recommended workflow for obtaining good results with both methods.

Note that if you want to reconstruct surfaces from multiple 3D object models without any camera setup information (e.g., sensors that return point clouds), you can use fuse_object_model_3d.

Below, we explain the main steps using the operator reconstruct_surface_stereo with the method 'surface_pairwise'. The code fragments belong to the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Robot-Vision\locate_pipe_joints_stereo.hdev, which reconstructs pipe joints. For the reconstruction, four cameras are used. Afterwards, surface-based 3D matching is performed to estimate the pose of the individual pipe joints, see Solution Guide I, section 11.3.2 on page 108). Figure 5.1 on page 117 shows the four camera images and the reconstructed surface.

After creating and initializing the stereo model, you must call the operator set_stereo_model_image_pairs to specify which cameras form pairs, i.e., between which camera images the disparity images are to be computed. In the example, the cameras 0 and 1 and the cameras 2 and 3, respectively, form pairs. Before that, we set some parameters that specify the interpolation mode and sub-sampling factor for the rectification maps.

```
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 surface reconstruction must be restricted to a specific part of the 3D space, which is realized by the definition of a bounding box. This box is built by the coordinates of its front lower left corner and its back upper right corner. In the program, a camera setup model is used for which the coordinate system was moved to the object, or more precisely, to a calibration plate that was used for the calibration of the scene in which the object was placed. Relative to this calibration plate, the coordinates are specified in meters.

```
set_stereo_model_param (StereoModelID, 'bounding_box', [-0.2, -0.07, -0.075, \
                        0.2, 0.07, -0.004])
```

Before calling the reconstruction operator, the model can be configured for the stereo reconstruction using set_stereo_model_param. With this operator, several parameters are adjusted that control the reconstruction. For example, as the multi-view surface reconstruction is based on computing binocular disparity images (see section 5.3.5.1 on page 130), several parameters are used to configure the binocular image rectification and the internally called binocular stereo operators.

```
* -> Subsampling X, Y, Z
set_stereo_model_param (StereoModelID, 'sub_sampling_step', 3)
* -> Binocular disparity parameters.
set_stereo_model_param (StereoModelID, 'binocular_filter', \
                        'left_right_check')
```

Then, the actual surface reconstruction is applied with the operator reconstruct_surface_stereo.

```
    reconstruct_surface_stereo (Images, StereoModelID, PipeJointPileOM3DID)
```

If the reconstruction fails, please refer to the Reference Manual entry of reconstruct_surface_stereo, which contains detailed information about troubleshooting.

The reconstructed surface is returned as a 3D object model (see page 38), which by default consists of points and their normals. If you need a surface that contains meshing information, e.g., because you want to apply a 3D primitives fitting (see section 4.5 on page 111) to the 3D object model, you have to additionally set the parameter 'point_meshing' within set_stereo_model_param before building the camera pairs for the reconstruction.

In the example, the 3D object model is visualized by the procedure `visualize_object_model_3d`, which allows to interactively rotate, move, and zoom into the model.

```
create_pose (0.0, 0.0, 0.5, -30, 0, 180, 'Rp+T', 'gba', 'point', PoseIn)
if (Index == 1)
    visualize_object_model_3d (WindowHandle1, PipeJointPileOM3DID, \
                               CamParam0, PoseIn, ['color', \
                               'point_size'], ['yellow', 1], \
                               'Reconstructed scene in ' + ReconsTime$'.3' + ' s', \
                               [], Instructions, PoseOut)
endif
```

5.4.2.2 Reconstructing 3D Points

Multi-view stereo also allows to reconstruct the 3D coordinates of selected points. The main advantages of using the multi-view approach in comparison to the binocular variant (see, section 5.3.5.9 on page 137) are that the reconstruction is more accurate when more than two lines of sight can be taken into account and that points located on different sides of an object can be reconstructed.

After creating and initializing the stereo model, you can directly use multi-view stereo to reconstruct the 3D coordinates from point correspondences. This is shown in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Multi-View\reconstruct_points_stereo.hdev`, where four cameras are used to reconstruct the coordinates of the calibration marks of a calibration plate in three different poses (see figure 5.20).

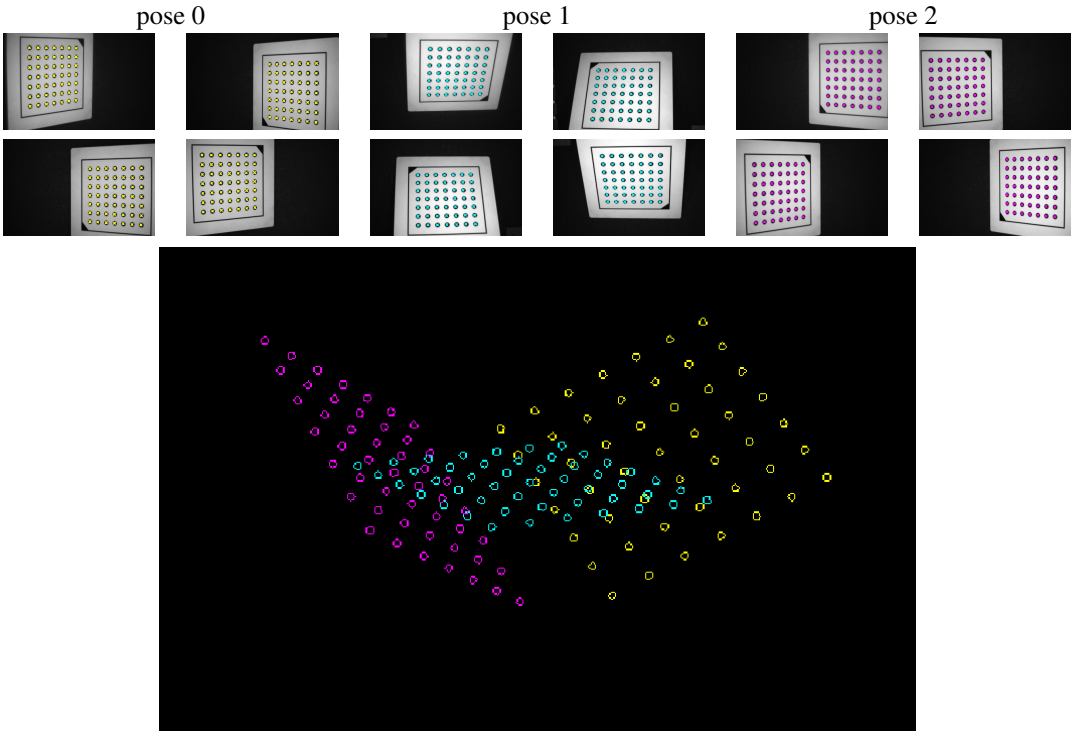


Figure 5.20: Top: images of four cameras of the calibration plate with extracted marks in three different poses; Bottom: reconstructed points.

The main input for the reconstruction operator `reconstruct_points_stereo` are the corresponding points from the multi-view images. They must be passed as tuples in the parameters

- `Row` (row coordinate of the point),
- `Column` (column coordinate of the point),
- `CameraIdx` (index of the camera), and

- PointIdx (index of the point).

Generally, you must extract the corresponding points by yourself. In the example, this task is easy, because here the points correspond to the marks of a calibration plate, which can be easily extracted using an initialized calibration data model and the operators find_calib_object and get_calib_data_observ_points. For other objects than the HALCON calibration plate, the extraction may be a bit more challenging.

In the example, the extraction is realized as follows: First, caltab_points is used to derive the number of calibration marks from the description of the used calibration plate. As with find_calib_object and get_calib_data_observ_points the calibration marks are always extracted in the same order, the tuple with the point indices of the correspondence information for a single image can be created using tuple_gen_sequence. As the parameter CameraIdx must be a tuple of the same length as Row, Column, and PointIdx, a tuple with the same length for which each element is '1' is created, which is used later to assign the correct correspondence values to the camera indices.

```
caltab_points (CaltabDescr, X, Y, Z)
tuple_gen_sequence (0, |X| - 1, 1, Indices)
Ones := gen_tuple_const(|X|,1)
```

Now, a calibration data model is created and prepared for the extraction of the calibration marks. Then, for each pose of the calibration plate, empty tuples for the correspondence information are created and filled with the values obtained for each camera that images the calibration plate under this pose. In particular, for each camera, the calibration marks are extracted with find_calib_object and get_calib_data_observ_points. The resulting row and column coordinates are added to the tuples AllRow and AllColumn. The tuple with the point indices is added to the tuple AllIndices and the corresponding elements with the specific camera index are added to the tuple AllCams.

```
Objects3D := []
create_calib_data ('calibration_object', 4, 1, CalibDataID)
set_calib_data_calib_object (CalibDataID, 0, CaltabDescr)
for PoseIndex := 0 to 2 by 1
    AllRow := []
    AllColumn := []
    AllIndices := []
    AllCams := []
    for CameraIndex := 0 to 3 by 1
        get_camera_setup_param (CameraSetupModelID, CameraIndex, 'params', \
                                CameraParam)
        ImageFile := ImgPath + 'multi_view_calib_cam_' + CameraIndex + '_'  \
                     + (13 + PoseIndex)$'02'
        read_image (Image, ImageFile)
        set_calib_data_cam_param (CalibDataID, CameraIndex, [], CameraParam)
        find_calib_object (Image, CalibDataID, CameraIndex, 0, 0, [], [])
        get_calib_data_observ_points (CalibDataID, CameraIndex, 0, 0, Row, \
                                      Column, Index, Pose)
        AllRow := [AllRow,Row]
        AllColumn := [AllColumn,Column]
        AllIndices := [AllIndices,Indices]
        AllCams := [AllCams,CameraIndex * Ones]
    endfor
```

After accumulating the correspondence information for all cameras that image the calibration plate under the specific pose, the reconstruction is applied with reconstruct_points_stereo.

```
    reconstruct_points_stereo (StereoModelID, AllRow, AllColumn, [], \
                               AllCams, AllIndices, X, Y, Z, CovWP, \
                               PointIndexOut)
```

It returns tuples with the x, y, and z coordinates and with the index of those points that could be reconstructed, i.e., which were extracted in two or more images. In the example, the coordinates are transformed via x, y, and z images into a 3D object model and the models of all three calibration plate poses are interactively visualized with visualize_object_model_3d, which allows to rotate, move, and zoom into the model.

```
gen_image_const (ImageX, 'real', 1, |X|)
gen_image_const (ImageY, 'real', 1, |X|)
gen_image_const (ImageZ, 'real', 1, |X|)
...
    set_grayval (ImageX, Indices, 0 * Indices, X)
    set_grayval (ImageY, Indices, 0 * Indices, Y)
    set_grayval (ImageZ, Indices, 0 * Indices, Z)
    xyz_to_object_model_3d (ImageX, ImageY, ImageZ, ObjectModel3DID)
    Objects3D := [Objects3D,ObjectModel3DID]
    disp_continue_message (WindowHandles[3], 'black', 'true')
    stop ()
endfor
visualize_object_model_3d (WindowHandle, Objects3D, CameraParam, [], \
                           'color_attrib', 'coord_z', [], [], Instructions, \
                           PoseOut)
```

Please note that the example shows only the main functionality of the stereo point reconstruction. For more information, e.g., about input and output covariances or the influence of the bounding box of the stereo model, please refer to the Reference Manual entry of `reconstruct_points_stereo`.

Chapter 6

Laser Triangulation with Sheet of Light

Laser triangulation can be used to reconstruct the surface of a 3D object by approximating it via a set of height profiles. HALCON provides operators for a special type of laser triangulation that is called sheet-of-light technique.

6.1 The Principle of Sheet of Light

The basic idea of the sheet-of-light technique is to project a thin luminous straight line, e.g., generated by a laser line projector, onto the surface of the object that is to be reconstructed and then image the projected line with a camera. As shown in figure 6.1 the projection of the laser line builds a plane that is called light plane or sheet of light. The optical axis of the camera and the light plane form an angle α, which is called angle of triangulation. The points of intersection between the laser line and the camera view depend on the height of the object. Thus, if the object onto which the laser line is projected differs in height, the line is not imaged as a straight line but represents a profile of the object. Using this profile, we can obtain the height differences of the object. To reconstruct the whole surface of an object, i.e., to get many height profiles, the object must be moved relative to the measurement system, i.e., the unit built by the laser line projector and the camera.

The sheet-of-light technique can be applied either to a calibrated measurement setup or to the uncalibrated setup. If the setup is calibrated, the measurement returns the disparities, the x, y, and z coordinates of the points that build the profiles in the world coordinate system (WCS, see figure 6.1), and a 3D object model that is derived from the x, y, and z coordinates. The disparities are returned in form of a disparity image, i.e., the disparities of each measured profile are stored in one row of the disparity image (see figure 6.2 and note that the camera must be oriented such that the profiles are roughly parallel to the rows of the image). The x, y, and z coordinates are also not explicitly returned as values but are expressed as values of pixels within images. That is, three images are returned, one for the x coordinates (`X`), one for the y coordinates (`Y`), and one for the z coordinates (`Z`). The 3D object model contains information about the 3D coordinates and the corresponding 2D mapping. If the setup is uncalibrated, only the disparity image and a score that describes how reliable the measurement result is can be returned by the measurement.

Note that the disparity image for sheet of light has not exactly the same meaning as the disparity image described for stereo matching in section 5.1 on page 117 and section 5.3.5.1 on page 130. For stereo, the disparity describes the difference between the row coordinates of the left and right stereo images, whereas for sheet of light, the disparity is built by the subpixel row values at which the profile was detected.

6.2 The Measurement Setup

The hardware needed for a sheet-of-light measurement consists of a projector that is able to project a thin luminous line, a camera, a positioning system, and the object to measure. In the following, we assume the projector to be a laser line projector and the positioning system to be linear (e.g., a conveyor belt), as these are very common in laser triangulation applications.

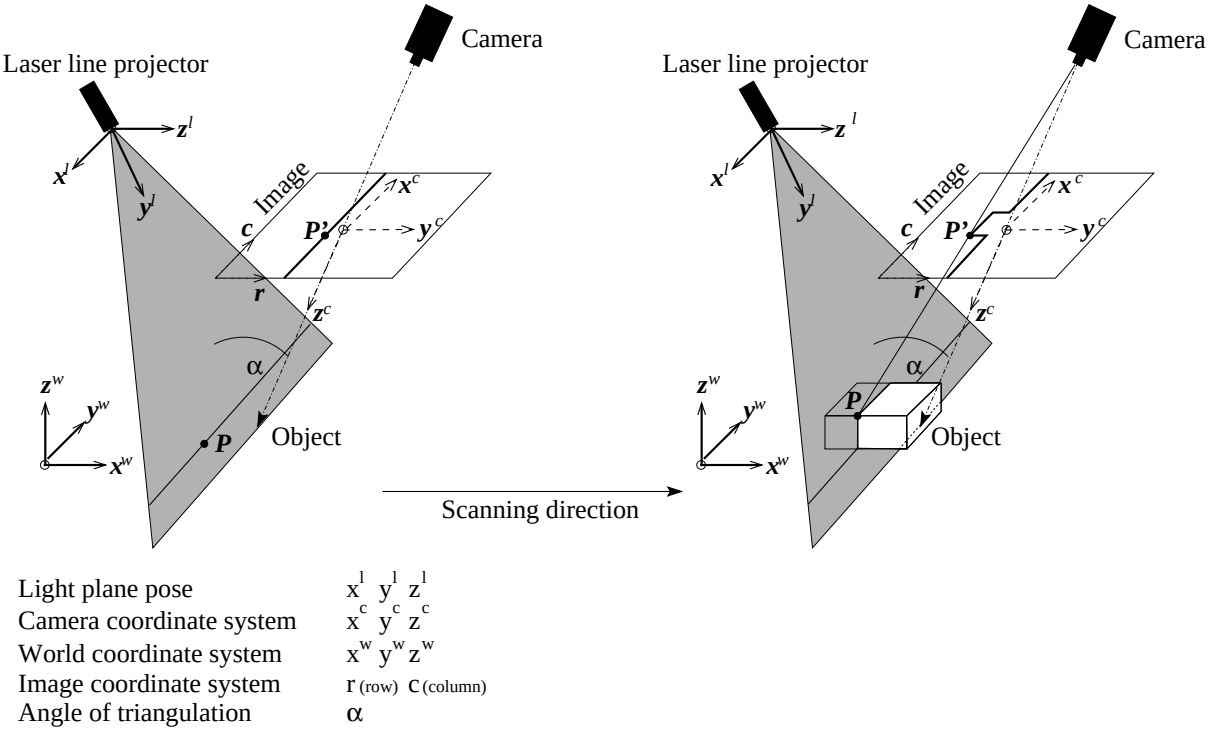


Figure 6.1: Basic principle of sheet of light (light plane marked in gray).

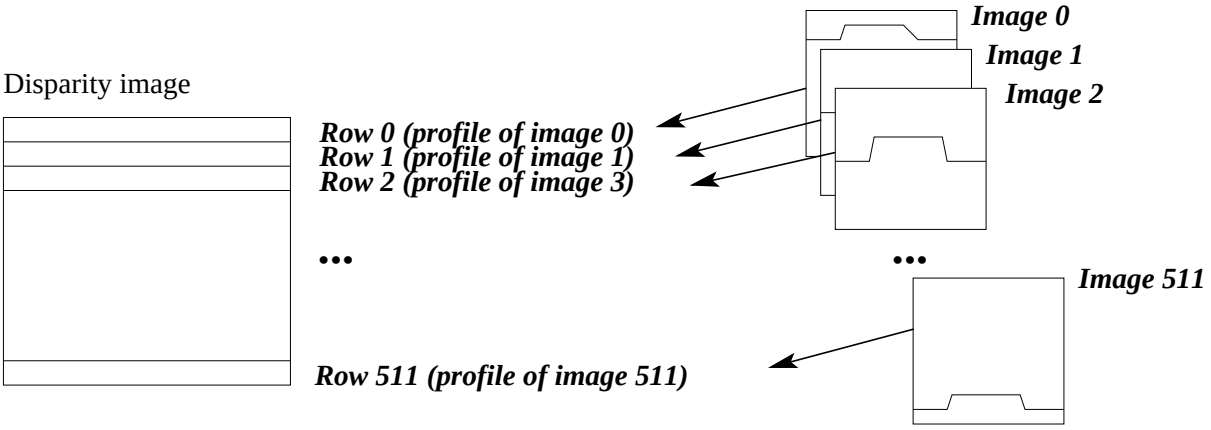


Figure 6.2: Disparity image: the disparity obtained from each profile (or image, respectively) is stored in a row of the disparity image.

The relation between the projector, the camera, and the linear positioning system must not be changed, whereas the position of the object that is transported by the positioning system changes in relation to the projector-camera unit. Since the profile images are processed column by column, the profiles must be oriented roughly horizontal, i.e., roughly parallel to the rows of the image.

The relation between the laser line projector, the camera, and the object to measure can be described by different measurement setups. Figure 6.3 shows the three apparent configurations for the three components. In the first case, the camera view is orthogonal to the object and the light plane is tilted. The second case shows a tilted camera view and an orthogonal light plane. For the third case, both the camera view and the laser line are tilted.

Figure 6.4 exemplarily shows a measurement setup as it is used for the examples that will be discussed in the following sections.

Which measurement configuration to use depends on the goal of the measurement and the geometry of the object. The configuration in figure 6.3 (a), e.g., is especially suitable if an orthogonal projection of the object in the image is needed for any reason. Then, a cuboid is imaged as a rectangle. For all configurations in which the camera is

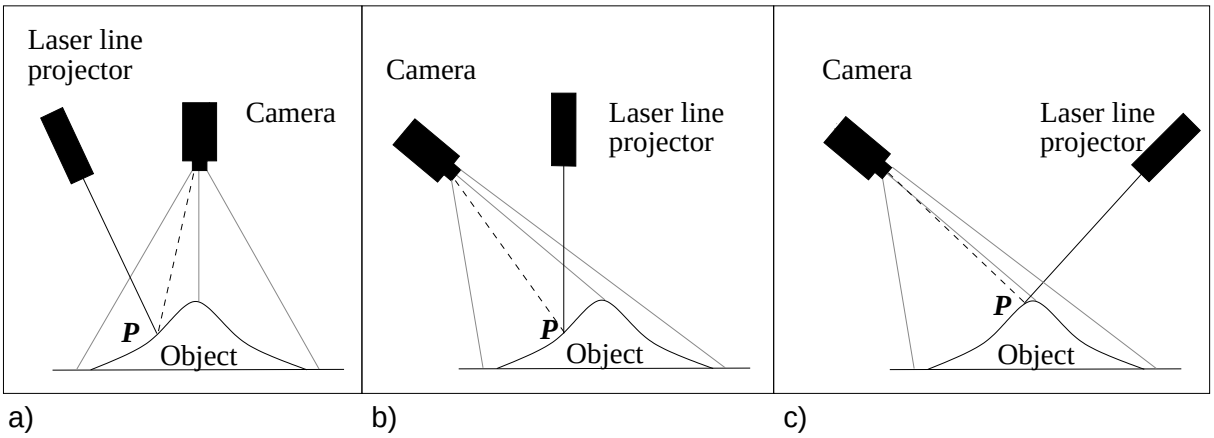


Figure 6.3: Basic configurations possible for a sheet-of-light measurement setup.

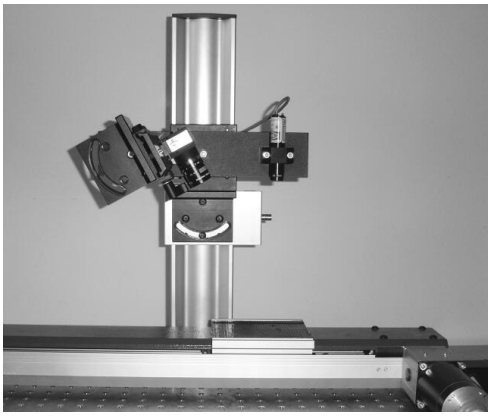


Figure 6.4: Exemplary setup for a sheet-of-light measurement consisting of a camera, a laser line projector, and a positioning system.

not placed orthogonal, it would be imaged as a trapezoid because of the perspective deformations (see figure 6.5 on page 150).

The most important criterion for the selection of the measurement setup is the geometry of the object. The setup should be selected such that the amount of shadowing effects and occlusions is minimized. Occlusions occur if an object point is illuminated by the laser line but is not visible in the image, because other parts of the object lie between the line of sight of the camera and the object point (see figure 6.6, top). Shadowing effects occur if an object point is visible in the image but is not illuminated by the laser line, because other parts of the object lie between the laser projection and the imaged object point (see figure 6.6, bottom).

For all three setup configurations, the angle of triangulation, i.e., the angle between the light plane and the optical axis of the camera, should be in a range of 30° to 60° to get a good measurement accuracy. If the angle is smaller, the accuracy decreases. If the angle is larger, the accuracy increases, but more problems because of occlusions and shadowing effects are to be expected. Thus, you have to find a trade-off between the accuracy and the completeness of the measurement.

Additionally, the accuracy decreases if the light plane is out of focus of the camera. That may happen, if the angle of triangulation is too small, i.e., the angle between the light plane and the focus plane is too large, and therefore only a small part near the intersection of both planes is in focus. To overcome this problem, a tilt lens can be used to align the focus plane with the light plane using the Scheimpflug principle (see section 2.2.3 and section 5.1.3 for details).

6.3 Calibrating the Sheet-of-Light Setup

A sheet-of-light setup can be calibrated in two different ways:

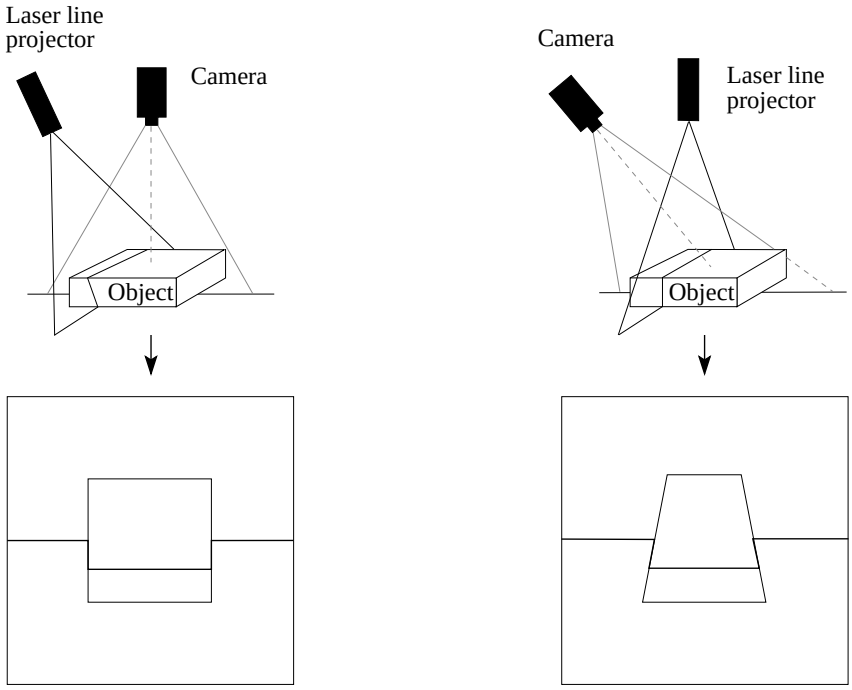


Figure 6.5: With the camera being orthogonal to the object, an orthogonal projection of the object is possible: (left) orthogonal camera view, (right) perspective view.

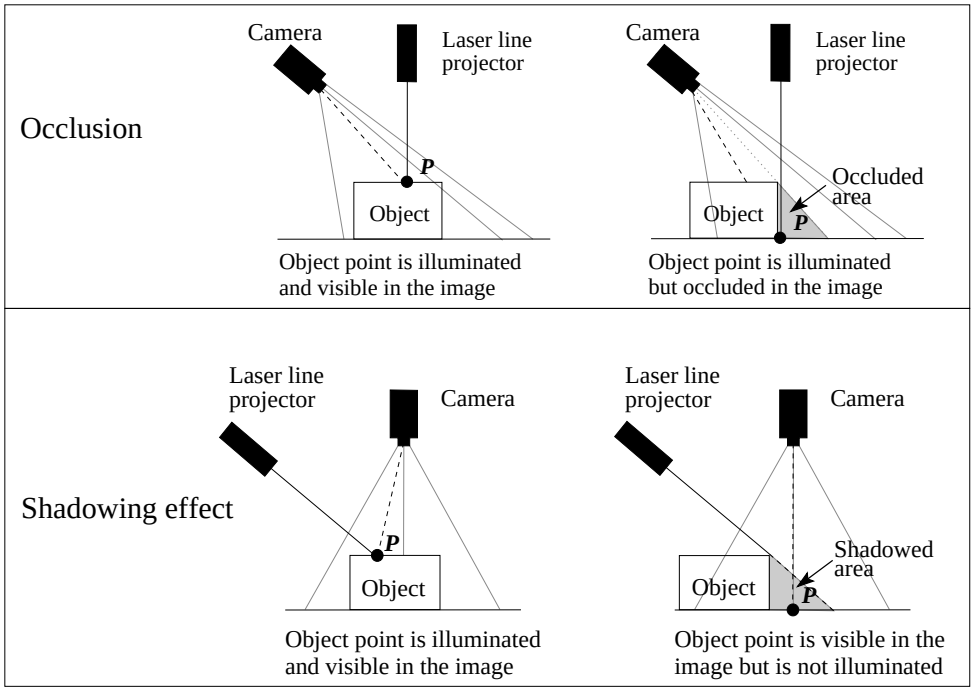


Figure 6.6: Problems that have to be considered before selecting the measurement setup: (top) occlusions and (bottom) shadowing effects.

Calibration using a standard HALCON calibration plate

To calibrate a sheet-of-light setup using a standard HALCON calibration plate (see section 6.3.1), first the camera is calibrated conventionally. Then, the pose of the light plane and the movement of the objects to be measured must be determined based on additional images of the calibration plate.

Calibration using a special 3D calibration object

For the calibration of a sheet-of-light setup with a special 3D calibration object (see section 6.3.2), first the 3D calibration object must be provided. The calibration itself requires only one (uncalibrated) reconstruction of the 3D calibration object, i.e., its disparity image.

6.3.1 Calibrating the Sheet-of-Light Setup using a standard HALCON calibration plate

This section describes how to calibrate the sheet-of-light measurement setup. If the uncalibrated case is sufficient for your application, you can skip this section and proceed with section 6.4 on page 157.

The calibration of the sheet-of-light setup is applied to get the internal and external camera parameters, the orientation of the light plane in the WCS, and the relative movement of the object between two successive measurements. The calibration consists of the following steps:

1. Calibrate the camera.
2. Determine the orientation of the light plane with respect to the WCS.
3. Calibrate the movement of the object relative to the measurement setup.

The camera is calibrated by a standard camera calibration as described in section 3.2 on page 61. As result, the camera calibration returns the internal camera parameters and the pose of the WCS relative to the camera coordinate system (external camera parameters).

To determine the light plane and its pose, we need at least three corresponding points (see figure 6.7), in particular two points in the plane of the WCS with `'z=0'` (`P1`, `P2`) and one point that differs significantly in z direction (`P3`). Thus, you place the calibration object, e.g., the standard HALCON calibration plate, once or twice so that it lies in the plane of the WCS with `'z=0'`, and once so that a higher position can be viewed, i.e., the plate is either translated in z direction or it is placed in a tilted position. For each position of the calibration plate, you take two images, one showing the calibration plate and one showing the laser line. Note that you have to adapt the lighting in between to get one image with a clearly represented calibration plate and one image that shows a well-defined laser line. The translated or tilted position of the calibration plate should be selected so that the plane that is built by the points `P1`, `P2`, and `P3` becomes as large as possible. The height difference should be at least as big as the height differences expected for the objects to measure.

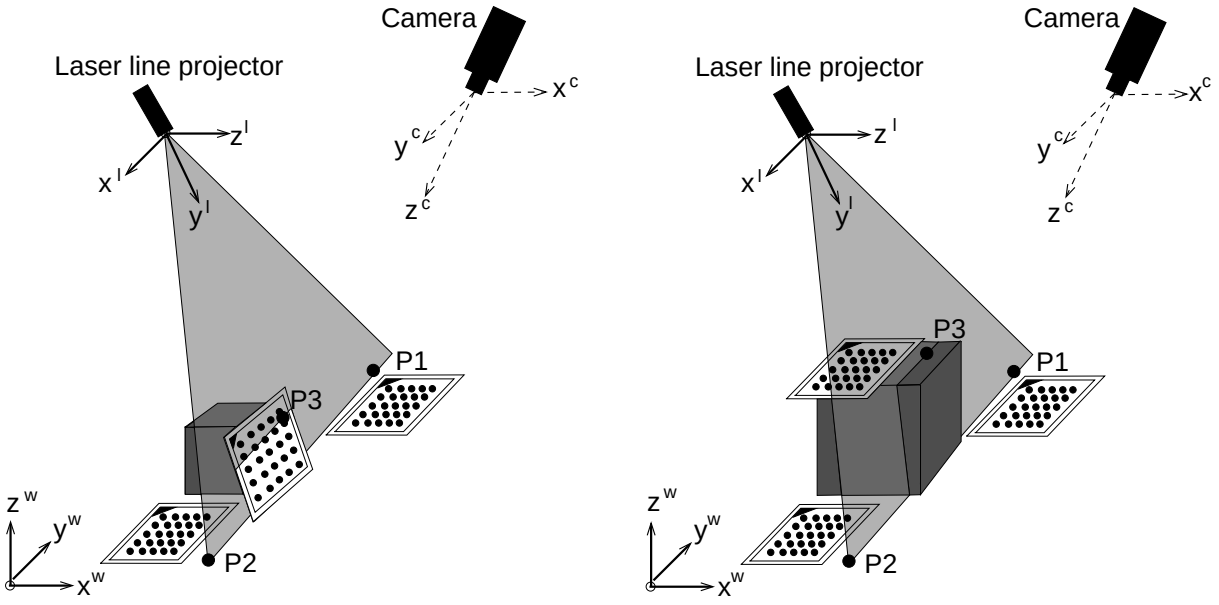


Figure 6.7: Position of the third point (P3) obtained by (left) tilted calibration plate or (right) translated calibration plate.

Note that the laser line has to be projected onto the same plane in which the calibration plate is placed. But if possible, it should not be directly projected onto the calibration plate but only near to it. This is because the used standard HALCON calibration plate is made of a ceramic that shows a strong volume scattering. This leads to a broadened profile (see figure 6.8), which results in a poor accuracy. If you use a calibration object that consists of a material with different reflection properties, this might be no problem.

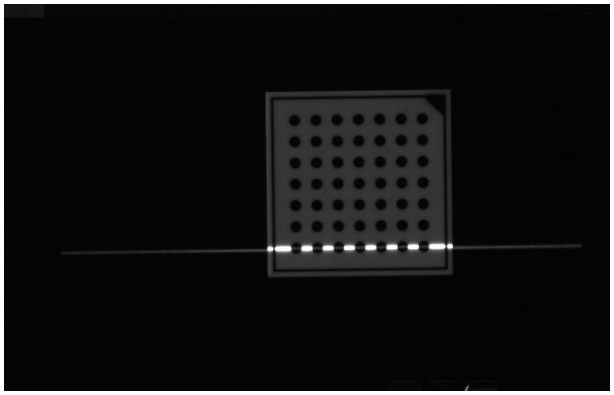


Figure 6.8: The white parts of the used HALCON calibration plate show a very broad laser line because of volume scattering.

Note further that the three corresponding points described above represent the minimum number of points needed to obtain a plane. To enhance the precision of the calibration, redundancy is needed; thus, we recommended to measure more than three corresponding points. Then, the light plane is approximated by fitting a plane into the obtained point cloud.

In the final step, the pose describing the movement of the object must be calibrated using two images containing a calibration plate that was moved by the positioning system by a known number of movement steps.

Summarized, we have to acquire:

- a set of images for the camera calibration,
- at least two images that clearly show the laser line in different planes and which correspond to images that were used for the calibration, and
- at least two images that show the calibration plate in the plane of the linear positioning system. Between the acquisition of the first and the second image, the calibration plate must be moved by the positioning system by a known number of movement steps.

The HDevelop example program
`%HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\calibrate_sheet_of_light_calplate.hdev`
shows in detail how to calibrate a sheet-of-light measurement setup with a standard HALCON calibration plate.

For the first step, i.e., the camera calibration, initial values for the internal camera parameters and for the thickness of the calibration plate are set.

```
gen_cam_par_area_scan_polynomial (0.0125, 0.0, 0.0, 0.0, 0.0, 0.0, 0.000006, \
                                  0.000006, 376.0, 120.0, 752, 240, \
                                  StartParameters)
CalTabDescription := 'caltab_30mm.descr'
* Note that the thickness of the calibration target used for this example  \
* is 0.63 mm.
* If you adapt this example program to your application, it is necessary  \
* to determine
* the thickness of your specific calibration target and to use this value  \
* instead.
CalTabThickness := .00063
```

Then, the calibration images are read. These should fulfill the requirements that are described for a camera calibration in the section “How to take a set of suitable images?” in the chapter reference “Calibration”. Now, for

each image, the calibration plates are searched, the contours and centers of their marks are extracted, and the pose of the calibration plate is estimated. The obtained information is stored in the calibration data model.

```
NumCalibImages := 20
for Index := 1 to NumCalibImages by 1
    read_image (Image, 'sheet_of_light/connection_rod_calib_' + Index$'.2')
    find_calib_object (Image, CalibDataID, 0, 0, Index, [], [])
endfor
```

With the obtained data, the actual camera calibration is performed, so that the internal camera parameters (`CameraParameters`) and the external camera parameters (camera poses) for all calibration images can be obtained. The internal camera parameters and the camera pose for one of the calibration images are the first two variables that we need for the sheet-of-light measurement that is described in the next section.

```
calibrate_cameras (CalibDataID, Errors)
get_calib_data (CalibDataID, 'camera', 0, 'params', CameraParameters)
```

Note that by selecting the camera pose of one of the calibration images you define the origin of the WCS used for the measurement.

For the second step, i.e., the orientation of the light plane in relation to the WCS, the poses of two of the calibration images are needed. The images show the calibration plates in different heights. The pose of one image is used to define the WCS and the pose of the other image is used to define a temporary coordinate system (TCS). For both images, the origins of the poses are shifted with `set_origin_pose` to consider the thickness of the calibration plate.

```
Index := 19
get_calib_data (CalibDataID, 'calib_obj_pose', [0,Index], 'pose', \
                CalTabPose)
set_origin_pose (CalTabPose, 0.0, 0.0, CalTabThickness, CameraPose)
Index := 20
get_calib_data (CalibDataID, 'calib_obj_pose', [0,Index], 'pose', \
                CalTabPose)
set_origin_pose (CalTabPose, 0.0, 0.0, CalTabThickness, TmpCameraPose)
```

For each of the two calibration images a corresponding laser line image was acquired. There, the laser line is clearly projected onto the same plane that contained the calibration plate in the calibration image. With the two laser line images and the poses obtained from the two corresponding calibration images the procedure `compute_3d_coordinates_of_light_line` calculates the 3D coordinates of the points that build the laser lines. The obtained point cloud consists of the points of the light plane in the plane of the WCS with `'z=0'` (see `P1` and `P2` in figure 6.7 on page 151) and the points of the light plane in the plane of the TCS with `'z=0'` (see `P3` in figure 6.7 on page 151).

```
read_image (ProfileImage1, \
            'sheet_of_light/connection_rod_lightline_019.png')
compute_3d_coordinates_of_light_line (ProfileImage1, MinThreshold, \
                                      CameraParameters, [], CameraPose, \
                                      X19, Y19, Z19)
read_image (ProfileImage2, \
            'sheet_of_light/connection_rod_lightline_020.png')
compute_3d_coordinates_of_light_line (ProfileImage2, MinThreshold, \
                                      CameraParameters, TmpCameraPose, \
                                      CameraPose, X20, Y20, Z20)
```

Now, the procedure `fit_3d_plane_xyz` fits a plane into the point cloud. This plane is the light plane, for which the pose is needed as the third variable for the calibrated sheet-of-light measurement. This pose (`LightPlanePose`) is calculated from the plane using the procedure `get_light_plane_pose`.

```
procedure fit_3d_plane_xyz (X, Y, Z, Ox, Oy, Oz, Nx, Ny, Nz, MeanResidual)
get_light_plane_pose (Ox, Oy, Oz, Nx, Ny, Nz, LightPlanePose)
```

In the third step, i.e., the calibration of the movement of the object in relation to the measurement setup, the calibration plate is moved in discrete steps by the linear positioning system that will be used also for the following measurement. To calibrate the movement of the linear positioning system, two images with different movement states are needed. To enhance the accuracy, we do not use images of two succeeding movement steps but use images with a known number of movement steps between them. Here, the number of movement steps between both images is 19.

```
read_image (CaltabImagePos1, 'sheet_of_light/caltab_at_position_1.png')
read_image (CaltabImagePos20, 'sheet_of_light/caltab_at_position_2.png')
StepNumber := 19
```

Now, for both images the poses of the calibration plates are derived.

```
find_calib_object (CaltabImagePos1, CalibDataID, 0, 0, NumCalibImages + 1, \
                   [], [])
get_calib_data_observ_points (CalibDataID, 0, 0, NumCalibImages + 1, Row1, \
                              Column1, Index1, CameraPosePos1)
find_calib_object (CaltabImagePos20, CalibDataID, 0, 0, NumCalibImages + 2, \
                   [], [])
get_calib_data_observ_points (CalibDataID, 0, 0, NumCalibImages + 2, Row1, \
                              Column1, Index1, CameraPosePos20)
```

Then, the pose that describes the transformation between these two poses, i.e., the transformation needed for 19 movement steps, is calculated (`MovementPoseNSteps`). Note that a rotation is not assumed and therefore all rotational elements are set to 0.

```
pose_to_hom_mat3d (CameraPosePos1, HomMat3DPos1ToCamera)
pose_to_hom_mat3d (CameraPosePos20, HomMat3DPos20ToCamera)
pose_to_hom_mat3d (CameraPose, HomMat3DWorldToCamera)
hom_mat3d_invert (HomMat3DWorldToCamera, HomMat3DCameraToWorld)
hom_mat3d_compose (HomMat3DCameraToWorld, HomMat3DPos1ToCamera, \
                   HomMat3DPos1ToWorld)
hom_mat3d_compose (HomMat3DCameraToWorld, HomMat3DPos20ToCamera, \
                   HomMat3DPos20ToWorld)
affine_trans_point_3d (HomMat3DPos1ToWorld, 0, 0, 0, StartX, StartY, StartZ)
affine_trans_point_3d (HomMat3DPos20ToWorld, 0, 0, 0, EndX, EndY, EndZ)
create_pose (EndX - StartX, EndY - StartY, EndZ - StartZ, 0, 0, 0, 'Rp+T', \
             'gba', 'point', MovementPoseNSteps)
```

To get the pose for a single movement step (`MovementPose`), the elements of `MovementPoseNSteps` that describe a translation are divided by the number of steps. `MovementPose`, together with the internal and external camera parameters and the pose of the light plane can now be used to apply a calibrated sheet-of-light measurement.

```
MovementPose := MovementPoseNSteps / StepNumber
```

For details about the proceedings inside the stated procedures, we refer to the example.

6.3.2 Calibrating the Sheet-of-Light Setup Using a Special 3D Calibration Object

Figure 6.9 shows a sheet-of-light setup together with a special 3D calibration object. To calibrate the sheet-of-light setup, one disparity image of the 3D calibration object is acquired with the sheet-of-light setup. The sheet-of-light setup is then calibrated using this disparity image with the operator `calibrate_sheet_of_light`.

The calibration of a sheet-of-light setup with a 3D calibration object is simpler but slightly less accurate than the calibration of a sheet-of-light setup with a standard HALCON calibration plate, which is described in section 6.3.1 on page 151. Nevertheless, first a suitable 3D calibration object must be provided.

In the following, the steps that are necessary for the calibration are described. The HDevelop example program `%HALCONEXAMPLES%\hdevelop\3D-Reconstruction\Sheet-Of-Light\calibrate_sheet_of_light_3d_calib_object.hdev` shows in detail how to calibrate a sheet-of-light setup with a special 3D calibration object.

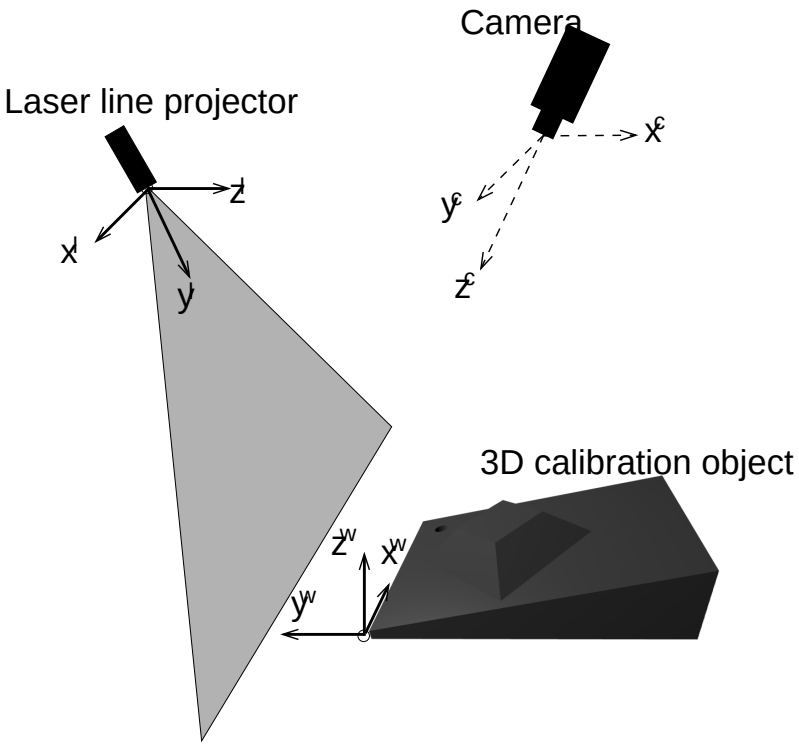


Figure 6.9: Sheet-of-light setup with 3D calibration object.

Supply of a 3D Calibration Object

A special 3D calibration object must be provided. This calibration object must correspond to the CAD model created with `create_sheet_of_light_calib_object`. The 3D calibration object has an inclined plane on which a truncated pyramid is located. It has a thinner side, which is hereinafter referred to as front side. The thicker side is referred to as back side of the calibration object. Figure 6.10 shows the 3D calibration object.

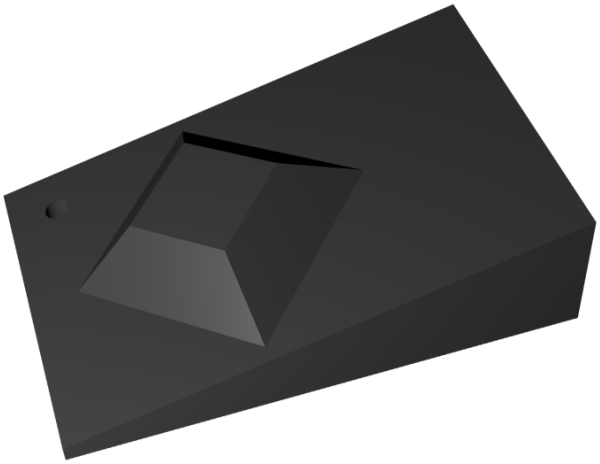


Figure 6.10: 3D calibration object.

The dimensions of the calibration object should be chosen such that the calibration object covers the complete measuring volume. Be aware, that only parts on the 3d calibration object above the minimum height of the tilted plane (see the parameter `HeightMin` of the operator `create_sheet_of_light_calib_object`) are taken into account.

The CAD model, which is written as a DXF file by `create_sheet_of_light_calib_object`, also serves as description file of the calibration object. This CAD model can then be used to manufacture the individual calibration

object. Note that MVTec does not offer such 3D calibration objects. More information on how to manufacture a calibration object is provided by the description of create_sheet_of_light_calib_object.

Preparation of the Sheet-Of-Light Model

To prepare a sheet-of-light model for the calibration, the following steps must be performed.

1. Create a sheet-of-light model with create_sheet_of_light_model.
2. Set the initial parameters of the camera with set_sheet_of_light_param. So far, only pinhole cameras with the division model are supported.
3. Set the description file of the calibration object (created with create_sheet_of_light_calib_object) with set_sheet_of_light_param.

```
create_sheet_of_light_model (Domain, [], [], SheetOfLightModelID)
set_sheet_of_light_param (SheetOfLightModelID, 'camera_parameter', \
                          CameraParam)
set_sheet_of_light_param (SheetOfLightModelID, 'calibration_object', \
                          'calib_object.dxf')
```

Uncalibrated Reconstruction of the 3D Calibration Object

The 3D calibration object must be reconstructed with the (uncalibrated) sheet-of-light model prepared above, i.e., a disparity image of the 3D calibration object must be created (see figure 6.11).

```
for Index := 1 to 1000 by 1
    measure_profile_sheet_of_light (ProfileImage, SheetOfLightModelID, [])
endfor
```

Figure 6.11: Disparity image of the 3D calibration object.

For this, the calibration object must be oriented such that either its front side or its back side intersect the light plane first (i.e., the movement vector should be parallel to the Y axis of the calibration object, see figure 6.9 on page 155 or create_sheet_of_light_calib_object). As far as possible, the domain of the disparity image of the calibration object should be restricted to the calibration object. Besides, the domain of the disparity image should have no holes on the truncated pyramid. All four sides of the truncated pyramid must be clearly visible.

If the disparity image is already provided by the camera, it can be set with set_sheet_of_light_param directly.

```
set_profile_sheet_of_light (CalibObjectDisparity, SheetOfLightModelID, [])
```

Calibration of the Sheet-Of-Light Setup

The calibration is then performed with calibrate_sheet_of_light.

```
calibrate_sheet_of_light (SheetOfLightModelID, Error)
```

The returned error is the RMS of the distance of the reconstructed points to the calibration object in meters.

For sheet-of-light models calibrated with calibrate_sheet_of_light, you can obtain the calibrated camera parameters, camera pose, light plane pose, and movement pose using get_sheet_of_light_param.

6.4 Performing the Measurement

A sheet-of-light measurement is applied to get height information for the object to measure. This height information is presented by a disparity image in which each row contains the disparities of one measured profile of the object (see figure 6.2 on page 148), by the images X, Y, and Z that express the x, y, and z coordinates of the measured profiles as values of pixels within images, or by a 3D object model that contains the coordinates of the object's 3D points and the corresponding 2D mapping. The images X, Y, and Z as well as the 3D object model can be obtained only for a calibrated measurement setup, whereas the disparity image can be obtained also for the uncalibrated case. A sheet-of-light measurement consists of the following basic steps:

1. Calibrate the measurement setup (if a calibrated measurement is needed) as described in the previous section.
2. Create a sheet-of-light model with create_sheet_of_light_model and set additional parameters with successive calls to set_sheet_of_light_param.
3. Acquire images for each profile to measure, e.g., using grab_image_async.
4. Measure the profile of each image with measure_profile_sheet_of_light.
5. Get the results of the measurement with successive calls to get_sheet_of_light_result or, if a 3D object model is required, with a single call to get_sheet_of_light_result_object_model_3d.
6. If only the uncalibrated case was applied and a disparity image was obtained, but the x, y, and z coordinates or the 3D object model are still needed, you can subsequently apply a calibration. Then, you have to calibrate the measurement setup like described in the previous section and add the obtained camera parameters to the model with set_sheet_of_light_param. Afterwards you call the operator apply_sheet_of_light_calibration with the disparity image and the adapted sheet-of-light model. The resulting images that contain the coordinates X, Y, and Z or the 3D object model are queried from the model with get_sheet_of_light_result or get_sheet_of_light_result_object_model_3d, respectively. How to subsequently apply a sheet-of-light calibration to a disparity image is shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\calibrate_sheet_of_light_calplate.hdev.

Optionally, you can query all parameters that you have already set for a specific model or that were set by default using get_sheet_of_light_param. To query all parameters that can be set for a sheet-of-light model you call query_sheet_of_light_params.

6.4.1 Calibrated Sheet-of-Light Measurement

How to apply a calibrated sheet-of-light measurement is shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\reconstruct_connection_rod_calib.hdev, which measures the object shown in figure 6.12.

Figure 6.12: Object to measure.

The first step is to assign the information obtained by the calibration of the sheet-of-light measurement setup (see previous section) to a set of variables.

```
gen_cam_par_area_scan_polynomial (0.0126514, 640.275, -2.07143e+007, \
                                  3.18867e+011, -0.0895689, 0.0231197, \
                                  6.00051e-006, 6e-006, 387.036, 120.112, \
                                  752, 240, CamParam)
create_pose (-0.00164029, 1.91372e-006, 0.300135, 0.575347, 0.587877, \
             180.026, 'Rp+T', 'gba', 'point', CamPose)
create_pose (0.00270989, -0.00548841, 0.00843714, 66.9928, 359.72, 0.659384, \
             'Rp+T', 'gba', 'point', LightplanePose)
create_pose (7.86235e-008, 0.000120112, 1.9745e-006, 0, 0, 0, 'Rp+T', 'gba', \
             'point', MovementPose)
```

Then, a sheet-of-light model is created for a rectangular region of interest using `create_sheet_of_light_model`. The ROI should be selected as large as necessary but as small as possible. That is, it should approximately be some pixels larger than the width of the object in width and the maximal expected displacement of the laser line caused by the height of the object, i.e., the largest expected disparity, in height.

Now, some parameters are set with `set_sheet_of_light_param`. As a calibrated measurement is applied, the parameter `'calibration'` is set to `'xyz'`. For an uncalibrated measurement, it would be `'none'`, which is the default. Further, the variables with the calibration information are passed as values to the corresponding parameters for the internal camera parameters (`'camera_parameter'`), the external camera parameters (`'camera_pose'`), the pose of the light plane (`'lightplane_pose'`), and the movement of the object relative to the measurement setup (`'movement_pose'`).

```
gen_rectangle1 (ProfileRegion, 120, 75, 195, 710)
create_sheet_of_light_model (ProfileRegion, ['min_gray', 'num_profiles', \
                             'ambiguity_solving'], [70, 290, 'first'], \
                             SheetOfLightModelID)
set_sheet_of_light_param (SheetOfLightModelID, 'calibration', 'xyz')
set_sheet_of_light_param (SheetOfLightModelID, 'scale', 'mm')
set_sheet_of_light_param (SheetOfLightModelID, 'camera_parameter', CamParam)
set_sheet_of_light_param (SheetOfLightModelID, 'camera_pose', CamPose)
set_sheet_of_light_param (SheetOfLightModelID, 'lightplane_pose', \
                          LightplanePose)
set_sheet_of_light_param (SheetOfLightModelID, 'movement_pose', \
                          MovementPose)
```

Then, for each profile to measure an image is acquired (see, e.g., figure 6.13) to apply the actual measurement. Here, the images for each movement step are read from file with `read_image`. In practice, you will most probably grab the images directly from your image acquisition device using `grab_image_async` (see Solution Guide II-A for details about image acquisition). For each image, the profile within the rectangular region of interest is

measured with measure_profile_sheet_of_light, i.e., the disparities for the profile are determined and stored in the sheet-of-light model.

```
for Index := 1 to 290 by 1
    read_image (ProfileImage, 'sheet_of_light/connection_rod_' + Index$'.3')
    measure_profile_sheet_of_light (ProfileImage, SheetOfLightModelID, [])
endfor
```

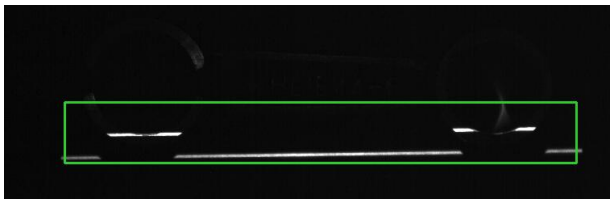


Figure 6.13: Measure profile inside a rectangular ROI.

The default for the number of profiles to measure is 512. You can change it with the parameter 'num_profiles' within create_sheet_of_light_model or set_sheet_of_light_param. If you measure more than the specified number of profiles, the value of 'num_profiles' is automatically adapted in the model. Nevertheless, this adaptation requires additional runtime. Thus, we recommend to set 'num_profiles' to a suitable value before starting the measurement. Note that the number of measured profiles defines the number of rows and the width of the ROI used for the measurement defines the number of columns for the result images (i.e., the disparity image, the X, Y, and Z images, and the score image).

After all measurements were performed, the results of the sheet-of-light measurement are queried with calls to get_sheet_of_light_result and get_sheet_of_light_result_object_model_3d. Here, we query the disparity image (ResultName set to 'disparity'), the images X, Y, and Z (ResultName set to 'x', 'y', and 'z', respectively), and the 3D object model. The images X, Y, and Z are shown in figure 6.14. The 3D object model is interactively displayed using the procedure visualize_object_model_3d as shown in figure 6.15.

The interpretation of the gray values of the disparity image and the images X, Y, and Z is as follows: black parts are outside of the domain of the resulting image, i.e., they indicate parts for which no 3D information could be reconstructed. For the pixels inside the domain of the image bright parts show low object parts and dark parts show higher object parts. Note that in this example the images are not visualized by their default gray values but are converted additionally by a look-up table so that the images are colored. This is done because the human eye can separate much more colors than gray values. Thus, details can be better distinguished during a visual inspection.

```
get_sheet_of_light_result (Disparity, SheetOfLightModelID, 'disparity')
get_sheet_of_light_result (X, SheetOfLightModelID, 'x')
get_sheet_of_light_result (Y, SheetOfLightModelID, 'y')
get_sheet_of_light_result (Z, SheetOfLightModelID, 'z')
get_sheet_of_light_result_object_model_3d (SheetOfLightModelID, \
                                           ObjectModel3DID)
```


6.4.2 Uncalibrated Sheet-of-Light Measurement

The uncalibrated sheet-of-light measurement is shown in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\reconstruct_connection_rod_uncalib.hdev. Here, no calibration results are needed, so we simply create the model for the specified region of interest and set the few needed parameters directly within create_sheet_of_light_model.

```
gen_rectangle1 (ProfileRegion, 120, 75, 195, 710)
create_sheet_of_light_model (ProfileRegion, ['min_gray', 'num_profiles', \
                             'ambiguity_solving', 'score_type'], [70, 290, \
                             'first', 'width'], SheetOfLightModelID)
```

The actual measurement is applied by the same process used for the calibrated measurement.

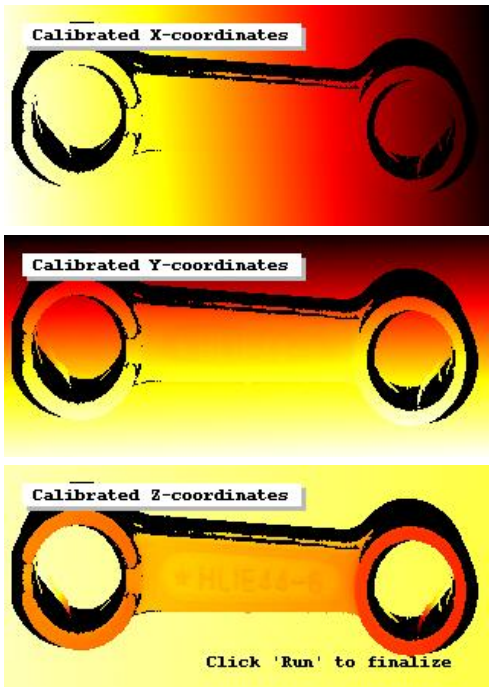


Figure 6.14: Result of calibrated sheet-of-light measurement: images representing the (from top to bottom) x, y, and z coordinates of the object.

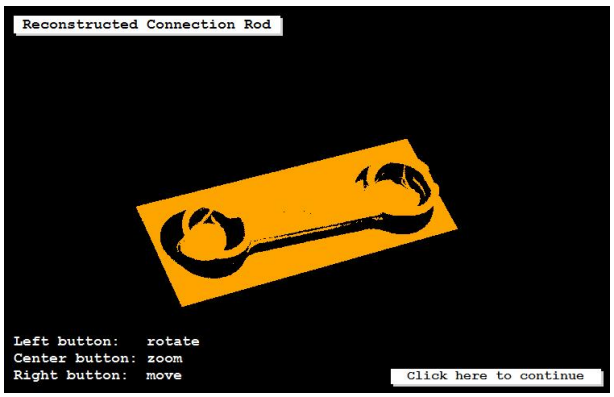


Figure 6.15: Result of calibrated sheet-of-light measurement: 3D object model.

```
for Index := 1 to 290 by 1
    read_image (ProfileImage, 'sheet_of_light/connection_rod_' + Index$'.3')
    measure_profile_sheet_of_light (ProfileImage, SheetOfLightModelID, [])
endfor
```

As result, we can only query the disparity image (see figure 6.16) and the score (see section 6.5) of the measurement (`ResultName` set to `'score'`).

```
get_sheet_of_light_result (Disparity, SheetOfLightModelID, 'disparity')
get_sheet_of_light_result (Score, SheetOfLightModelID, 'score')
```

6.5 Using the Score Image

Caused by the specific characteristics of a laser line projector and the general principle of triangulation the results of a sheet-of-light measurement, i.e., the disparities or the calibrated coordinates, sometimes show disturbing

Figure 6.16: Result of uncalibrated sheet-of-light measurement: disparity image.

artifacts. The score image can be used to detect and partially remove artifacts.

There are two types of artifacts. The first type is caused by the geometry of the surface that is to be reconstructed. As illustrated in figure 6.17, compared to flat and smooth surfaces (e.g., the object in figure 6.16), curved surfaces with a small radius of curvature and surfaces with a significant slope lead to a broadening of the light line. Furthermore, the light distribution within the profile might be no longer symmetric, which leads to a reduced measurement accuracy.

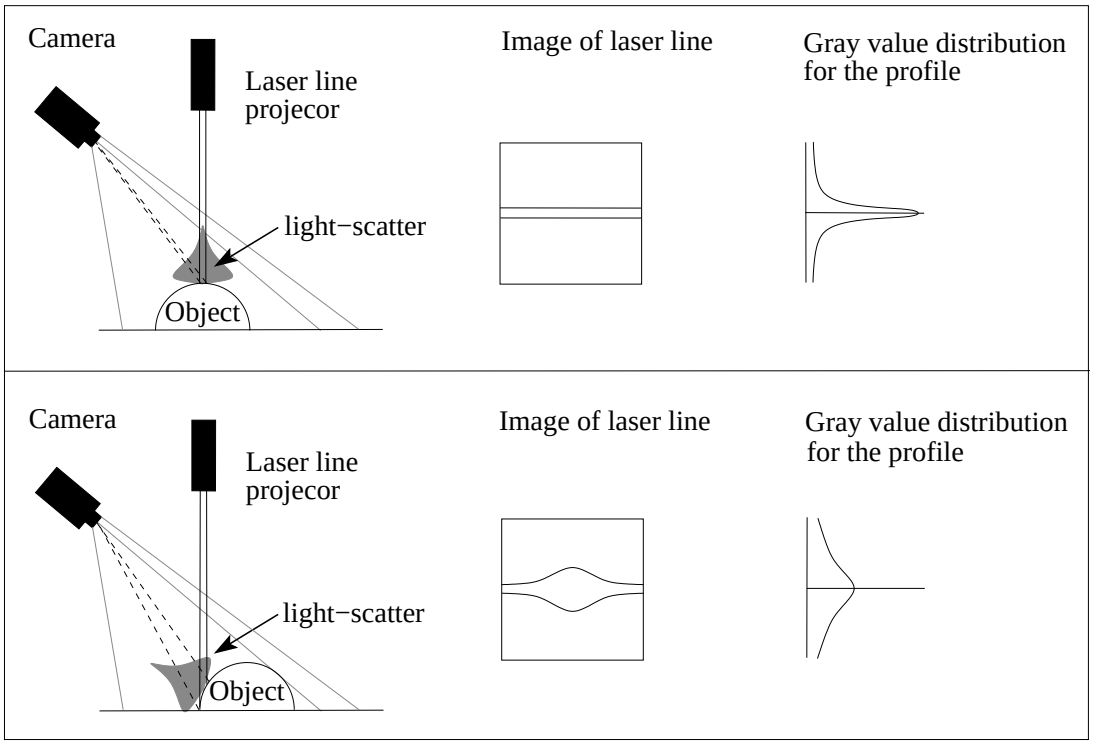


Figure 6.17: Curved surfaces with a small radius of curvature and surfaces with a significant slope lead to a broadening of the light line and thus to a low score: (top) small influence of curvature, (bottom) large influence of curvature.

By using the width of the profile stored in the score image (for each pixel of the disparity, the score value is set to the number of pixels used to determine the disparity value) it is possible to detect artifacts and to reject the corresponding disparities or the corresponding calibrated coordinates. Figure 6.18 shows the score image obtained by the uncalibrated sheet-of-light measurement performed in the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\reconstruct_connection_rod_uncalib.hdev`. The gray values inside the score image indicate the widths of the laser line in each pixel. Thus, artifacts, in this case parts with a significantly broadened laser line, can be recognized easily by their brightness.

In the example, the artifacts are rejected from the disparity image by applying a threshold to the score image (pixels with a value larger than `7.5` are rejected) and reducing the disparity image to the obtained region.

Figure 6.18: Result of uncalibrated sheet-of-light measurement: score image (`score_type` set to `'width'`, i.e., bright parts indicate artifacts).

```
threshold (Score, ScoreRegion, 1.5, 7.5)
reduce_domain (Disparity, ScoreRegion, DisparityReduced)
```

The second type of artifacts is caused by the interaction of the coherent laser light with the surface of the object. Laser light produces disturbing interference patterns when it is projected on a rough textured surface. Those interference patterns are called speckle and can be considered as a non-additive noise, which means that this noise can not be reduced by averaging during the image acquisition. In this case, the only way to increase the measurement accuracy is to use a higher aperture for the image acquisition or a low-speckle line projector. Note that enlarging the aperture for the image acquisition device will also reduce the depth of field which might be an undesired side effect. If your application requires high accuracy, we strongly recommend to use low-speckle projection devices. Note that speckle in most cases is the limiting factor for the accuracy of laser triangulation systems.

6.6 3D Cameras for Sheet of Light

The proceeding described in the previous sections works for any standard 2D camera that is suitable for machine vision. An alternative is to use specific 3D cameras for which the sheet-of-light measurement is applied inside the camera. These cameras are more expensive than standard 2D cameras, but the sheet-of-light measurement becomes significantly faster because of the reduced CPU load. Using one of these cameras, you simply access the camera with HALCON and basically leave the measurement to the camera.

Generally, we distinguish between cameras with an inbuilt laser, i.e., the camera and the laser are integrated in a single unit, and cameras for which the laser is mounted separately.

If the camera and the laser are integrated in a single unit, the measurement setup is restricted to a fixed angle of triangulation and should be oriented in a defined way. For example, the SICK Ruler camera should be oriented so that the laser is perpendicular to the linear positioning system. Because of the preset measurement setup, the camera and the orientation of the light plane with respect to the world coordinate system are already calibrated. Thus, no further processing with HALCON is needed to obtain calibrated height profiles.

If the camera and the laser are mounted separately, any configuration of the measurement setup is possible (see section 6.2 on page 147), but by default the result of the measurement is uncalibrated. If the result of the measurement is needed in world coordinates, you can either query the uncalibrated data from the camera and subsequently apply a calibration with HALCON as described in section 6.4 on page 157, or, before performing the actual measurement, you apply a calibration that is provided specifically for the selected camera. For the SICK Ranger cameras, e.g., the camera-specific calibration needs the software provided with the camera (the SICK Coordinator tool) and a specific calibration object that has to be purchased separately.

Note that in contrast to the proceeding described in the previous sections, the movement of the linear positioning system is mostly assumed to be known, because the measurement of each profile is triggered by a signal coming from an encoder on the linear positioning system. That is, when working with an encoder and if the thus obtained accuracy is sufficient, it is not necessary to calibrate the distance between two profiles.

Chapter 7

Depth from Focus

Depth from focus (DFF) is a method that enables the reconstruction of 3D surface information from several images taken at different focus distances between camera and object. It allows a highly accurate non-destructive 3D measurement of surfaces. The example shown in figure 7.1 was done using microscopic optics with 10x magnification and reaches an accuracy of about 5 micrometers. DFF is even more precise than the methods stereo chapter 5 on page 117 and laser triangulation with sheet of light chapter 6 on page 147. Furthermore, the setup requires only a single camera, therefore, it is possibly more compact than, e.g., a stereo setup. DFF requires, however, cameras with telecentric or microscope lenses in order to achieve a (nearly) parallel projection. Therefore, DFF is only suitable for small objects. Examples for suitable objects in semiconductors industry are a ball grid array (BGA) (the result of DFF on a single ball of a BGA can be viewed in figure 7.1) or solder paste inspection, another application in the engineering sector is the inspection of indexable inserts.

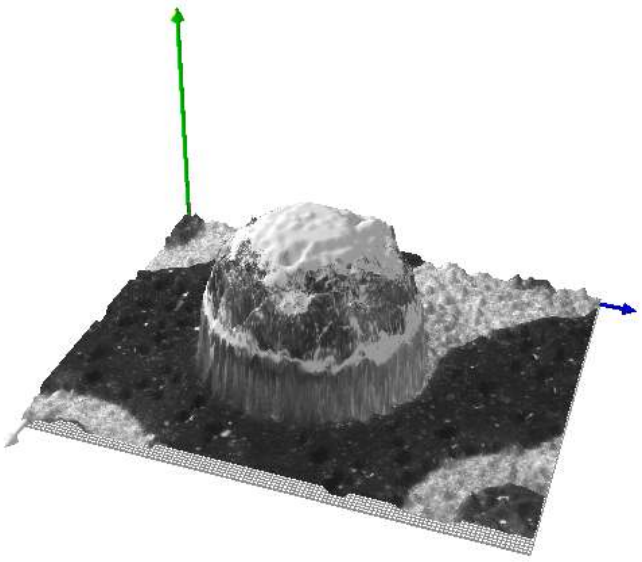


Figure 7.1: This image shows the 3D surface reconstruction of a single ball on a ball grid array.

7.1 The Principle of Depth from Focus

With depth from focus, you can reconstruct the surface of a 3D object based on the knowledge that object points have different distances to the camera and the camera has a limited depth of field. Depending on the distance and the focus, object points are displayed more or less sharply in the image, i.e., only those pixels within the correct distance to the camera are focused. Taking images with various object distances, each object point can be displayed sharply in at least one image. Such a sequence of images is called “focus stack”. By determining in which image an object point is in focus, i.e., sharply imaged, the distance of each object point to the camera can be calculated. This principle is clarified in figure 7.2.

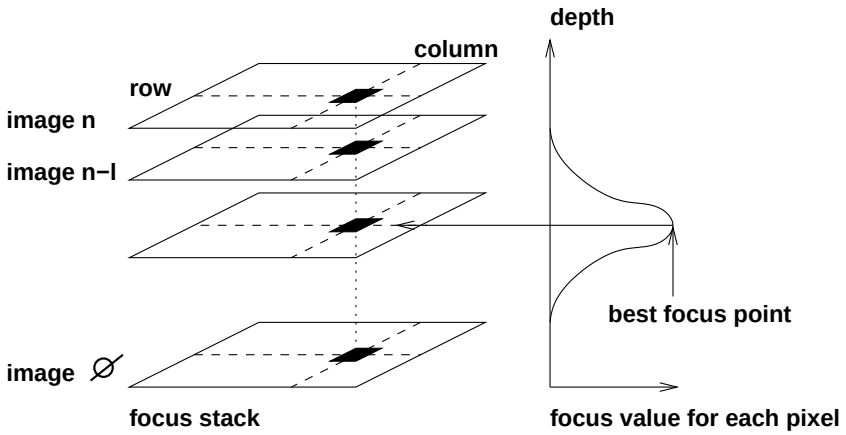


Figure 7.2: This figure shows the focus stack of images on the left side and the corresponding focus value - that is determined for each pixel - on the right side. The best focus point is the image where a pixel has the highest sharpness.

For more information about determining sharpness, please refer to `%HALCONEXAMPLES%\hdevelop\Applications\General\determine_sharpness.hdev`. In this example, a flat object is imaged, and the global sharpness of the image is determined. In the case of DFF, the sharpness will be determined for each single pixel. If you need real three-dimensional information, e.g., if you want to further use your DFF results for any 3D measurements or surface-based matching (see section 4.3 on page 104 you need to telecentrically calibrate your system. Information on how to perform a calibration can be found in the chapter about 3D camera calibration (section 3.2 on page 61).

Depth of field (DOF) is a similar term which, however, is not a method but a technical term concerning the camera. The depth of field is the range of distance within which the image is sharp as opposed to the best focus point, which is the point with perfect sharpness in the image. The DOF depends on the pixel size, the aperture (f-number), the focal length in the case of a non-telecentric lens, and the focusing distance. A low depth of field means that only a small slice of the object is sharply imaged (see figure 7.3), whereas a high depth of field means that a big part or maybe the whole image is sharp. For DFF, a low depth of field leads to a higher precision. In order to obtain a low depth of field, use lenses with a high aperture which is a small f-number on your lens.

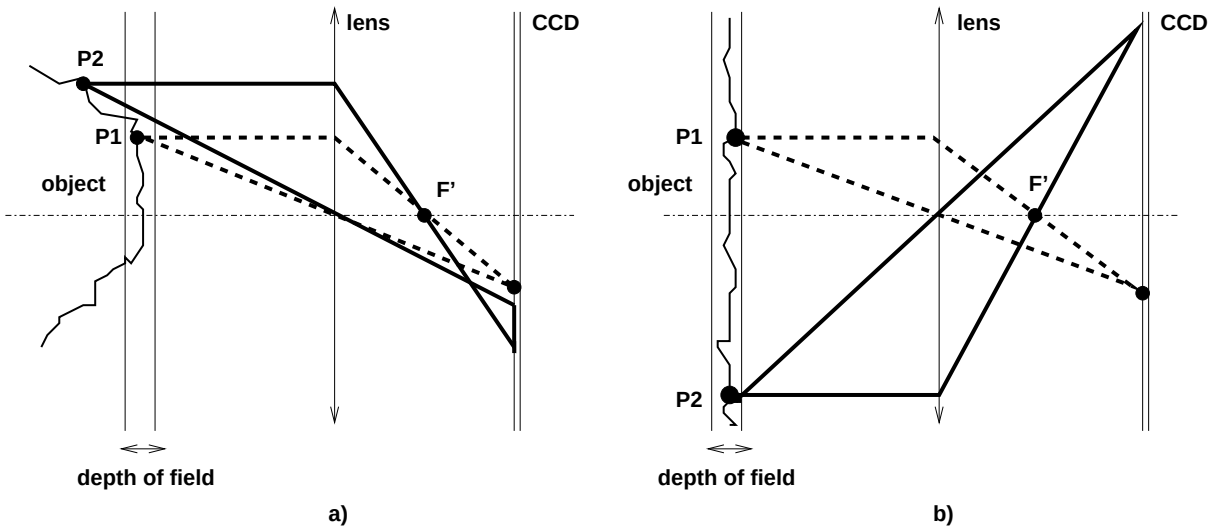


Figure 7.3: a) Only a small slice of the image is sharp. Therefore point P1 is mapped to a single point and is therefore sharp, whereas P2 is mapped to a spot and is consequently blurry. b) If the object is very flat, the whole object can be sharp at the same time, even if the depth of field is not very high.

A method for 3D surface measurement, similar to DFF, is depth from defocus. This method requires one sharp image of the foreground and one sharp image of the background. The distance of all points that lie between foreground and background is interpolated by their amount of blur. As it only depends on two images, it may be

faster, but it is also not as precise as depth from focus. Depth from defocus is not available in HALCON.

7.1.1 Speed vs. Accuracy

As mentioned before, depth from focus allows highly accurate 3D measurements. However, this accuracy comes at the price of a longer runtime.

7.1.1.1 Depth from Focus

DFF may consume more processing time due to the number of images that may need to be processed. This does, however, depend on the actual number of images that are necessary for the specific task. Generally speaking, the more images need to be processed, the higher the accuracy, the longer the runtime. Therefore, for some applications the runtime will naturally not be very high, because the required accuracy may be lower. Nevertheless, the runtime can also be improved for applications that require a high accuracy as well as for those which require a low accuracy. Internally, the speed of HALCON's `depth_from_focus` is increased using parallelization (only available in the `'local'` mode).

7.1.1.2 Depth from Focus vs. Other Methods

Due to the high number of images that have to be processed, the data volume is also high. This leads to a reduced speed compared to other methods. DFF has proved to be very useful in the microscopical range (small objects that are magnified more than once) and often easier to realize than other methods. In many cases, the setup can be more compact than stereo (chapter 5 on page 117) which needs quite a lot of space due to the two cameras. Nevertheless, the choice of a suitable lens is very important for DFF, and it can counter the advantage of using a single camera. For small objects having dimensions smaller than some ten millimeters, Sheet of Light (chapter 6 on page 147) might become very expensive because of the thin laser line that is needed and less precise due to speckle. Photometric stereo (see "3D Reconstruction ▷ Photometric Stereo") is usually more precise and easier to realize for flat objects without steep geometric edges. For macroscopic measurements (measurement range up to 100 mm), stereo or sheet of light would be quicker as they require less images.

7.2 Setup

Before actually starting your application with depth from focus, it is important to first set up your application environment properly. This section concentrates on the camera and application object setup to prepare the image acquisition. Additional information about equipment and image acquisition can be read in section 7.3.1 on page 170.

7.2.1 Camera

7.2.1.1 Recommended Camera Setup and Adjustments

As performance of depth from focus for your application depends on the used lens, the depth of field and the precision of the movements, it is important to use the right camera with the right adjustments. Please read our recommended camera setup and adjustments for the best results:

1. **If possible, use a camera with a telecentric lens.** In order to perform depth from focus, a camera with a telecentric lens or an almost telecentric lens, e.g., a microscopic imaging system, will produce the best results. Only a telecentric lens enables you to take images with exactly the same field of view at different focus positions. This is important because depth from focus uses image coordinates and for each pixel finds the image in which it is displayed sharply. Only in images taken with a telecentric lens, those pixel are comparable, i.e., in the same position. Therefore, DFF is a good method for measuring small objects, like, e.g., microelectronic workpieces. Good results have been achieved, for example, when performing DFF with the XENOPLAN telecentric lens series by Schneider-Kreuznach. Note however that even when using a telecentric lens, aberration - the effect that not all pixels are in focus on a planar surface - can

occur. Aberration influences the accuracy of DFF and should be calibrated. How to calibrate aberration is described in section 7.4.1 on page 172. It is also possible, though not recommended due to the reasons mentioned before, to use DFF with a standard lens. For performing DFF with a standard lens, please read section 7.6 on page 174. A problem that is related to the lens is the correspondence problem. This refers to effect that the position of the pixels outside the optical axis shifts when the distance is changed during image acquisition. Those planes that lie close to the border of the field of view as well as those that lie diagonal to the optical axis are affected most. A small depth of field reduces the effect for planes with other directions. The correspondence problem results in wrong focus distances in the depth image. This effect is minimal when using a telecentric lens. For telecentric lenses, it is important that the movement causing a change of focus is applied parallel to the optical axis. For more information about this effect see figure 7.5.

2. **Use mirrors to obtain focus images.** For DFF, mirrors are a very good solution because they can be moved very quickly and accurately and - if necessary - can be replaced easily and inexpensively. Moving the camera may be harmful to the camera sensor which suffers under the vibrations. Moving the object might be difficult as the object is probably located on a conveyor belt. All movements for DFF have to be performed very precisely. What such a setup with camera and mirrors can look like is shown in figure 7.4. Note that the focus should only be moved along the optical axis.

3. **Use a low depth of field to achieve a higher accuracy.** This requires, however, more images at different focus positions. In contrast, large distances between the images require a higher depth of field and lead to a less precise height reconstruction. A low depth of field requires - as stated before - images at various focus positions.

4. **Use a high aperture.** The aperture needs to be as open as possible as this reduces the depth of field which is responsible for a higher precision as mentioned before. The highest possible precision is, therefore, limited by the depth of field.

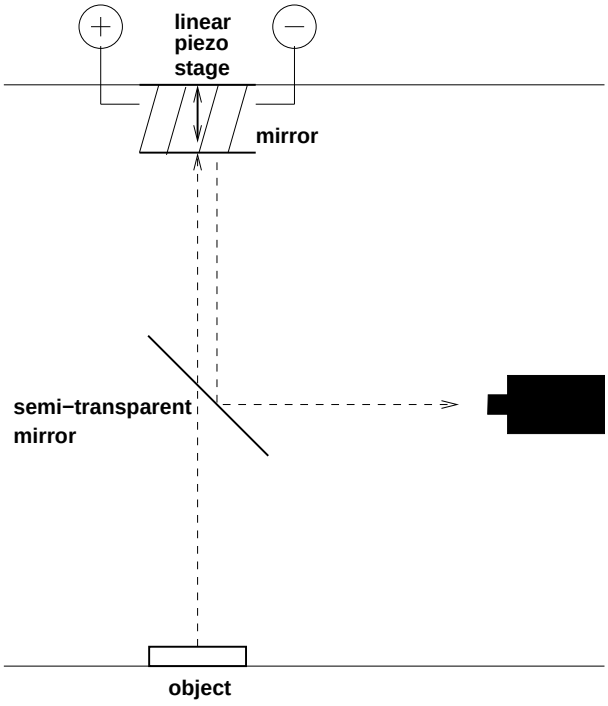


Figure 7.4: Rays from the object first cross a semi-transparent mirror and are then reflected by a mirror mounted on a linear piezo stage. The rays then reach the semi-transparent mirror again and are this time reflected back to the camera. By moving the mirror that is mounted on the linear piezo stage, the distance between the object and the camera can be varied in a controlled way, which makes it possible to acquire a sequence of images with varying focus.

7.2.1.2 Acquire Measurement Range

There are four rules that define how the measurement range can be acquired.

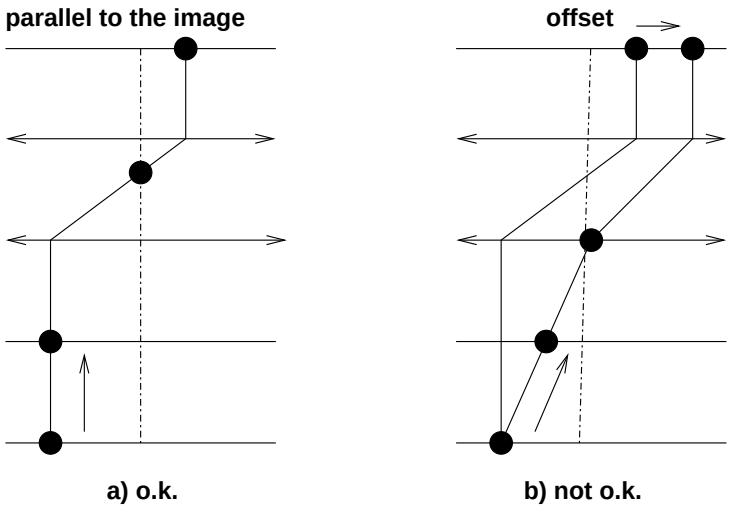


Figure 7.5: This figure illustrates the correspondence problem. In figure a), a change of focus is applied parallel to the optical axis, which is good. If the focus is not changed parallel to the optical axis, a lateral offset occurs, as depicted in figure b).

First rule: The distance range has to exceed the height of the measure object.

Second rule: The range in which the image part changes from blurred over sharp to blurred should be approximately five images (as depicted in figure 7.6). Otherwise, areas of the object's surface cannot be determined correctly and, therefore, cannot be measured precisely.

Third rule: As the depth of field of the used lens is fixed, the minimum number of focus positions has to exceed:

$$\frac{focus\ range\ in\ m}{depth\ of\ field\ of\ the\ used\ lens}$$

The reason for this is the shifted depth of field area of the lens. This shift needs to be smaller than the depth of field of the lens so that the depth of field areas from two successive images can overlap. The more those images overlap, the higher the achieved precision. The downside is an increased runtime due to the high number of images. You cannot, however, increase precision indefinitely as it is also limited by the camera noise. The smaller the overlap, the smaller the achieved precision, the shorter the runtime.

Fourth rule: If there is a limit to the number of images that can be acquired, for example, because you have limited runtime, the depth of field of the used lens needs to be increased (e.g., by closing the aperture of the lens).

Images from a focus stack can be viewed in figure 7.7.

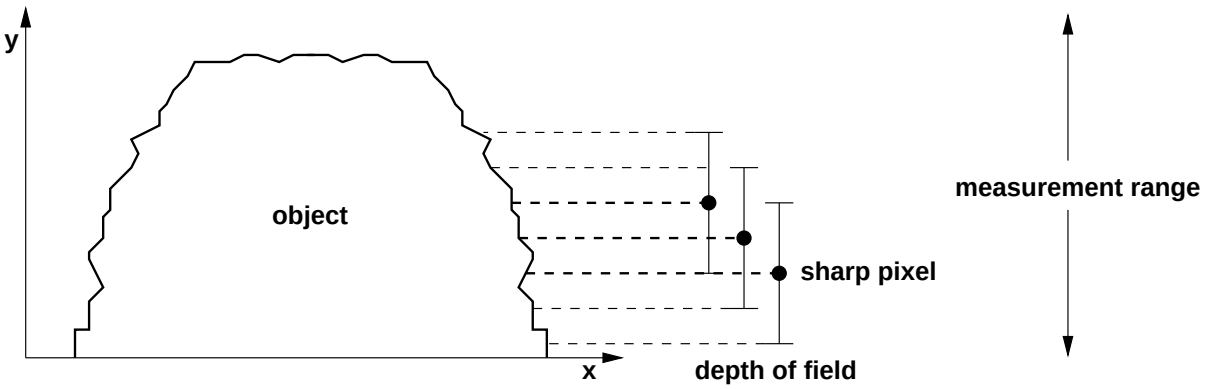
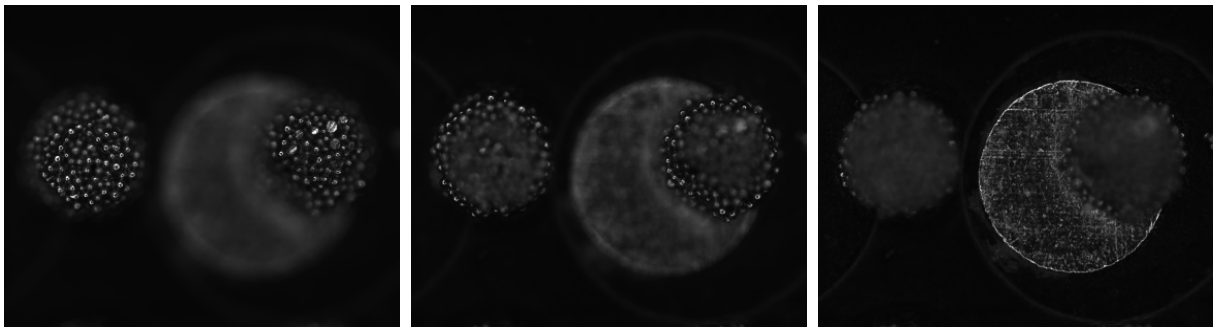


Figure 7.6: This image depicts an object from which images are taken at different focusing distances (indicated by the dashed lines) for a focus sequence. It also exemplarily shows the depth of field, consisting of five images in which the image with the sharpest pixel is marked by a dot.

Depending on the direction of image acquisition, i.e., from the camera or towards the camera, the result of the measurement is either a distance image or a height image.



a) b) c)

Figure 7.7: The images above are part of a focus sequence. They show the same object but are taken at different focus positions. Image a) focuses on the top of the object, image b) is sharp at a medium height and image c) is sharp in the background.

7.2.2 Illumination

7.2.2.1 Choosing Illumination

In order to enhance the surface texture of the object, direct illumination is needed. At the same time, reflections have to be minimized. Therefore, coaxial and light-field lighting would not work. A suitable lighting could be an illumination from various directions, as depicted in figure 7.8, because lighting that comes from different directions enhances the structure very well and also causes few reflections. As an alternative dark-field lighting is also possible, because the low angle enhances the surface structure - it does, however, result in an image that has quite dark parts in some areas. Other illumination setups can be used as long as the lighting enhances the surface texture and causes as few reflections as possible. A suitable illumination, therefore, highly depends on the application object's features. For more information about lighting, please refer to the Solution Guide II-A,appendix C.1 on page 49.

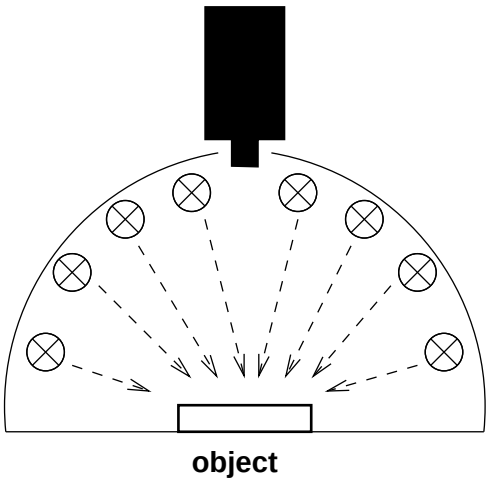


Figure 7.8: Illumination from various directions leads to good results for depth from focus. It can, for example, be produced with lights that are arranged within a dome.

7.2.2.2 Overexposure

Overexposure is an illumination-connected problem that may occur and reduce the accuracy of the resulting image. It leads to a loss of information in the overexposed region which then reaches saturation (a gray value of 255). Furthermore, overexposure causes the detection of false sharp pixels in blurry areas which are the result of high frequencies between saturated image areas and blurry areas. The results of overexposure are visualized in figure 7.9. In order to avoid overexposure, it is recommended to take darker images for your measurements

and improve visualization with the operators `scale_image` and `scale_image_max` as presented in the example section 7.3.2.1 on page 171.

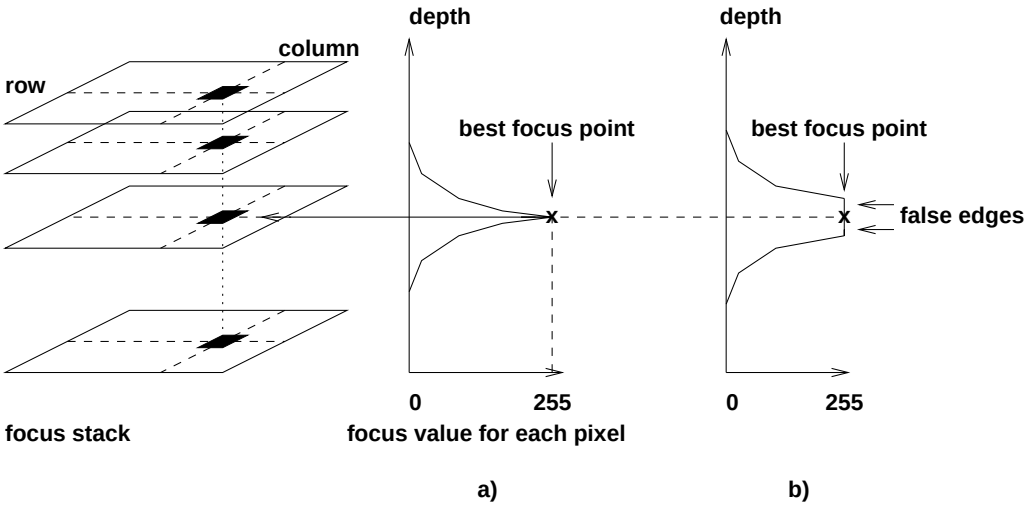


Figure 7.9: a) The sharp pixel in this figure is overexposed. b) If wider parts of the image are overexposed, instead of detecting the correct location of a sharp pixel, wrong sharp pixels are detected at the border between the saturated image areas and the blurry image areas.

7.2.3 Object

In order to successfully perform DFF, it is important to know your application object.

First of all, DFF is a practical method for any small object with textured surface. For objects that are larger than that, there are several other 3D methods available like binocular stereo (chapter 5 on page 117). DFF can, however, still be used for very high objects that would usually require a very high number of images and thus slow down the measuring process a lot. If the height of such an object is required there are two possibilities of gaining these measuring results.

- One possibility is to take a certain number of images (e.g., 10) that depict the bottom of the object, then also take a certain number of images showing the top of the object. Because you know the distance between the highest image that is taken of the bottom of the object and the first image taken of the top of the object, you can just add this height to the images taken of the top of the object.
- Another possibility is to acquire two sequences: one has to be taken around the bottom of the object and another one at the top of the objects. Within each of the sequences the sharpest level is detected. As the movement of the motor that determines the acquisition of images at different focus levels is known, the distance between those images can be calculated.

Two image sequences are acquired, one taken around the highest point of measurement and one sequence around the lowest point of measurement. Once, the sharpest pixels at the highest and lowest point are found, the height can be calculated.

Regular (passive) DFF can be used for objects with a textured surface. Fortunately, small objects mostly have a structured surface when observed at a suitable magnification. The exception are perfectly polished, i.e., specular surfaces.

If there is very little texture on the object, it is possible to still perform the so-called active depth from focus. Active DFF compensates for the missing surface structure by projecting a texture on the object. When performing active DFF, it is possible to produce focus images with the projector. It is, however, necessary that - for every image that is taken - the depth of field of the camera is conform with the depth of field of the projector. Active DFF does work for objects with little texture on their surfaces, it does not work for reflecting surfaces, though.

7.3 Working with Depth from Focus

Depth from focus requires a focus stack of images, i.e., images that each have a different distance to the object and in which, therefore, different pixels are sharp. Depth from focus is then applied to this focus stack returning a sharp image, i.e., an image containing only gray values from focused, sharp pixels as these are the relevant pixels for DFF, as well as a depth image showing the three-dimensional shape of the object that is inspected. Those two results can be viewed in figure 7.10.

7.3.1 Rules for Taking Images

There are several rules that should be followed when taking images for measuring with depth from focus. Those rules concern image quality as well as the system setup.

7.3.1.1 Rules for Achieving a High Image Quality

1. **Avoid overexposure.**
2. **Avoid reflections.**
3. **Use a camera with a wide dynamic range.**
4. **Use direct illumination** - preferably from several directions.
5. **Use a camera with low noise.**
6. **The maximum number of images is limited by camera noise.** Taking more images to cover the whole object only improves the precision of the reconstruction to a certain degree which is, amongst others, limited by the camera noise.
7. **As a rule of thumb, the minimum number of images that should be acquired for DFF is 10 and the maximum number is about 150.** While 10 images can be just sufficient for measuring (depending on the required accuracy), more than 150 images will usually not improve accuracy any more. The corresponding formula would be:
$\frac{object\ height}{depth\ of\ field} \times 5$

7.3.1.2 Rules for the Best Results with your System Setup

1. **Cover the whole distance range.**
2. **Let the depth of field areas overlap (see figure 7.6).**
3. **The distance between the light source and the object should remain constant.**
4. **The optical system should enable an orthographic projection (parallel projection)**, i.e., use a telecentric lens or a microscope in your system setup. If this is impossible, please read section 7.6 on page 174.
5. **The axis of the focusing displacement must be parallel to the optical axis of the lens**, otherwise the object shifts laterally within the focus sequence.
6. **Do not move the camera or object in x or y direction**. DFF does not work on an object that is, e.g., moving on a conveyor belt between images or an object that is moved by any kind of vibration or agitation.

Note that the quality of your measuring results does depend on the quality of the input images and you should therefore aim to achieve the highest possible quality.

7.3.2 Practical Use of Depth from Focus

7.3.2.1 Example Application: Inspecting a PCB with DFF

This section describes an example application where the task is to test if a PCB board is covered by an appropriate amount of soldering paste. This inspection is performed with DFF and can be viewed in the HDevelop example program %HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\measure_solder_paste_dff.hdev (for more information also refer to the example %HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\measure_bga_dff.hdev which is a similar program showing how a ball grid array is measured with DFF). An image sequence is acquired and a height map of the single circuits and pads is calculated. This way parts that have no soldering paste can be identified as well as those that are covered sufficiently.

First all necessary images for the focus series have to be acquired and combined to a multi-channel image with the operator channels_to_image.

```
read_image (ImageArray, 'dff/focus_pcb_solder_paste_' + Sequence$'02')
channels_to_image (ImageArray, Image)
```

Then, depth from focus is performed, a depth map is calculated and all sharp gray values are selected. Both results can be used for further processing. Depth from focus is performed with the HALCON operator depth_from_focus. The channel number is returned for each pixel together with a confidence value, which is an indicator for the quality of the distance value. Pixels with the best focus are chosen. The method can be selected with the parameters Filter and Selection.

It is striking that the images are very dark, when looking at the images in the example %HALCONEXAMPLES%\hdevelop\Applications\Measuring-3D\measure_solder_paste_dff.hdev. They have been acquired like this on purpose to avoid overexposure. However, to improve the visibility of the object's surface in the image, the operator scale_image enhances the sharpness, scale_image_max spreads the gray values in the image to improve visibility despite of the darkness. For more information on depth from focus and overexposure please refer to section 7.2.2.2 on page 168. median_rect suppresses unwanted outliers.

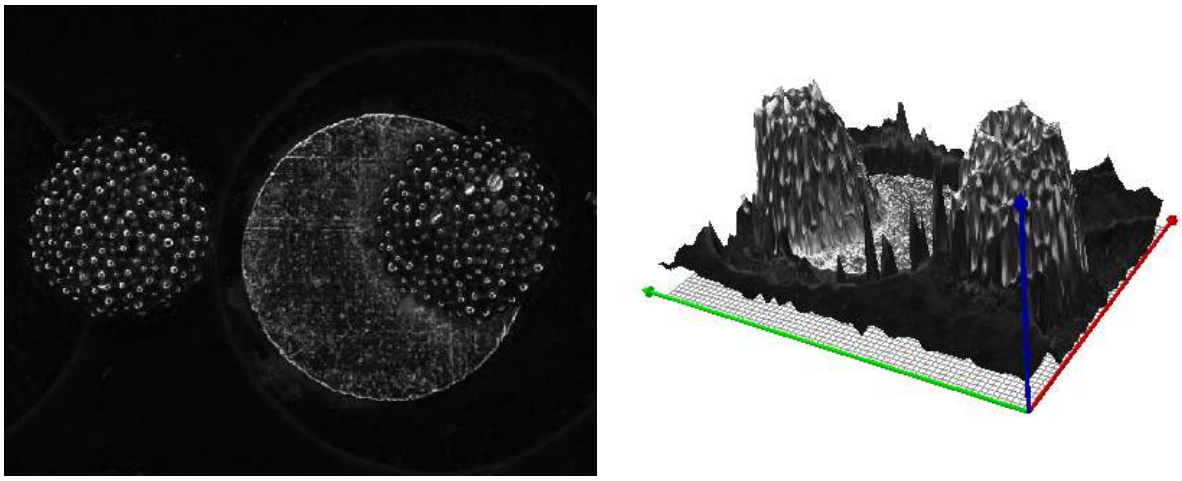
```
depth_from_focus (Image, Depth, Confidence, 'bandpass', 'next_maximum')
select_grayvalues_from_channels (Image, Depth, SharpenedImage)
scale_image (SharpenedImage, ImageScaled, 4, 0)
scale_image_max (Depth, ImageScaleMax)
median_rect (ImageScaleMax, DepthMean, 25, 25)
```

Finally, the results - the sharp image as well as the 3D plot - are displayed as shown in figure 7.10. A sharp image is reconstructed by selecting the gray value of each pixel that is in focus for each coordinate using the depth image as index table. The focus stack and the depth image are used as input to reconstruct a focused image using the parameters MultiChannelImage, IndexImage and Selected. Sharp gray values can be identified by high frequencies, i.e., high edge amplitudes in the image, i.e., where the gray-value information changes quickly. For each sharp pixel, a confidence score is returned.The amount of sharpness defines the score. Furthermore, a 3D plot of the object in the image is calculated which is helpful for detecting defects.

```
dev_open_window (0, Width * 0.7 + 5, Width * 0.7, Height * 0.7, 'black', \
                 WindowHandle3D)
dev_set_paint (['3d_plot', 'texture'])
compose2 (DepthMean, ImageScaled, MultiChannelImage)
dev_display (MultiChannelImage)
```

7.3.3 Volume Measurement with Depth from Focus

In contrast to stereo, the height information is not calibrated for depth from focus. The values in the height image are indices of input images. To measure a real world height or volume, the distance in between these images must be known. The easiest case is that images are taken with the same movement z. If two coordinates differ by an index value of n, the real world distance will be $z \times n$, with the unit of z. The unit in x and y, i.e., the size of a pixel must be known. Make sure that all dimensions are given in the same unit. Volume can be determined with the operator area_center_gray by adding up the pixel values which are equal to the height values. The resulting value must be multiplied by x, y and z.



a) b)

Figure 7.10: Results of the PCB solder inspection: a) a synthetic sharp image and b) a 3D plot of the object.

7.4 Solutions for Typical Problems With DFF

There are two main problems that occur with depth from focus: overexposure and reflections.

1. **Overexposure leads to extreme values** (255) and results in peaks within the constant gray-value range of an image. This means that even within parts of the images that are blurry, the difference between neighboring gray values is so dominant that false edges might be detected (this effect is visualized in figure 7.9 on page 169). Adjusting your illumination to minimize reflections can reduce overexposure For more information on suitable illumination and handling overexposure, please refer to section 7.2.2 on page 168.
2. **Object surfaces may reflect too much** for measuring with depth from focus. One advantage of depth from focus is, however, that you are very close to the object. Therefore, some objects that seem to have a reflecting surfaces might under the microscope show some surface texture after all. Try to minimize reflections with diffuse illumination as described in section 7.2.2 on page 168. Furthermore, aberration occurs for DFF especially when images were taken with a standard lens but it also cannot totally be avoided for telecentric lenses either.

7.4.1 Calibrating Aberration

7.4.1.1 Aberration

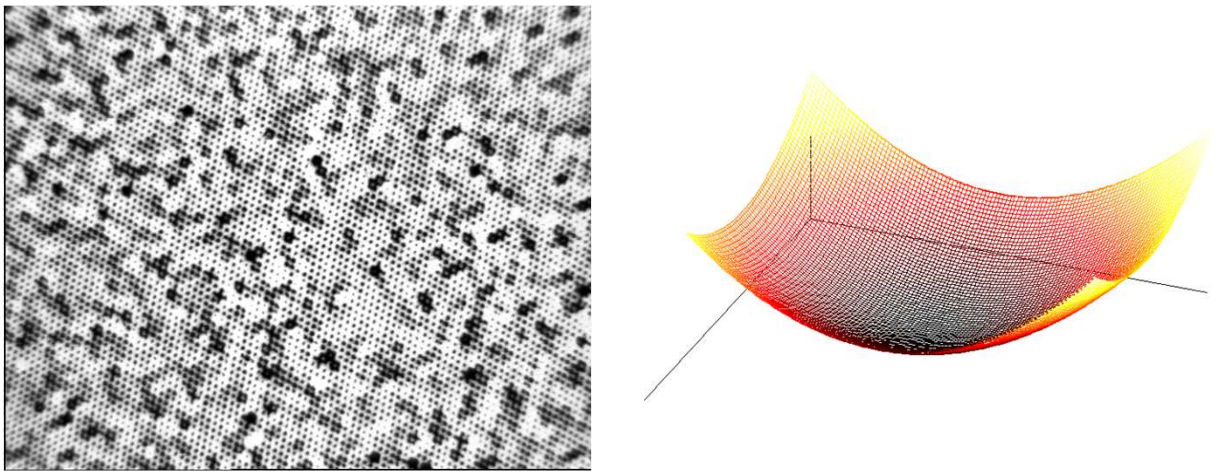
Aberration is the effect that, when looking perpendicularly on a planar surface, not all pixels are in focus at the same time. Either the center or the outer part of the image are completely in focus. This effect is illustrated in figure 7.11.

Aberration is the curvature of field that effects images taken with cameras using standard lenses but also cannot be completely avoided when using cameras with telecentric lenses. It results in an error in the depth image.

There are two main kinds of optical aberration:

1. **Spherical aberration** occurs when light rays have different focus points depending on their distance to the optical axis, i.e., the center part and the border of the image are not simultaneously sharp, even though a planar object is imaged.
2. **Coma** is the asymmetric accumulation of light intensity for off-axis points. It effects the periphery of the field of view. This kind of aberration is, however, not relevant here and is just mentioned for the sake of completeness.

The following paragraph describes how to calibrate spherical aberration and therefore avoid errors.



a) b)

Figure 7.11: Image a) is an example for aberration. It is sharp in the middle and pixels become more blurry the further away they are from the center. Image b) shows a graphic that clarifies what happens when aberration occurs.

7.4.1.2 Setup Aberration Calibration

In order to get an accurate result from your depth from focus application, aberration has to be calibrated. This enables the calculation of correct sharp depth images and also enable further processing of the DFF results.

Rules for setting up your calibration to correct aberration.

1. **A planar surface with reasonable texture is needed** as reference plane.
2. **The camera has to be mounted perpendicular to the surface** (angles can be determined with HALCON's camera calibration if necessary).
3. **It is important that the same distance and camera setting are used** as will be applied during application!
4. **Using depth from focus, the "curvature on the surface", i.e., the aberration is determined.**
5. **The extracted reference surface is used to correct the later measurements.**
6. **It is recommended to store the reference image to file** for further use.
7. **It is recommended to store the used parameters to file** for further use.
8. **The aberration can be approximated by a paraboloidal function.**
9. **The approximation has the advantage of reducing noise effects** that influence the measurements.
10. **The parameters of the paraboloidal function can be used to generate a reference image.**
11. **By subtracting the reference image from the depth measurement the error caused by the aberration can be corrected.**

7.5 Special Cases

Depth from focus may not always be used to reconstruct the three-dimensional surface of an object in order to subsequently measure it.

It may also be used to obtain a sharp image of the object if this is not otherwise possible due to the setup and continue working with this image.

Another possibility of using the DFF method is simply checking whether an object is present or not. Therefore, the lens is focused on a certain depth. If sharp pixels are found within an image at this certain depth, the object is present, otherwise it is missing. In this case, only one image can be sufficient.

Similarly, it can be used to check whether an object is tilted or not.

7.6 Performing Depth from Focus with a Standard Lens

Even though not recommended, it is possible to perform depth from focus with standard lenses. Using a non-telecentric lens does, however, require some adaptations of the DFF measuring process to ensure the best results.

Note that the accuracy of DFF measurements with a standard lens instead of a telecentric lens is reduced. For DFF with standard lenses, the focal length needs to be as long as possible in order to keep the perspective shift of the points in the field of view small. Furthermore, the depth of field needs to be small.

If the points shift too much within the field of view, measuring with DFF produces an erroneous result.

Calibrate aberration as described in section 7.4.1 on page 172.

Note that the rules that have been described for DFF with a telecentric lens are also valid for DFF with a standard lens.

Chapter 8

Robot Vision

A typical application area for 3D vision is robot vision, i.e., whenever robots are equipped with cameras that supply information about the parts to be handled. Such systems are also called “hand-eye systems” because the robotic “hand” is guided by mechanical “eyes”.

In order to use the information extracted by the camera, it must be transformed into the coordinate system of the robot. Thus, besides calibrating the camera(s) you must also calibrate the hand-eye system, i.e., determine the transformation between camera and robot coordinates. The following sections explain how to perform this hand-eye calibration with HALCON.

Please note that in order to use HALCON’s hand-eye calibration, **the camera must observe the workspace of the robot**. If the camera does not observe the workspace of the robot, e.g., if the camera observes parts on a conveyor belt, which are then handled by a robot further down the line, you must determine the relative pose of robot and camera with different means.

The calibration result can be used for different tasks. Typically, the results of machine vision, e.g., the position of a part, are to be transformed from camera into robot coordinates to create the appropriate robot commands, e.g., to grasp the part. Section 8.7 on page 185 describes such an application. Additionally, the HDevelop programs `%HALCONEXAMPLES%\hdevelop\Applications\Robot-Vision\pick_and_place_with_2d_matching_moving_cam.hdev` and `%HALCONEXAMPLES%\hdevelop\Applications\Robot-Vision\pick_and_place_with_2d_matching_stationary_cam.hdev` show how to grasp objects using the hand-eye calibration data. These examples can be adapted easily for all kinds of pick-and-place-applications.

Another possible application for hand-eye systems is to transform the information extracted from different camera poses of a moving camera into a common coordinate system.

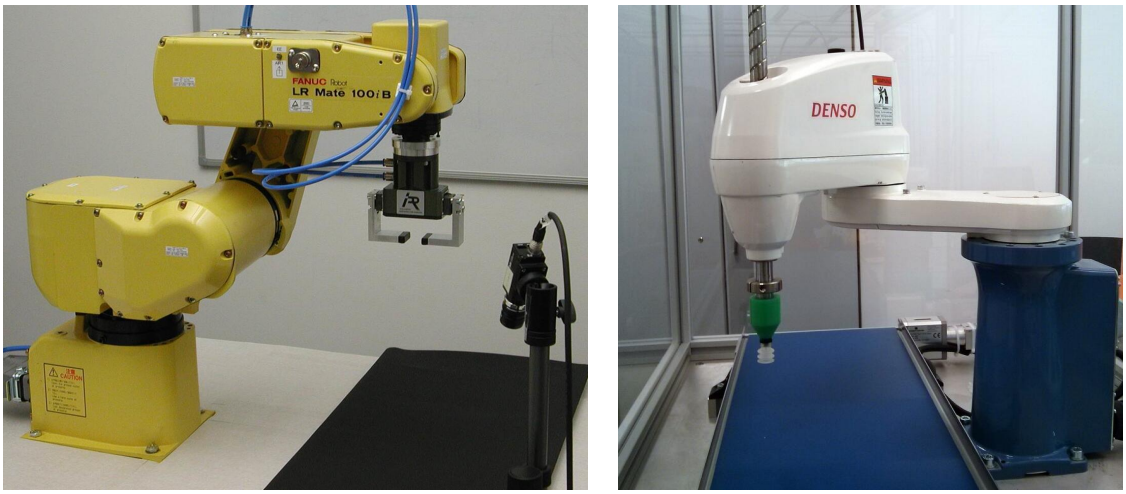
8.1 Supported Configurations

The hand-eye calibration of HALCON supports different configurations, i.e., kinds of robots and sensors and ways, the sensor is mounted. In the following, these configurations are briefly discussed.

8.1.1 Articulated Robot vs. SCARA Robot

The arm of an articulated robot (figure 8.1a) typically has three rotary joints covering 6 degrees of freedom (3 translations and 3 rotations). In contrast, SCARA (selective compliance articulated robot arm) robots (figure 8.1b) have typically three parallel rotary joints and one parallel prismatic joint covering only 4 degrees of freedom (3 translations and 1 rotation).

Articulated (anthropomorphic) robots can pick up a part, no matter how it is oriented and then insert it into a package that may require a special angle under which it is approached. Therefore, they can be used in a very flexible manner. In contrast, SCARA robots are restricted in their movements. They cannot tilt the tool. But they offer faster and more precise performance. They are best suited for high-speed pick and place, packaging,



(a) (b)

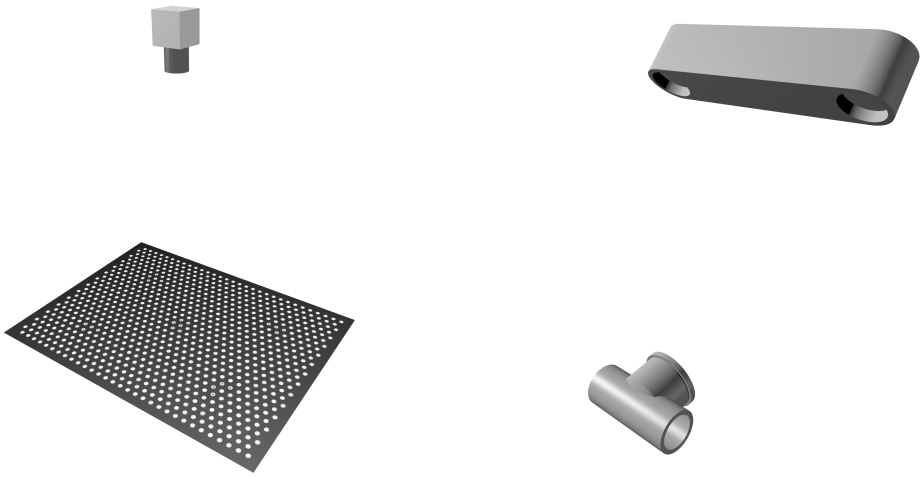
Figure 8.1: Different kinds of robots: (a) Articulated robot and (b) SCARA robot.

and assembly applications. Because of their compact structure, SCARA robots are often preferred if only limited space is available.

The hand-eye calibration of articulated robots and of SCARA robots is very similar, with the exceptions that for SCARA robots the camera must be calibrated prior to the actual hand-eye calibration and that it is necessary to resolve one ambiguity by manually moving the tool of the robot to a known position. These points where the calibration of SCARA robots differs from the calibration of articulated robots are described in section 8.3 on page 179 and section 8.6 on page 184.

8.1.2 Camera and Calibration Plate vs. 3D Sensor and 3D Object

Depending on the application, either an optical camera or a 3D sensor is used to identify the objects and to determine their pose. If a camera is used, for the calibration of the hand-eye system, a calibration plate or calibration object is required (figure 8.2a). Note that the automatic detection of the calibration object works only if one of the HALCON calibration plates (section 3.2.3.1 on page 67) is used. If a 3D sensor is used (figure 8.2b), the pose of known 3D objects can be determined, e.g., with surface-based matching (Solution Guide I, chapter 11 on page 101). These poses can then directly be used for the calibration of the hand-eye system.



(a) (b)

Figure 8.2: Different kinds of sensors: (a) Camera with HALCON calibration plate and (b) 3D sensor with 3D object.

Note that, although in the following only systems with a single camera are described, you can of course also use a

stereo camera system (see section 5.2.4 on page 124). By distinctly calibrating the stereo system beforehand you determine the relation between its cameras. In this case, when calibrating your hand-eye system afterwards (with its own calibration model), you typically calibrate only the relation of the robot to one of the cameras.

8.1.3 Moving Camera vs. Stationary Camera

There are two possible scenarios for mounting the camera:

Moving camera
The camera is mounted at the tool and is moved to different positions by the robot.

Stationary camera
The camera is mounted externally and does not move with respect to the robot base.

Figure 8.3 depicts these two scenarios. Note that different poses must be determined by the hand-eye calibration with a moving camera compared to the hand-eye calibration with a stationary camera. In the following, the paragraphs that affect only one of these scenarios are marked respectively.

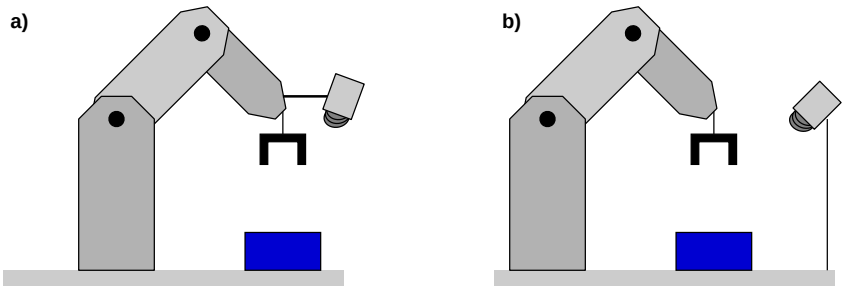


Figure 8.3: Robot vision scenarios: (a) moving camera, (b) stationary camera.

Note that HALCON's hand-eye calibration is not restricted to systems with a "hand", i.e., a manipulator. You can also use it to calibrate cameras mounted on a pan-tilt head or surveillance cameras that rotate to observe a large area. Both systems correspond to a camera mounted on a robot; the calibration then allows you to combine visual information from different camera poses.

For the hand-eye calibration with a stationary camera, the example `%HALCONEXAMPLES%\hdevelop\Calibration\Hand-Eye\calibrate_hand_eye_stationary_cam_approx.hdev` is available, which shows two alternative workflows to perform a hand-eye calibration, either with or without a HALCON calibration plate.

8.1.4 Calibrating the Camera in Advance vs. Calibrating It During Hand-Eye Calibration

For articulated robots that are used together with *one* camera, it is possible to calibrate the camera together with the hand-eye setup. For SCARA robots and all systems that use stereo or multi-view setups, the camera(s) must be calibrated distinctly in advance.

8.2 The Principle of Hand-Eye Calibration

Like the camera calibration (see section 3.2 on page 61), the hand-eye calibration is based on providing multiple images of a known calibration object. But in contrast to the camera calibration, here, the calibration object is not moved manually. Either it is moved by the robot in front of a stationary camera or the robot moves the camera over a stationary calibration object. The pose, i.e., the position and orientation, of the robot tool in robot base coordinates for each calibration image **must be known with high accuracy!**

This results in a chain of coordinate transformations (see figure 8.4 and figure 8.5). In this chain, two transformations (poses) are known: the pose of the robot tool in robot base coordinates ${}^{base}\mathbf{H}_{tool}$ and the pose of the calibration object in camera coordinates ${}^{cam}\mathbf{H}_{cal}$, which is determined from the calibration images. The hand-eye

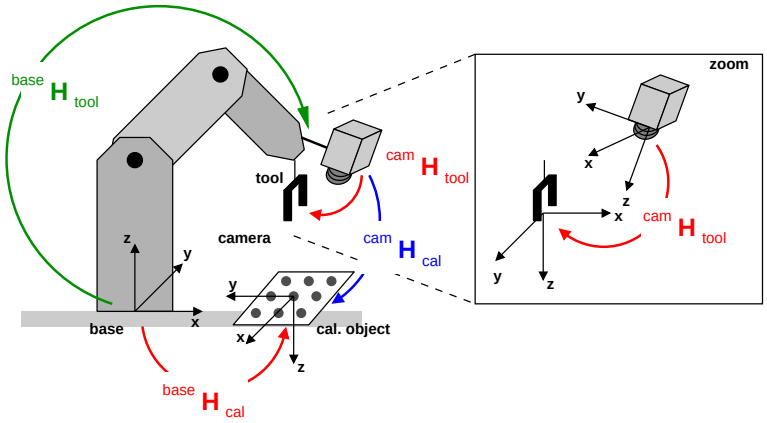


Figure 8.4: Chain of transformations for a moving camera system.

calibration then estimates the other two poses, i.e., the relation between the robot and the camera and between the robot and the calibration object, respectively. Note that the chain consists of different poses depending on the used scenario.

Moving camera

For a *moving camera*, the pose of the robot tool in camera coordinates and the pose of the calibration object in robot base coordinates are determined (see figure 8.4):

$$^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{cal} \tag{8.1}$$

In this chain the inverse input pose, i.e., the pose of the robot base in tool coordinates, is used.

$$^{tool}\mathbf{H}_{base} = ({}^{base}\mathbf{H}_{tool})^{-1} \tag{8.2}$$

However, the inversion of the pose is done internally by the hand-eye calibration algorithm.

Stationary camera

For a *stationary camera*, the pose of the robot base in camera coordinates and of the calibration object in robot tool coordinates are determined (see figure 8.5):

$$^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cal} \tag{8.3}$$

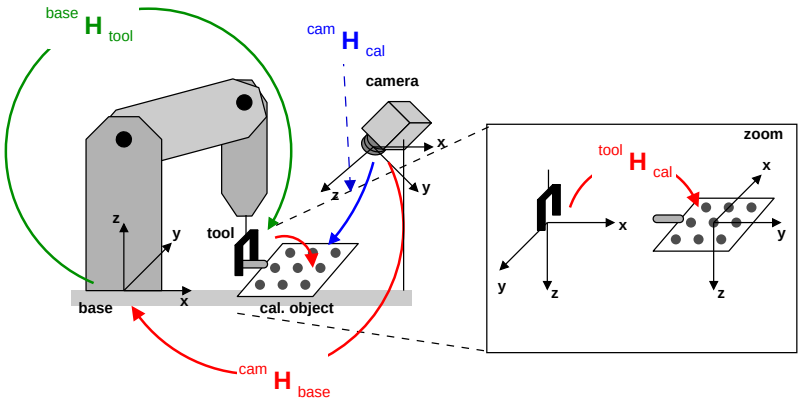


Figure 8.5: Chain of transformations for a stationary camera system.

Once all input data has been set as detailed in section 8.4, the hand-eye calibration is performed with a single call of the operator calibrate_hand_eye.

```
calibrate_hand_eye (CalibDataID, Errors)
```

Let's have a brief look at the parameters. The referenced sections contain more detailed information:

- CalibDataID

  As for the camera calibration, you have to prepare a calibration input data model by setting all required input data. Specifically, the observed poses of the calibration object in camera coordinates and the corresponding poses of the robot tool in robot base coordinates have to be set. This is described in detail in section 8.4.

- Errors

  These errors represent the quality assessment of the transformations determined during the hand-eye calibration. How to use the transformations and how to interpret the quality values in robot vision applications is described in section 8.7 on page 185. If you get a high error, use the procedure check_hand_eye_calibration_input_poses to check the input poses for consistency.

Besides the coordinate systems described above, two others may be of interest in a robot vision application: First, sometimes results must be transformed into a reference (world) coordinate system. You can define such a coordinate system easily based on a calibration image. Secondly, especially if the robot system uses different tools (grippers), it might be useful to place the tool coordinate system used in the calibration at the mounting point of the tools and introduce additional coordinate systems at the gripper (tool center point). The example application in section 8.7 on page 185 shows how to handle both cases.

8.3 Calibrating the Camera in Advance

This step is not required for 3D sensors and it is optional for articulated robots that are used together with one camera, but it is obligatory for SCARA robots and for all systems with a stereo or multi-view camera setup.

Articulated robot

For hand-eye systems with an articulated robot and one camera, typically, the camera is calibrated together with the whole hand-eye system. Therefore, it is not necessary to calibrate the camera in advance.

If a stereo or multi-view camera setup is used instead of one camera, these cameras must be calibrated in advance by the method described in section 5.2 on page 122. If such a calibrated camera setup is used for hand-eye calibration (or if one already calibrated camera is used whose camera parameters should be preserved), the internal camera parameters must be excluded from the optimization with

```
set_calib_data (CalibDataID, 'camera', 'general', 'excluded_settings', \
                'params')
```

SCARA robot

For hand-eye systems with a SCARA robot, the camera must be calibrated in advance (see, e.g., section 3.2 on page 61). The internal camera parameters are automatically excluded from the optimization.

For a simple camera calibration, have a look at the HDevelop example program %HALCONEXAMPLES%\hdevelop\Calibration\Hand-Eye\calibrate_hand_eye_stationary_cam_approx.hdev, which includes a camera calibration with a single calibration image.

8.4 Preparing the Calibration Input Data

Below, we show how to prepare the data in the calibration data model CalibDataID that is then used by calibrate_hand_eye. The code examples mostly stem from the HDevelop programs %HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_movingcam_calibration.hdev and

%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_stationarycam_calibration.hdev, which perform the calibration of hand-eye systems with a moving and a stationary camera, respectively. The program %HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_movingcam_calibration_poses.hdev performs the hand-eye calibration by directly setting the poses of the calibration object obtained with a 3D sensor.

8.4.1 Creating the Data Model

First, a calibration data model for the hand-eye calibration has to be created. The following types are supported:

Robot type (see section 8.1.1 on page 175):

- Articulated robot
- SCARA robot

Sensor type (see section 8.1.2 on page 176):

- Camera (with calibration plate)
- 3D sensor (with 3D object)

Camera mounting type (see section 8.1.3 on page 177):

- Stationary camera
- Moving camera

Note that in connection with the mounting type (stationary or moving camera (='eye')), always the term camera is used, even if a 3D sensor is used.

The robot type and the camera mounting type are defined in the parameter `CalibSetup` of `create_calib_data`. For example, *'hand_eye_moving_cam'* defines a system of an articulated robot with a moving camera, while *'hand_eye_scara_stationary_cam'* defines a system of a SCARA robot with a stationary camera.

The sensor type is defined by the two parameters `NumCameras` and `NumCalibObjects` of `create_calib_data`. If a camera with a calibration plate is used, both parameters must be set to *1*. If a 3D sensor is used, both parameters must be set to *0*.

In the following, the respective calls of `create_calib_data` are shown for all eight combinations of the above three categories:

Articulated robot — Camera — Stationary

```
create_calib_data ('hand_eye_stationary_cam', 1, 1, CalibDataID)
```

Articulated robot — Camera — Moving

```
create_calib_data ('hand_eye_moving_cam', 1, 1, CalibDataID)
```

Articulated robot — 3D sensor — Stationary

```
create_calib_data ('hand_eye_stationary_cam', 0, 0, CalibDataID)
```

Articulated robot — 3D sensor — Moving

```
create_calib_data ('hand_eye_moving_cam', 0, 0, CalibDataID)
```

SCARA robot — Camera — Stationary

```
create_calib_data ('hand_eye_scara_stationary_cam', 1, 1, CalibDataID)
```

SCARA robot — Camera — Moving

```
create_calib_data ('hand_eye_scara_moving_cam', 1, 1, CalibDataID)
```

SCARA robot — 3D sensor — Stationary

```
create_calib_data ('hand_eye_scara_stationary_cam', 0, 0, CalibDataID)
```

SCARA robot — 3D sensor — Moving

```
create_calib_data ('hand_eye_scara_moving_cam', 0, 0, CalibDataID)
```


When a camera with a calibration plate is used, i.e., if the two parameters NumCameras and NumCalibObjects of create_calib_data are both set to *1*, the camera parameters and the calibration plate must be set in the calibration data model:

```
set_calib_data_cam_param (CalibDataID, 0, [], StartCamParam)
set_calib_data_calib_object (CalibDataID, 0, CalTabFile)
```

There are different algorithms to choose from for the optimization of the calibrated poses. While a *'linear'* and a more precise *'nonlinear'* algorithm can be used in any case, there is also the *'stochastic'* algorithm, which is more robust but does not work with 3D cameras. This is the case because it also considers uncertainties of the robot poses and compensates them based on the provided images of the calibration plate.

'nonlinear' is the default method used by calibrate_hand_eye. It can also be set explicitly as follows. For further information about the optimization methods, see the documentation of calibrate_hand_eye.

```
set_calib_data (CalibDataID, 'model', 'general', 'optimization_method', \
                'nonlinear')
```

8.4.2 Poses of the Calibration Object

Camera with HALCON calibration plate

When using a camera and the HALCON calibration plate, the pose of the calibration plate is determined in each calibration image and saved in the calibration data model for the hand-eye calibration using find_calib_object. Please note that for the hand-eye calibration **we strongly recommend to use the calibration plate with hexagonally arranged calibration marks** (see section 3.2.3.1 on page 67). This calibration plate is much easier to handle than the one with rectangularly arranged calibration marks because it may be partly occluded or placed partly outside the image. Do not use the old calibration plate with rectangularly arranged calibration marks and without the asymmetric pattern in one of its corners because if even in a single calibration image the pose of the old, symmetric calibration plate is estimated wrongly because it is rotated by more than 90 degrees, the calibration will fail!

```
for I := 0 to NumImages - 1 by 1
    dev_set_window (WindowHandle)
    dev_clear_window ()
    read_image (Image, ImageNameStart + I$'02d')
    dev_display (Image)
    find_calib_object (Image, CalibDataID, 0, 0, I, [], [])
```

Camera with generic calibration object

If a different calibration plate is used, the extracted marker position can be set with set_calib_data_observ_points. These are then used internally to determine the pose of the calibration plate. For more details on using generic calibration plates refer to section 3.2.3.2 on page 70.

3D sensor

If a generic 3D sensor is used, the observed poses are set explicitly in the calibration data model.

```
    set_calib_data_observ_pose (CalibDataID, 0, 0, I, CalObjInCamPose)
```

8.4.3 Poses of the Robot Tool

For each of the calibration images or observed poses, the corresponding pose of the robot must be specified. Note that **the accuracy of the poses is critical to obtain an accurate hand-eye calibration**. There are two ways to "feed" the poses into HALCON: In many cases, you will simply read them from the robot control unit and then enter them into your HALCON program manually. For this, you can use the HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_create_robot_poses.hdev`, which lets you input the poses in a text window and writes them into files.

As an alternative, if the robot has a serial or socket interface, you can also send them via this connection to your HALCON program (see the sections "System ▷ Serial" and "System ▷ Sockets" in the Reference Manual for more information).

In both cases, you then convert the data into HALCON 3D poses using the operator `create_pose`. As described in section 2.1.4 on page 20 (and in the Reference Manual entry for `create_pose`), you can specify a pose in more than one way, because the orientation can be described by different sequences of rotations. Therefore, you must first check which sequence is used by your robot system. In many cases, it will correspond to

$$\mathbf{R}_{abg} = \mathbf{R}_z(\texttt{RotZ}) \cdot \mathbf{R}_y(\texttt{RotY}) \cdot \mathbf{R}_x(\texttt{RotX}) \tag{8.4}$$

If this is the case, select the value *'abg'* for the parameter `OrderOfRotation` of `create_pose`. For the inverse order, select *'gba'*.

If your robot system uses yet another sequence, you cannot use `create_pose` but must create a corresponding homogeneous transformation matrix and convert it into a pose using `hom_mat3d_to_pose`. If, e.g., your robot system uses the following sequence of rotations where the rotations are performed around the z-axis, then around the y-axis, and finally again around the z-axis

$$\mathbf{R}_{zyz} = \mathbf{R}_z(Rl) \cdot \mathbf{R}_y(Rm) \cdot \mathbf{R}_z(Rr) \tag{8.5}$$

the pose can be created with the following code:

```
hom_mat3d_identity (HomMat3DIdentity)
hom_mat3d_translate (HomMat3DIdentity, Tx, Ty, Tz, HomMat3DTranslate)
hom_mat3d_rotate_local (HomMat3DTranslate, rad(Rl), 'z', HomMat3DT_Rl)
hom_mat3d_rotate_local (HomMat3DT_Rl, rad(Rm), 'y', HomMat3DT_Rl_Rm)
hom_mat3d_rotate_local (HomMat3DT_Rl_Rm, rad(Rr), 'z', HomMat3D)
hom_mat3d_to_pose (HomMat3D, Pose)
```

Note that the rotation operators expect angles to be given in radians, whereas `create_pose` expects them in degrees!

The example program `%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_create_robot_poses.hdev` allows you to enter poses of the three types described above. If your robot system uses yet another sequence of rotations, you can easily extend the program by modifying (or copying and adapting) the code for ZYZ poses.

The HDevelop example programs `%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_movingcam_calibration.hdev` and `%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_stationarycam_calibration.hdev` read the robot pose files in the loop of processing the calibration images.

For each pose of the calibration object, the pose of the robot tool in robot base coordinates that was used for its observation is read from file using `read_pose` and accumulated in the calibration data model `CalibDataID`.

```
read_pose (DataNameStart + 'robot_pose_' + I$'02d' + '.dat', ToolInBasePose)
set_calib_data (CalibDataID, 'tool', I, 'tool_in_base_pose', ToolInBasePose)
```

To be more robust against uncertainties of the provided robot poses use the optimization method *'stochastic'* (see section 8.4.1 on page 180).

The procedure of setting the robot tool pose in the calibration data model is identical for a hand-eye system with a moving camera and a system with a stationary camera.

8.5 Performing the Calibration

Similar to the camera calibration, the main effort lies in collecting the input data. The calibration itself is performed with a single operator call.

```
calibrate_hand_eye (CalibDataID, Errors)
```

Of course, you should check whether the calibration was successful by looking at the output parameter `Errors`, which is a measure for the accuracy of the pose parameters. It contains the pose error of the complete chain of transformations in form of a tuple with the following four elements:

- the root-mean-square error of the translational part in meter
- the root-mean-square error of the rotational part in degree
- the maximum error of the translational part in meter
- the maximum error of the rotational part in degree

In a pose, typically, the translation is entered in meter and the rotation is entered in degree, therefore the respective error has the same unit. The error have to be interpreted in the context of the hand-eye calibration setup, i.e., the size of the robot and the distance between the camera and the calibration object.

Stationary camera

For a hand-eye system with a stationary camera, the pose of the robot base in camera coordinates and the pose of the calibration object in robot tool coordinates is computed by the hand-eye calibration. These poses can be queried from the calibration data model as follows:

```
get_calib_data (CalibDataID, 'camera', 0, 'base_in_cam_pose', BaseInCamPose)
get_calib_data (CalibDataID, 'calib_obj', 0, 'obj_in_tool_pose', \
                ObjInToolPose)
```

Moving camera

For a hand-eye system with a moving camera, the pose of the robot tool in camera coordinates and the pose of the calibration object in robot base coordinates is computed by the hand-eye calibration. These poses can be queried from the calibration data model as follows:

```
get_calib_data (CalibDataID, 'camera', 0, 'tool_in_cam_pose', ToolInCamPose)
get_calib_data (CalibDataID, 'calib_obj', 0, 'obj_in_base_pose', \
                CalObjInBasePose)
```

Typically, you then save the calibrated poses in files so that your robot vision application can read them at a later time. The following code does so for a system with a moving camera:

```
write_pose (ToolInCamPose, DataNameStart + 'final_pose_cam_tool.dat')
write_pose (CalObjInBasePose, \
            DataNameStart + 'final_pose_base_calplate.dat')
```

The example programs then visualize the calibrated poses by displaying the coordinate system of the calibration plate in each calibration image. For this, they compute the pose of the calibration plate in camera coordinates based on the calibrated poses.

Moving camera

For a moving camera system, this corresponds to the following code (compare equation 8.1 on page 178).

```
* CalibObjInCamPose = cam_H_calplate
*                   = cam_H_tool * tool_H_base * base_H_calplate
*                   = ToolInCamPose * BaseInToolPose * CalibrationPose
pose_invert (ToolInBasePose, BaseInToolPose)
pose_compose (ToolInCamPose, BaseInToolPose, BaseInCamPose)
pose_compose (BaseInCamPose, CalibObjInBasePose, CalibObjInCamPose)
```

This code is encapsulated in a procedure, which is called in a loop over all images.

```
for I := 0 to NumImages - 1 by 1
    read_image (Image, ImageNameStart + I$'02d')
```

The pose of the robot tool that was set in the calibration data model is queried and the corresponding pose of the calibration object in the camera coordinates is computed and visualized.

```
    get_calib_data (CalibDataID, 'tool', PoseIds[I], 'tool_in_base_pose', \
                    ToolInBasePose)
    * Compute the pose of the calibration object relative to the camera
    calc_calplate_pose_movingcam (CalObjInBasePose, ToolInCamPose, \
                                  ToolInBasePose, CalObjInCamPose)
    * Display the coordinate system
    disp_3d_coord_system (WindowHandle, CamParam, CalObjInCamPose, 0.01)
endfor
```

Stationary camera

The corresponding procedure for a stationary camera system is listed in appendix A.6 on page 233.

An observation can be deleted using the operator `remove_calib_data_observ` (compare section 3.2.8 on page 75). The corresponding pose of the robot tool has to be deleted using the operator `remove_calib_data`. To determine the effect of a deleted observation on the hand-eye calibration, the calibration has to be performed again.

8.6 Determine Translation in Z Direction for SCARA Robots

This step is not required for articulated robots, but it is obligatory for SCARA robots.

When calibrating SCARA robots, it is not possible to determine the Z translation of 'obj_in_base_pose' (moving camera) or 'obj_in_tool_pose' (stationary camera). To eliminate this ambiguity the Z translation is internally set to 0.0 in these poses and the poses 'tool_in_cam_pose' (moving camera) and 'base_in_cam_pose' (stationary camera), respectively, are calculated accordingly. It is necessary to determine the true translation in Z after the calibration by moving the robot to a pose of known height in the camera coordinate system. For this, the following approach can be applied:

Moving camera

The calibration plate is placed at an arbitrary position. The robot is then moved such that the camera can observe the calibration plate. Now, an image of the calibration plate is acquired and the current robot pose is queried (ToolInBasePose1). From the image, the pose of the calibration plate in the camera coordinate system can be determined (ObjInCamPose1). Afterwards, the tool of the robot is manually moved to the origin of the calibration plate and the robot pose is queried again (ToolInBasePose2). These three poses and the result of the calibration (ToolInCamPosePre) can be used to fix the Z ambiguity by using the following lines of code:

```
pose_invert (ToolInCamPosePre, CamInToolPose)
pose_compose (CamInToolPose, ObjInCamPose1, ObjInToolPose1)
pose_invert (ToolInBasePose1, BaseInToolPose1)
pose_compose (BaseInToolPose1, ToolInBasePose2, Tool2InTool1Pose)
ZCorrection := ObjInToolPose1[2] - Tool2InTool1Pose[2]
set_origin_pose (ToolInCamPosePre, 0, 0, ZCorrection, ToolInCamPose)
```

Stationary camera

A calibration plate (that is not attached to the robot) is placed at an arbitrary position such that it can be observed by the camera. The pose of the calibration plate must then be determined in the camera coordinate system (ObjInCamPose). Afterwards the tool of the robot is manually moved to the origin of the calibration plate and the robot pose is queried (ToolInBasePose). The two poses and the result of the calibration (BaseInCamPosePre) can be used to fix the Z ambiguity by using the following lines of code:

```
pose_invert (BaseInCamPosePre, CamInBasePose)
pose_compose (CamInBasePose, ObjInCamPose, ObjInBasePose)
ZCorrection := ObjInBasePose[2] - ToolInBasePose[2]
set_origin_pose (BaseInCamPosePre, 0, 0, ZCorrection, BaseInCamPose)
```

8.7 Using the Calibration Data

Typically, the result of the hand-eye calibration is used to **transform the results of machine vision from camera coordinates into robot base coordinates** (${}^{cam}\mathbf{H}_{obj} \rightarrow {}^{base}\mathbf{H}_{obj}$) to generate the appropriate robot commands, e.g., to grasp an object whose position has been determined in an image as in the application described in section 8.7.3 on page 187.

Stationary camera

For a stationary camera, this transformation corresponds to the following equation written using homogeneous transformation matrices (compare figure 8.5 on page 178):

$$ {}^{base}\mathbf{H}_{obj} = {}^{base}\mathbf{H}_{cam} \cdot {}^{cam}\mathbf{H}_{obj} \tag{8.6} $$

This equation can also be implemented using the corresponding 3D HALCON poses.

```
pose_invert (BaseInCamPose, CamInBasePose)
pose_compose (CamInBasePose, ObjInCamPose, ObjInBasePose)
```

Moving camera

For a moving camera system, the equation also contains the pose of the robot tool when acquiring the image of the object ${}^{base}\mathbf{H}_{tool}(\text{acq. pos.})$ (compare figure 8.4 on page 178):

$$ {}^{base}\mathbf{H}_{obj} = {}^{base}\mathbf{H}_{tool}(\text{acq. pos.}) \cdot {}^{tool}\mathbf{H}_{cam} \cdot {}^{cam}\mathbf{H}_{obj} \tag{8.7} $$

This equation can also be implemented by composing the corresponding 3D HALCON poses.

```
pose_invert (ToolInCamPose, CamInToolPose)
pose_compose (ToolInBasePose, CamInToolPose, CamInBasePose)
pose_compose (CamInBasePose, ObjInCamPose, ObjInBasePose)
```

8.7.1 Using the Hand-Eye Calibration for Grasping (3D Alignment)

Grasping an object corresponds to a very simple equation that says “move the robot gripper to the pose of the object” (“grasping pose”). This is also called 3D alignment.

Note that if the tool coordinate system used during hand-eye calibration is not placed at the gripper (tool center point), the equation also contains the transformation between tool and gripper coordinate system. This transformation cannot be calibrated with the hand-eye calibration but must be measured or taken from the CAD model or technical drawing of the gripper. Additionally, the procedure `calibrate_robot_touching_point` is available - have a look at the example program `%HALCONEXAMPLES%\hdevelop\Calibration\Hand-Eye\calibrate_hand_eye_stationary_cam_approx.hdev` to see how to use it. To grasp an object, the gripper pose in robot base coordinates has to be identical to the pose of the object in robot base coordinates.

$$ \begin{aligned} \text{tool} = \text{gripper:} \quad & {}^{base}\mathbf{H}_{tool}(\text{grip. pos.}) = {}^{base}\mathbf{H}_{obj} \\ \text{tool} \neq \text{gripper:} \quad & {}^{base}\mathbf{H}_{tool}(\text{grip. pos.}) \cdot {}^{tool}\mathbf{H}_{gripper} = {}^{base}\mathbf{H}_{obj} \\ & {}^{base}\mathbf{H}_{tool}(\text{grip. pos.}) = {}^{base}\mathbf{H}_{obj} \cdot ({}^{tool}\mathbf{H}_{gripper})^{-1} \end{aligned} \tag{8.8} $$

Stationary camera

If we replace ${}^{base}\mathbf{H}_{obj}$ according to equation 8.6, we get the "grasping equation" for a stationary camera:

$$ {}^{base}\mathbf{H}_{tool}(\text{grip. pos.}) = ({}^{base}\mathbf{H}_{cam}) \cdot {}^{cam}\mathbf{H}_{obj} \left[\cdot ({}^{tool}\mathbf{H}_{gripper})^{-1} \right] \tag{8.9} $$

The notation $\left[\cdot ({}^{tool}\mathbf{H}_{gripper})^{-1} \right]$ indicates that this part is only necessary if the tool coordinate system is not identical with the gripper coordinate system.

Accordingly using the HALCON poses, the pose of the robot tool in robot base coordinates (*ToolInBasePose*) that is needed for grasping an object is computed. When the gripper grasps an object GripperInCamPose has to equal:

```
pose_invert(BaseInCamPose, CamInBasePose)
pose_compose(CamInBasePose,ObjInCamPose, GripperInBasePose)
pose_invert (GripperInToolPose, ToolInGripper)
pose_compose (GripperInBasePose,ToolInGripper,ToolInBasePose)
```

Moving camera

For a moving camera, ${}^{base}\mathbf{H}_{obj}$ is replaced by equation 8.7 on page 185 resulting in the following equation:

$$ {}^{base}\mathbf{H}_{tool}(\text{grip. pos.}) = {}^{base}\mathbf{H}_{tool}(\text{acq. pos.}) \cdot ({}^{tool}\mathbf{H}_{cam}) \cdot {}^{cam}\mathbf{H}_{obj} \left[\cdot ({}^{tool}\mathbf{H}_{gripper})^{-1} \right] \tag{8.10} $$

The notation $\left[\cdot ({}^{tool}\mathbf{H}_{gripper})^{-1} \right]$ indicates that this part is only necessary if the tool coordinate system is not identical with the gripper coordinate system.

Using the HALCON poses, the above equation appears as follows:

```
pose_invert(ToolInCamPose, CamInToolPose)
pose_compose(ToolInBasePose_acq,CamInToolPose, CamInBasePose)
pose_invert (GripperInToolPose, ToolInGripper)
pose_compose (CamInBasePose,ToolInGripper,ToolInBasePose_grip)
```

8.7.2 How to Get the 3D Pose of the Object

The 3D pose of the object in camera coordinates (${}^{cam}\mathbf{H}_{obj}$) or more general in sensor coordinates can stem from different sources:

- With a **binocular stereo system** or a generic 3D sensor, you can determine the 3D pose of unknown objects directly. More information on binocular stereo vision is provided in (see chapter 5 on page 117).
- For single camera systems, HALCON provides multiple methods. The most powerful one is **shape-based 3D Matching** (see section 4.2 on page 95 or the Solution Guide I, chapter 11 on page 101), which performs a full object recognition, i.e., it not only estimates the pose but first locates the object in the image. If only one, planar side of the object is visible, fast alternatives are the calibrated perspective deformable matching (section 4.6 on page 114) and the calibrated descriptor-based matching (section 4.7 on page 114). Additionally, you might use simple **shape-based 2D matching**, as demonstrated in the examples `%HALCONEXAMPLES%\hdevelop\Applications\Robot-Vision\pick_and_place_with_2d_matching_moving_cam.hdev` and `%HALCONEXAMPLES%\hdevelop\Applications\Robot-Vision\pick_and_place_with_2d_matching_stationary_cam.hdev`.
- If a full object recognition is not necessary, you can use **pose estimation** to determine the 3D pose of known objects (see section 8.7.3 for an example application and chapter 4 on page 91 for more details on pose estimation).
- Finally, you can determine the 3D coordinates of unknown objects if **object points lie in a known plane** (see section 8.7.3 for an example application and section 3.3 on page 76 for more details on determining 3D points in a known plane).

Please note that if you want to use the 3D pose for grasping the object, the extracted pose, in particular the orientation, must be identical to the pose of the gripper at the grasping position.

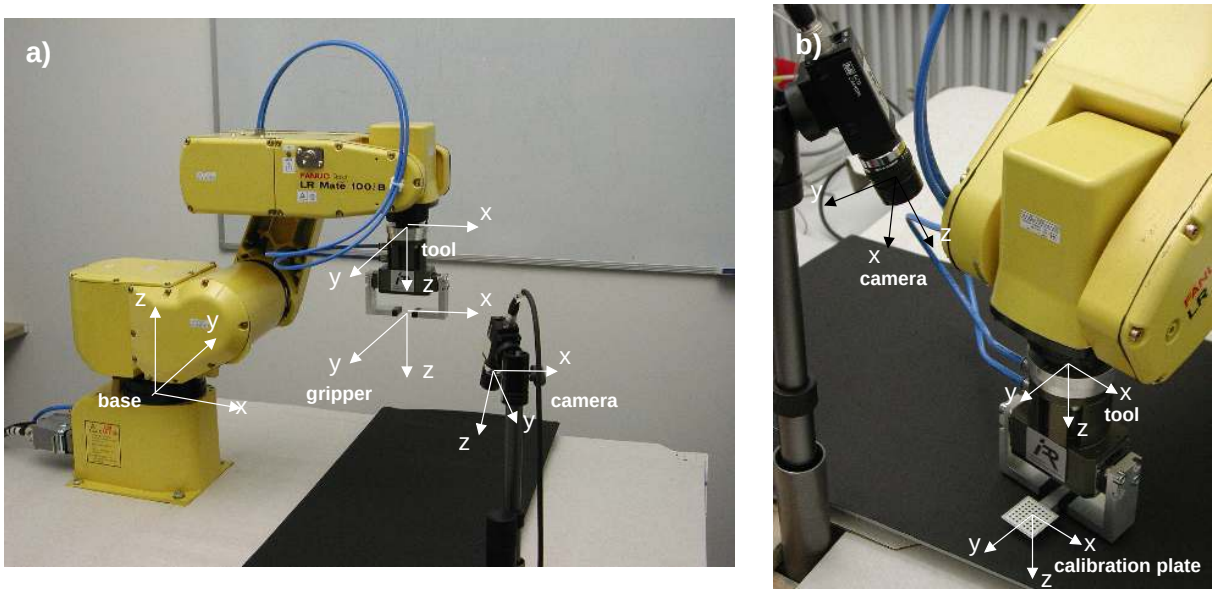


Figure 8.6: Example hand-eye system with a stationary camera: coordinate systems (a) of robot and camera, (b) with calibration plate.

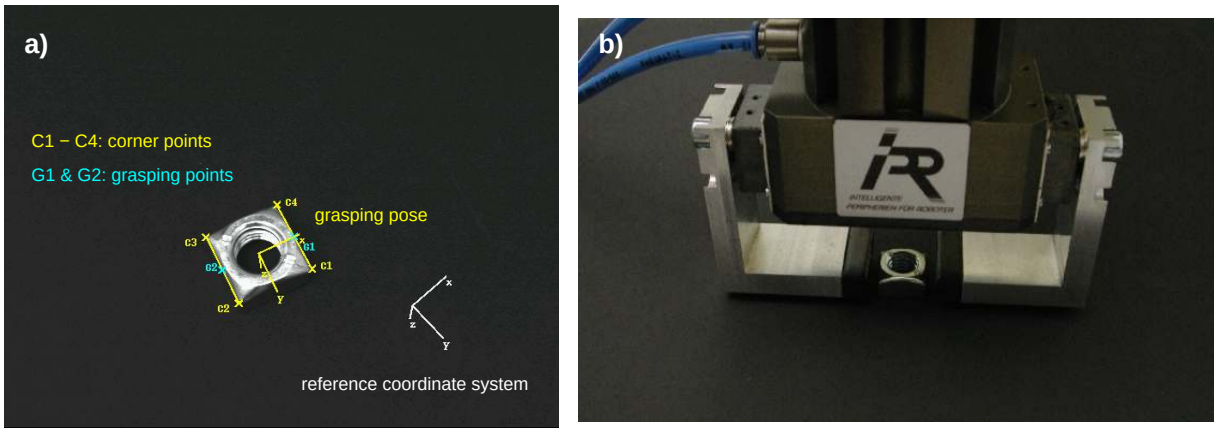


Figure 8.7: (a) Determining the 3D pose for grasping a nut; (b) robot tool at grasping pose.

8.7.3 Example Application with a Stationary Camera: Grasping a Nut

This section describes an example application realized with the hand-eye system depicted in figure 8.6. The task is to localize a nut and determine a suitable grasping pose for the robot tool (see figure 8.7). The HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_stationarycam_grasp_nut.hdev` performs the machine vision part and transforms the resulting pose into robot coordinates using the calibration data determined with `%HALCONEXAMPLES%\solution_guide\3d_vision\hand_eye_stationarycam_calibration.hdev` as described in the previous sections. As you will see, using the calibration data is the shortest part of the program, its main part is devoted to machine vision.

Step 1: Read calibration data

First, the calibrated poses are read from files; for later computations, the poses are converted into homogeneous transformation matrices.

```
read_cam_par (DataNameStart + 'final_campar.dat', CamParam)
read_pose (DataNameStart + 'final_pose_cam_base.dat', BaseInCamPose)
pose_to_hom_mat3d (BaseInCamPose, cam_H_base)
read_pose (DataNameStart + 'final_pose_tool_calplate.dat', \
           CalplateInToolPose)
pose_to_hom_mat3d (CalplateInToolPose, tool_H_calplate)
```

In the used hand-eye system, the tool coordinate system used in the calibration process is located at the mounting point of the tool; therefore, an additional coordinate system is needed between the fingers of the gripper (see figure 8.6a). Its pose in tool coordinates is also read from file.

```
read_pose (DataNameStart + 'pose_tool_gripper.dat', GripperInToolPose)
pose_to_hom_mat3d (GripperInToolPose, tool_H_gripper)
```

Step 2: Define reference coordinate system

Now, a reference coordinate system is defined based on one of the calibration images. In this image, the calibration plate has been placed into the plane on top of the nut. This allows to determine the 3D coordinates of extracted image points on the nut with a single camera and without knowing the dimensions of the nut. The code for defining the reference coordinate system is contained in a procedure, which is listed in appendix A.7 on page 234.

```
define_reference_coord_system (ImageNameStart + 'calib3cm_00', CamParam, \
                               CalplateFile, WindowHandle, PoseRef)
pose_to_hom_mat3d (PoseRef, cam_H_ref)
```

Step 3: Extract grasping points on the nut

The following code extracts grasping points on two opposite sides of the nut. The nut is found with simple blob analysis; its boundary is converted into XLD contours.

```
threshold (Image, BrightRegion, 60, 255)
connection (BrightRegion, BrightRegions)
select_shape (BrightRegions, Nut, 'area', 'and', 500, 99999)
fill_up (Nut, NutFilled)
gen_contour_region_xld (NutFilled, NutContours, 'border')
```

The contours are then processed to find long, parallel straight line segments; their corners are accumulated in tuples.

```
segment_contours_xld (NutContours, LineSegments, 'lines', 5, 4, 2)
fit_line_contour_xld (LineSegments, 'tukey', -1, 0, 5, 2, RowBegin, \
                      ColBegin, RowEnd, ColEnd, Nr, Nc, Dist)
gen_empty_obj (Lines)
for I := 0 to |RowBegin| - 1 by 1
    gen_contour_polygon_xld (Contour, [RowBegin[I],RowEnd[I]], [ColBegin[I], \
                             ColEnd[I]])
    concat_obj (Lines, Contour, Lines)
endfor
gen_polygons_xld (Lines, Polygon, 'ramer', 2)
gen_parallels_xld (Polygon, ParallelLines, 50, 100, rad(10), 'true')
get_parallels_xld (ParallelLines, Row1, Col1, Length1, Phi1, Row2, Col2, \
                   Length2, Phi2)
CornersRow := [Row1[0],Row1[1],Row2[0],Row2[1]]
CornersCol := [Col1[0],Col1[1],Col2[0],Col2[1]]
```

Step 4: Determine the grasping pose in camera coordinates

The grasping pose is calculated in 3D coordinates. For this, the 3D coordinates of the corner points in the reference coordinate system are determined using the operator `image_points_to_world_plane`. The origin of the grasping pose lies in the middle of the corners.

```
image_points_to_world_plane (CamParam, PoseRef, CornersRow, CornersCol, 'm', \
                             CornersX_ref, CornersY_ref)
CenterPointX_ref := sum(CornersX_ref) * 0.25
CenterPointY_ref := sum(CornersY_ref) * 0.25
```

The grasping pose is oriented almost like the reference coordinate system, only rotated around the z-axis so that it is identical to the gripper coordinate system, i.e., so that the gripper "fingers" are parallel to the sides of the nut. To calculate the rotation angle, first the grasping points in the middle of the sides are determined. Their angle can directly be used as the rotation angle.

```
GraspPointsX_ref := [(CornersX_ref[0] + CornersX_ref[1]) * 0.5, \
                     (CornersX_ref[2] + CornersX_ref[3]) * 0.5]
GraspPointsY_ref := [(CornersY_ref[0] + CornersY_ref[1]) * 0.5, \
                     (CornersY_ref[2] + CornersY_ref[3]) * 0.5]
GraspPhiZ_ref := atan((GraspPointsY_ref[1] - GraspPointsY_ref[0]) /  \
                 (GraspPointsX_ref[1] - GraspPointsX_ref[0]))
```

With the origin and rotation angle, the grasping pose is first determined in the reference coordinate system and then transformed into camera coordinates.

```
hom_mat3d_identity (HomMat3DIdentity)
hom_mat3d_rotate (HomMat3DIdentity, GraspPhiZ_ref, 'z', 0, 0, 0, \
                  HomMat3D_RZ_Phi)
hom_mat3d_translate (HomMat3D_RZ_Phi, CenterPointX_ref, CenterPointY_ref, 0, \
                     ref_H_grasp)
hom_mat3d_compose (cam_H_ref, ref_H_grasp, cam_H_grasp)
```

Alternatively, the example also shows how to calculate the grasping pose using pose estimation (see chapter 4 on page 91 for a detailed description). This method can be used when points on the object are known. In the example, we specify the 3D coordinates of the corners of the nut.

```
NX := [0.009, -0.009, -0.009, 0.009]
NY := [0.009, 0.009, -0.009, -0.009]
```

The grasping pose is then calculated by simply calling the operator `vector_to_pose`. Before, however, the image coordinates of the corners must be sorted such that the first one lies close to the x-axis of the reference coordinate system. Otherwise, the orientation of the reference coordinate system would differ too much from the grasping pose and the pose estimation would fail.

```
sort_corner_points (CornersRow, CornersCol, WindowHandle, NRow, NCol)
vector_to_pose (NX, NY, NZ, NRow, NCol, CamParam, 'iterative', 'error', \
                PoseCamNut, Quality)
disp_3d_coord_system (WindowHandle, CamParam, GripperInCamPose, 0.01)
```

The result of both methods is displayed in figure 8.7a on page 187.

Step 5: Transform the grasping pose in robot coordinates

Now comes the moment to use the results of the hand-eye calibration: The grasping pose is transformed into robot coordinates with the formula shown in equation 8.9 on page 186.

```
hom_mat3d_invert (cam_H_base, base_H_cam)
hom_mat3d_compose (base_H_cam, cam_H_grasp, base_H_grasp)
```

As already mentioned, the tool coordinate system used in the calibration process is placed at the mounting point of the tool, not between the fingers of the gripper. Thus, the pose of the tool in gripper coordinates must be added to the chain of transformations to obtain the pose of the tool in base coordinates.

```
hom_mat3d_invert (tool_H_gripper, gripper_H_tool)
hom_mat3d_compose (base_H_grasp, gripper_H_tool, base_H_tool)
```

Step 6: Transform pose type

Finally, the pose is converted into the type used by the robot controller.

```
hom_mat3d_to_pose (base_H_tool, PoseRobotGrasp)
convert_pose_type (PoseRobotGrasp, 'Rp+T', 'abg', 'point', \
                   PoseRobotGrasp_ZYX)
```

Figure 8.7b on page 187 shows the robot at the grasping pose.

Chapter 9

Calibrated Mosaicking

Some objects are too large to be covered by one single image. Multiple images that cover different parts of the object must be taken in such cases. You can measure precisely across the different images if the cameras are calibrated and their external parameters are known with respect to one common world coordinate system.

It is even possible to merge the individual images into one larger image that covers the whole object. This is done by rectifying the individual images with respect to the same measurement plane (see section 3.4.1 on page 80). In the resulting image, you can measure directly in world coordinates.

Note that the 3D coordinates of objects are derived based on the same principle as described in chapter 3 on page 59, i.e., a measurement plane that coincides with the object surface must be defined. Although two or more cameras are used, this is no stereo approach. For more information on 3D vision with a binocular stereo system, please refer to chapter 5 on page 117.

If the resulting image is not intended to serve for high-precision measurements in world coordinates, you can generate it using the mosaicking approach described in chapter 10 on page 205. With this approach, it is not necessary to calibrate the cameras.

A setup for generating a high-precision mosaic image from two cameras is shown in figure 9.1. The cameras are mounted such that the resulting pair of images has a small overlap. The cameras are first calibrated and then the images are merged together into one larger image. All further explanations within this section refer to such a two-camera setup.

Typically, the following two steps must be carried out:

1. Determination of the internal and external camera parameters, using *one* calibration object, to facilitate that the relation between the cameras can be determined.
2. Merge the images into one larger image that covers the whole object.

In this chapter, two approaches, both using a two-camera setup, are described:

The first approach demonstrates, how highly accurate mosaicking can be easily performed after calibrating your cameras simultaneously with a single standard HALCON calibration plate with hexagonally arranged marks. Thereby, in the calibration step, each camera has to be able to obtain adequate images of the calibration plate at the same time. You can also refer to the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Tools\Mosaicking\two_camera_calibrated_mosaicking.hdev`.

The second approach, which is used in the HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\two_camera_calibration.hdev`, shows a calibration setup for cases, where one calibration plate is not sufficient. Thus, a calibration object consisting of multiple calibration plates, whose relative positions are exactly known, is needed.

9.1 Setup

Two or more cameras must be mounted on a *stable* platform such that each image covers a part of the whole scene. The cameras can have an arbitrary orientation, i.e., it is not necessary that they are looking parallel or perpendicular onto the object surface.

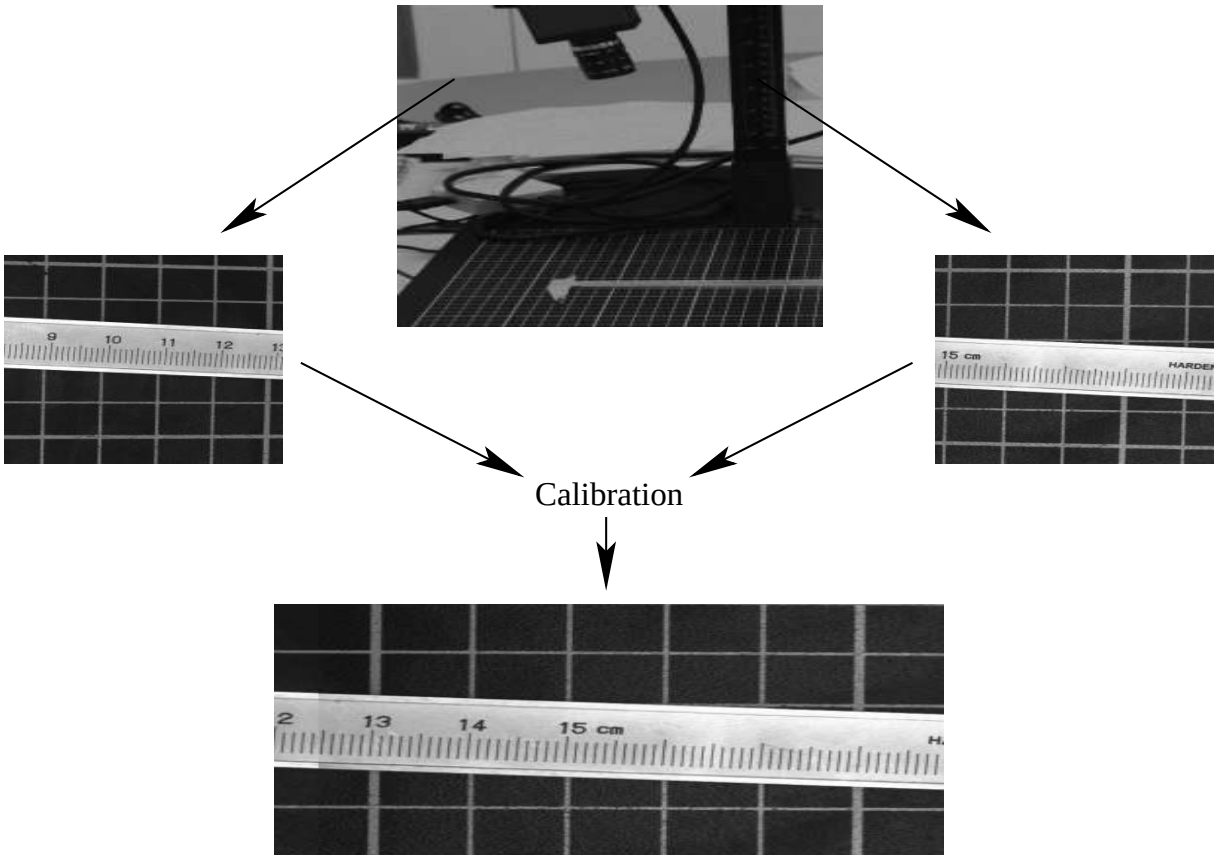


Figure 9.1: Two-camera setup.

To setup focus, illumination, and overlap appropriately, use a big reference object that covers all fields of view. To permit that the images are merged into one larger image, they must have some overlap (see figure 9.2 for an example). The overlapping area can be even smaller than depicted in figure 9.2, since the overlap is only necessary to ensure that there are no gaps in the resulting combined image.

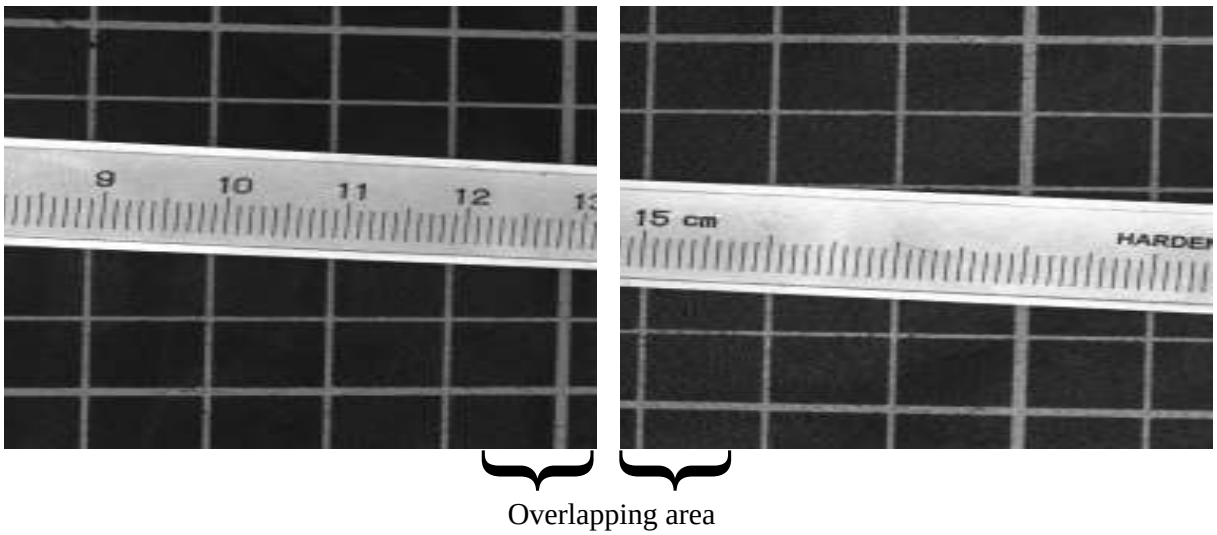


Figure 9.2: Overlapping images.

9.2 Approach Using a Single Calibration Plate

For the first approach, the calibration is performed using a series of images of a single standard HALCON calibration plate with hexagonally arranged marks. Thereby, internal and external camera parameters are determined at once. This approach is quite straightforward, however it only works, if all cameras of a setup can acquire adequate images of the calibration plate simultaneously. Usually this is only the case for a two-camera setup.

The plane, defined by a suitable pose of the calibration plate, that was received during the calibration step, can be used as the rectification plane onto which images from both cameras can then be mapped before they are stitched together.

9.2.1 Calibration

When using a calibration plate with hexagonally arranged marks, the calibration images do not need to contain the entire calibration plate (figure 9.3). Therefore, the following mosaicking approach is suitable, when having a setup where convenient calibration images can be acquired simultaneously by all cameras. See the chapter reference of "Calibration" to get information on how to take a suitable set of calibration images. Both internal and external camera parameters can be determined within this setup.

After setting initial values for the internal camera parameters with `gen_cam_par_area_scan_division`, a calibration data model is created using `create_calib_data`.

```
gen_cam_par_area_scan_division (0.012, 0, 4.4e-6, 4.4e-6, WidthCam1 / 2, \
                                HeightCam1 / 2, WidthCam1, HeightCam1, \
                                StartCamParam1)
gen_cam_par_area_scan_division (0.012, 0, 4.4e-6, 4.4e-6, WidthCam2 / 2, \
                                HeightCam2 / 2, WidthCam2, HeightCam2, \
                                StartCamParam2)
create_calib_data ('calibration_object', 2, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], StartCamParam1)
set_calib_data_cam_param (CalibDataID, 1, [], StartCamParam2)
set_calib_data_calib_object (CalibDataID, 0, 'calplate_160mm.cpd')
```

To achieve high accuracy, ten calibration images are taken by each camera. A simultaneous acquisition of the image pairs ensures, that the calibration plate is in the exact same position referring to world coordinates for both calibration images.

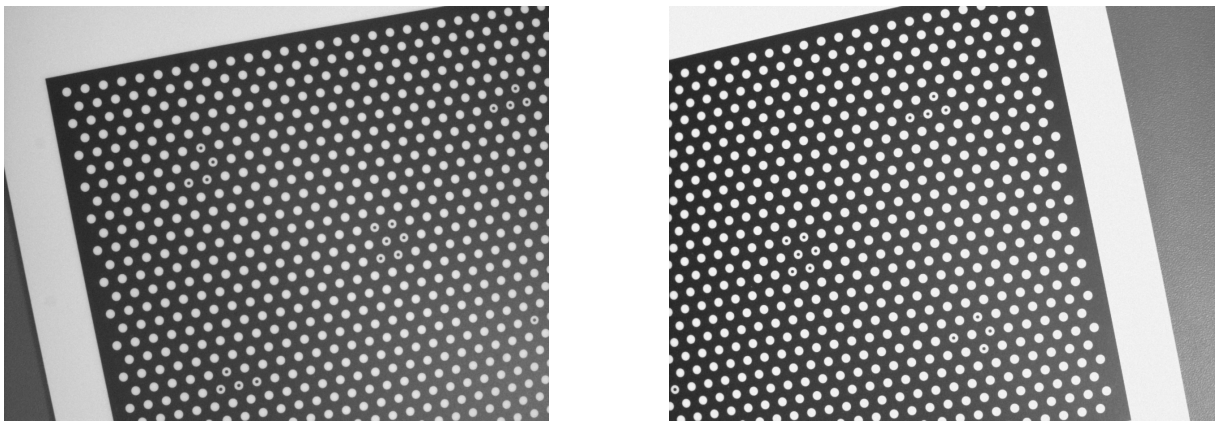


Figure 9.3: Pair of calibration images of the two-camera setup with hexagonally arranged marks.

With `find_calib_object`, the marks of the calibration plate are located and the extracted information is accumulated in the calibration data model for all calibration images. Using `calibrate_cameras` the two cameras can be calibrated simultaneously.

```
for I := 0 to NumCalibImages - 1 by 1
    read_image (ImageCam1, ImagePath + '/calib_cam_1_' + (I + 1)$'02d')
    read_image (ImageCam2, ImagePath + '/calib_cam_2_' + (I + 1)$'02d')
    find_calib_object (ImageCam1, CalibDataID, 0, 0, I, [], [])
    find_calib_object (ImageCam2, CalibDataID, 1, 0, I, [], [])
endfor
calibrate_cameras (CalibDataID, Errors)
```

9.2.2 Mosaicking

After the calibration of the cameras is finished successfully, the setup is now used to acquire images of an object (figure 9.4). In order to stitch images of the two cameras together, they need to be projected into a rectification

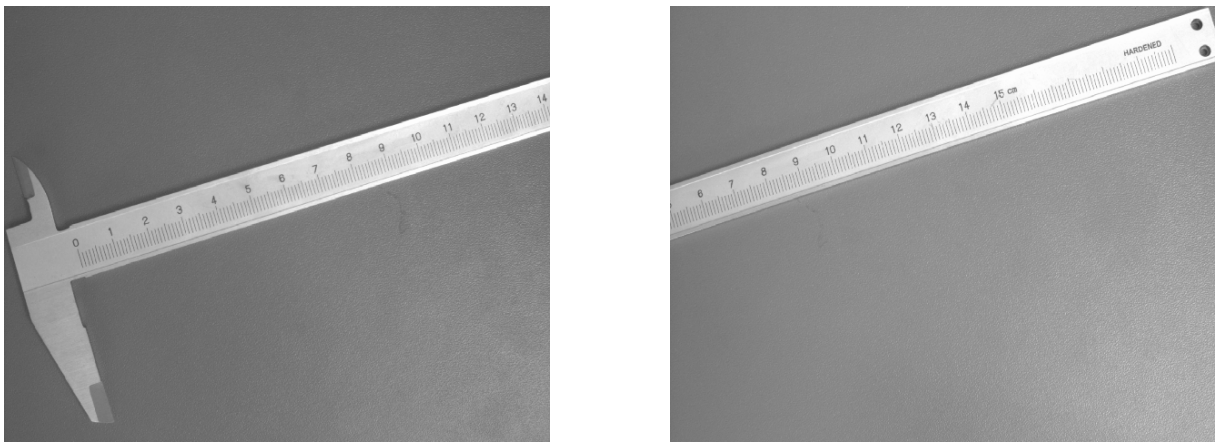


Figure 9.4: Pair of object images taken by the two camera setup.

plane. Therefore, a reference pose of the calibration plate (while positioned in the same plane as the object) with respect to the first camera is used. In order to take the thickness of the calibration plate into account, the z-component of the pose is adjusted by 4mm. Thus, the rectification plane corresponds to the measurement plane. Then, the relative pose of the second camera with respect to the first camera is inverted and combined with the absolute pose of the first camera in order to get its absolute pose.

```
get_calib_data (CalibDataID, 'calib_obj_pose', [0, 0], 'pose', Pose1)
set_origin_pose (Pose1, 0, 0, 0.004, Pose1)
get_calib_data (CalibDataID, 'camera', 1, 'pose', RelPose2)
pose_invert (RelPose2, RelPose2Inverted)
pose_compose (RelPose2Inverted, Pose1, Pose2)
```

Now, after the poses of the rectification plane and the cameras are known, a mapping for the object images is needed. Therefore, the camera poses have to be adjusted by the particular thickness of the respective object. Additionally the x- and y-coordinates are adjusted, so the origin of the rectification plane approximately matches the origin of the result image, hence the object will be contained in it entirely. The mappings are then generated by the operator gen_image_to_world_plane_map.

```
set_origin_pose (Pose1, -0.14, -0.07, HeightCorrections[OIdx], WorldPose1)
set_origin_pose (Pose2, -0.14, -0.07, HeightCorrections[OIdx], WorldPose2)
gen_image_to_world_plane_map (Map1, CamParam1, WorldPose1, WidthCam1, \
                              HeightCam1, TargetWidth, TargetHeight, Scale, \
                              'bilinear')
gen_image_to_world_plane_map (Map2, CamParam2, WorldPose2, WidthCam2, \
                              HeightCam2, TargetWidth, TargetHeight, Scale, \
                              'bilinear')
```

The mappings can be applied with the operator map_image, so both images are projected onto the rectification plane accordingly (see figure 9.5). The mapped images are slightly rotated, as the calibration plate in the image, chosen as reference image in the calibration, was rotated in regard to the image edges as well.

```
map_image (ImageCam1, Map1, ImageWorld1)
map_image (ImageCam2, Map2, ImageWorld2)
```

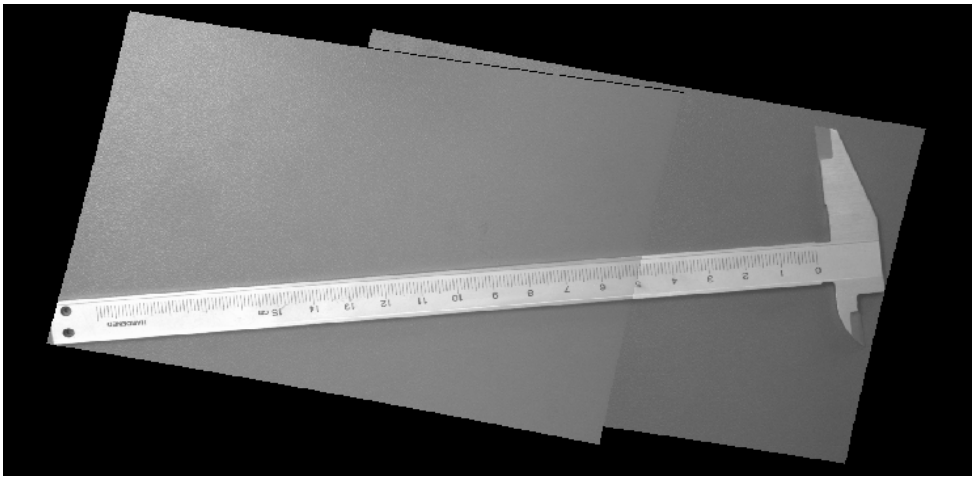


Figure 9.5: Projection of the two images onto the rectification plane.

To get a single result image, the mapped images need to be stitched together. When assembling images of a multi-view camera setup, you need to pay attention to overlapping image parts and the image domains.

```
get_domain (ImageWorld1, Domain1)
get_domain (ImageWorld2, Domain2)
intersection (Domain1, Domain2, RegionIntersection)
paint_region (RegionIntersection, ImageWorld1, ImageWorld1Blackended, 0, \
              'fill')
full_domain (ImageWorld1Blackended, ImagePart1)
full_domain (ImageWorld2, ImagePart2)
add_image (ImagePart1, ImagePart2, ImageFull, 1, 0)
```

By adding the two images, gray values of the overlapping parts are added up. Therefore, in one of the images, the color value of intersecting pixels is set to 0 by using paint_region on the respective image part beforehand. The domain of both individual images is extended, such that it is identical.

The final step is to remove the black borders. Therefore, the full image is rotated so that a rectangle, created by the operator gen_rectangle1, can be fitted in to define the new domain of the image with reduce_domain. This domain is eventually cropped with crop_domain and the image is oriented the favored way by flipping it twice with mirror_image (see final result in figure 9.6).

```
rotate_image (ImageFull, ImageRotated, 12, 'constant')
gen_rectangle1 (RectangleDomain, Borders[0], Borders[1], Borders[2], \
                Borders[3])
reduce_domain (ImageRotated, RectangleDomain, ImageReduced)
crop_domain (ImageReduced, ImageReduced)
mirror_image (ImageReduced, ImageReduced, 'row')
mirror_image (ImageReduced, ImageResult, 'column')
```

9.3 Approach Using Multiple Calibration Plates

The second mosaicking approach is more complex, compared to the first one, but allows you to cover a bigger area, as not all cameras need to have the same calibration plate in their field of view. In fact, in order to perform

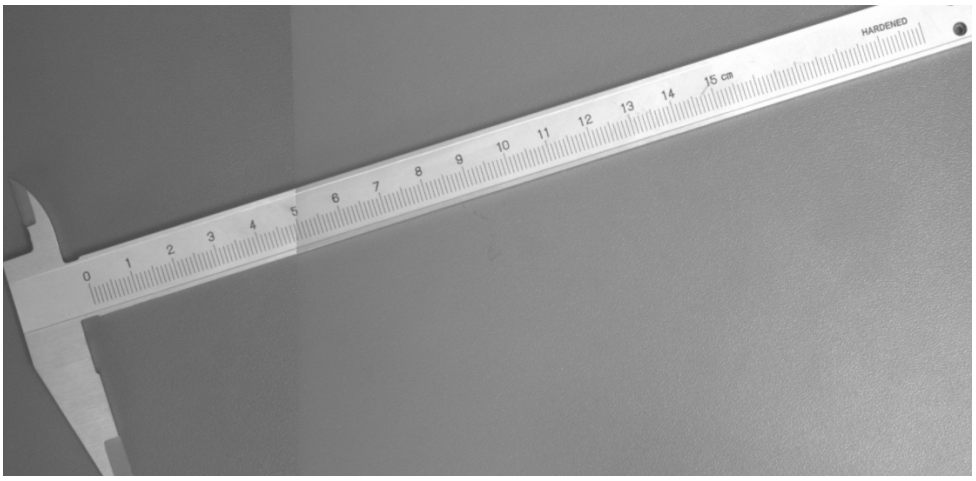


Figure 9.6: Final result of the mosaicking.

a measurement task, the images acquired by the cameras do not necessarily have to overlap. Instead of a single calibration plate, a calibration object consisting of multiple calibration plates is used. Thereby, the relative position of the plates to each other must be known very precisely. Internal and external camera parameters are therefore determined separately.

The idea for this approach is to rectify the two images in such a way, that they fit exactly next to each other. This is first done with images showing the calibration plates in order to create the rectification maps for this setup. Those mappings can then be applied to further image pairs.

9.3.1 Calibration

The calibration of the images can be broken down into two separate steps.

The first step is to determine the internal camera parameters for each of the cameras in use. This can be done for each camera independently, as described in section 3.2 on page 61.

The second step is to determine the external camera parameters for all cameras. Because the final coordinates should refer to *one* world coordinate system for all images, a big calibration object that appears in all images has to be used. For calibration plates with rectangularly arranged marks, which must be completely visible in the image, we propose to use a calibration object like the one displayed in figure 9.7, which consists of as many calibration plates as the number of cameras that are used.

For the determination of the external camera parameters, it is sufficient to use one calibration image from each camera only. Note that the calibration object must not be moved in between the acquisition of the individual images. Ideally, the images are acquired simultaneously. The pose of the calibration plates relative to the cameras is then extracted using an initialized calibration data model and the operators `find_calib_object` and `get_calib_data_observ_points`.

```
CaltabName := 'caltab_30mm.descr'
create_calib_data ('calibration_object', 2, 1, CalibDataID)
set_calib_data_calib_object (CalibDataID, 0, CaltabName)
set_calib_data_cam_param (CalibDataID, 0, [], CamParam1)
set_calib_data_cam_param (CalibDataID, 1, [], CamParam2)
*
* Find and display the calibration plate in the images.
find_calib_object (Image1, CalibDataID, 0, 0, 0, [], [])
get_calib_data_observ_points (CalibDataID, 0, 0, 0, RowCoord1, ColumnCoord1, \
                              Index1, Pose1)
*
find_calib_object (Image2, CalibDataID, 1, 0, 0, [], [])
get_calib_data_observ_points (CalibDataID, 1, 0, 0, RowCoord2, ColumnCoord2, \
                              Index2, Pose2)
```

Here, calibration plates with rectangularly arranged marks are used. The calibration is nevertheless easy if standard HALCON calibration plates mounted on some kind of carrier plate are used such that in each image one calibration plate is completely visible. An example for such a calibration object for a two-camera setup is given in figure 9.7. The respective calibration images for the determination of the external camera parameters are shown in figure 9.8. Note that the relative position of the calibration plates with respect to each other must be known precisely. This can be done with the pose estimation described in chapter 4 on page 91.

Note also that only the relative position of the calibration marks among each other shows the high accuracy stated in section 3.2.3.1 on page 67 but not the borders of the calibration plate. The rows of calibration marks may be slanted with respect to the border of the calibration plate and even the distance of the calibration marks from the border of the calibration plate may vary. Therefore, aligning the calibration plates along their boundaries may result in a shift in x- *and* y-direction with respect to the coordinate system of the calibration plate in its initial position.

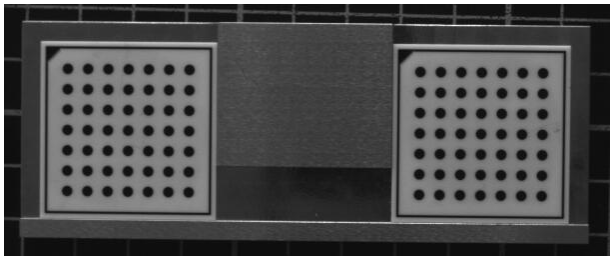


Figure 9.7: Calibration object for two-camera setup.

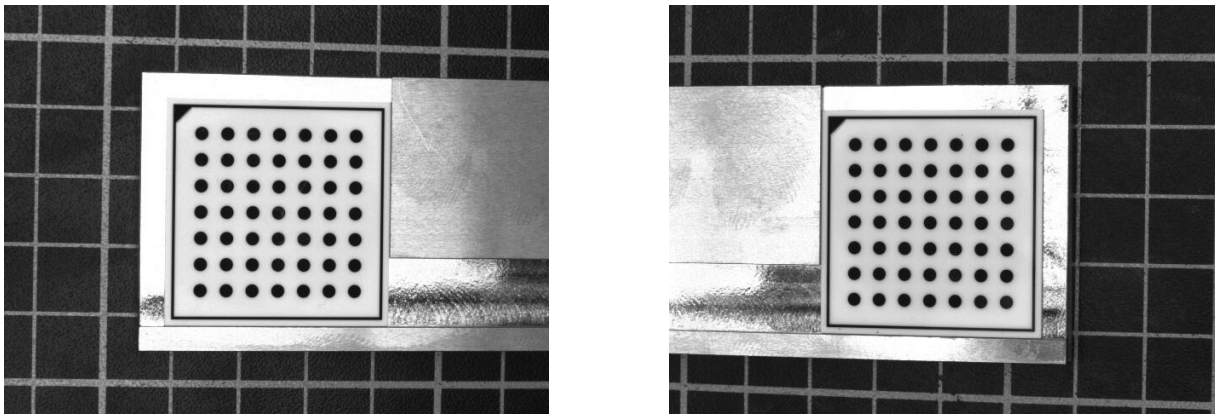


Figure 9.8: Calibration images for two-camera setup.

9.3.2 Merging the Individual Images into One Larger Image

At first, the individual images must be rectified, i.e., transformed so that they exactly fit together. This can be achieved by using the operators `gen_image_to_world_plane_map` and `map_image`. Then, the mosaic image can be generated by the operator `tile_images`, which tiles multiple images into one larger image. These steps are visualized in figure 9.9.

The operators `gen_image_to_world_plane_map` and `map_image` are described in section 3.4.1 on page 80. In the following, we will only discuss the problem of defining the appropriate image detail, i.e., the position of the upper left corner and the size of the rectified images. Again, the description is based on the two-camera setup.

9.3.2.1 Definition of the Rectification of the First Image

For the first (here: left) image, the determination of the necessary shift of the pose is straightforward. You can define the upper left corner of the rectified image in image coordinates, e.g., interactively or, as in the example program, based on a preselected border width.

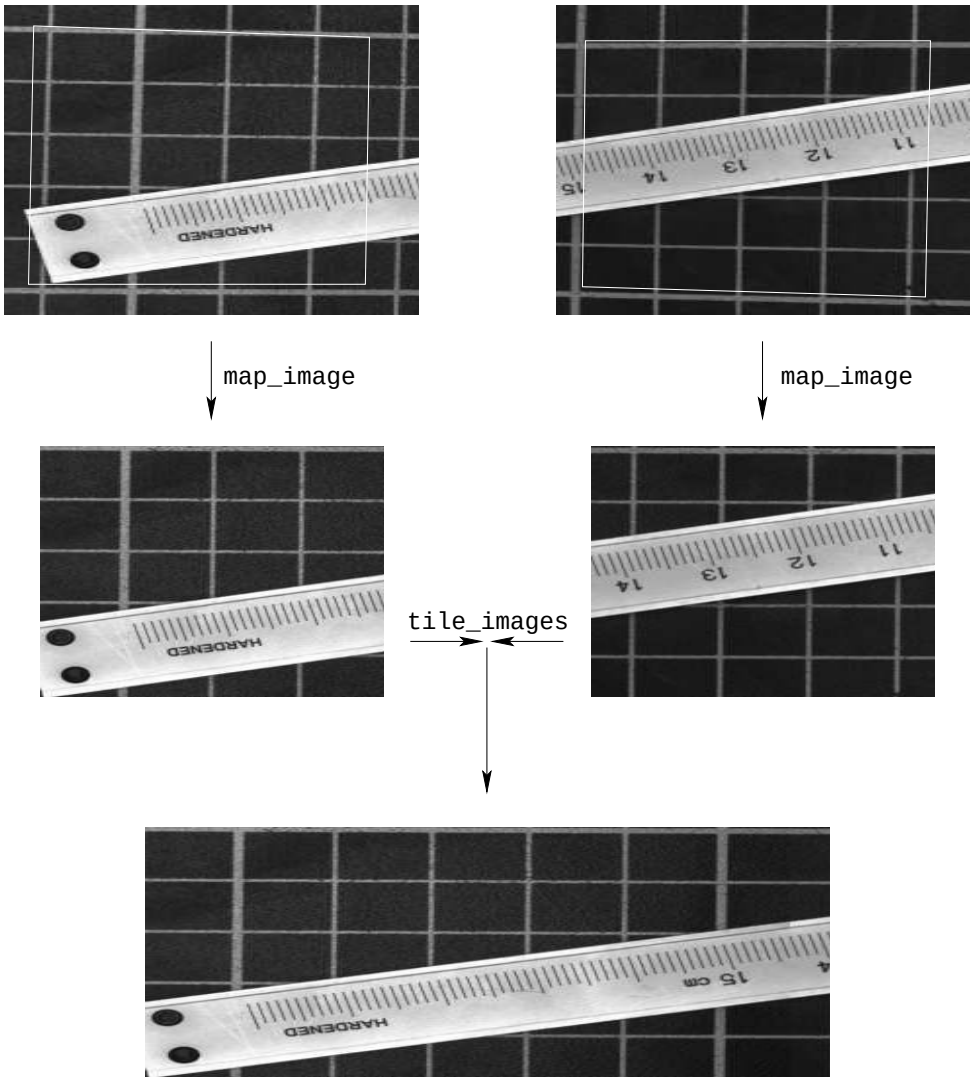


Figure 9.9: Image rectification and tiling.

```
UpperRow := HeightImage1 * BorderInPercent / 100.0
LeftColumn := WidthImage1 * BorderInPercent / 100.0
```

Then, this point must be transformed into world coordinates.

```
image_points_to_world_plane (CamParam1, Pose1, UpperRow, LeftColumn, 'm', \
                             LeftX, UpperY)
```

The resulting coordinates can be used directly, together with the shift that compensates the thickness of the calibration plate (see section 3.2.7.1 on page 73) to modify the origin of the world coordinate system in the left image.

```
set_origin_pose (Pose1, LeftX, UpperY, DiffHeight, PoseNewOrigin1)
```

This means that we shift the origin of the world coordinate system from the origin of the calibration plate to the position that defines the upper left corner of the rectified image (figure 9.10).

The size of the rectified image, i.e., its width and height, can be determined from points originally defined in image coordinates, too. In addition, the desired pixel size of the rectified image must be specified.

```
PixelSize := 0.0001
```

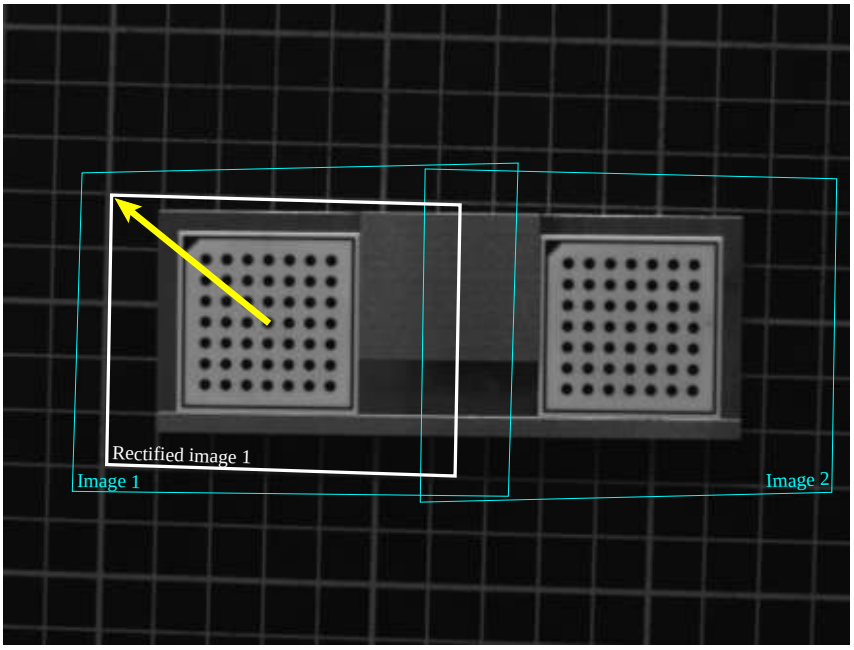


Figure 9.10: Definition of the upper left corner of the first rectified image.

For the determination of the height of the rectified image we need to define a point that lies near the lower border of the first image.

```
LowerRow := HeightImage1 * (100 - BorderInPercent) / 100.0
```

Again, this point must be transformed into the world coordinate system.

```
image_points_to_world_plane (CamParam1, Pose1, LowerRow, LeftColumn, 'm', \
                             X1, LowerY)
```

The height can be determined as the vertical distance between the upper left point and the point near the lower image border, expressed in pixels of the rectified image.

```
HeightRect := int((LowerY - UpperY) / PixelSize)
```

Analogously, the width can be determined from a point that lies in the overlapping area of the two images, i.e., near the right border of the first image.

```
RightColumn := WidthImage1 * (100 - OverlapInPercent / 2.0) / 100.0
image_points_to_world_plane (CamParam1, Pose1, UpperRow, RightColumn, 'm', \
                             RightX, Y1)
WidthRect := int((RightX - LeftX) / PixelSize)
```

Note that the above described definitions of the image points, from which the upper left corner and the size of the rectified image are derived, assume that the x- and y-axes of the world coordinate system are approximately aligned to the column- and row-axes of the first image. This can be achieved by placing the calibration plate in the first image approximately aligned with the image borders. Otherwise, the distances between the above mentioned points make no sense and the upper left corner and the size of the rectified image must be determined in a manner that is adapted for the configuration at hand.

With the shifted pose and the size of the rectified image, the rectification map for the first image can be derived.

```
gen_image_to_world_plane_map (MapSingle1, CamParam1, PoseNewOrigin1, Width, \
                              Height, WidthRect, HeightRect, PixelSize, \
                              'bilinear')
```

9.3.2.2 Definition of the Rectification of the Second Image

The second image must be rectified such that it fits exactly to the right of the first rectified image. This means that the upper left corner of the second rectified image must be identical with the upper right corner of the first rectified image. Therefore, we need to know the coordinates of the upper right corner of the first rectified image in the coordinate system that is defined by the calibration plate in the second image.

First, we express the upper right corner of the first rectified image in the world coordinate system that is defined by the calibration plate in the first image. It can be determined by a transformation from the origin into the upper left corner of the first rectified image (a translation in the example program) followed by a translation along the upper border of the first rectified image. Together with the shift that compensates the thickness of the calibration plate, this transformation is represented by the homogeneous transformation matrix ${}^{cp1}\mathbf{H}_{ur1}$ (see figure 9.11), which can be defined in HDevelop by:

```
hom_mat3d_translate_local (HomMat3DIdentity, LeftX + PixelSize * WidthRect, \
                           UpperY, DiffHeight, cp1Hur1)
```

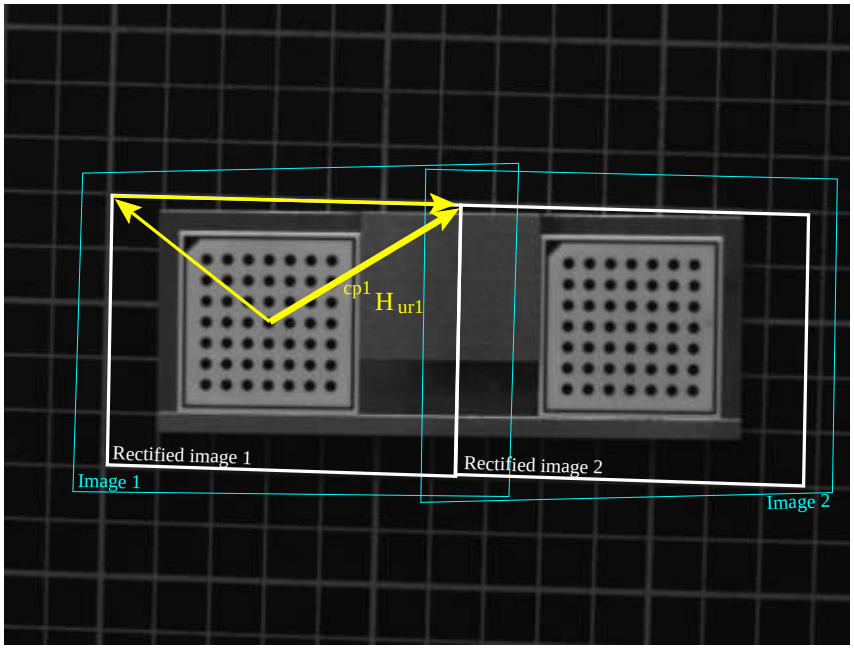


Figure 9.11: Definition of the upper right corner of the first rectified image.

Then, we need the transformation between the two calibration plates of the calibration object. The homogeneous transformation matrix ${}^{cp1}\mathbf{H}_{cp2}$ describes how the world coordinate system defined by the calibration plate in the first image is transformed into the world coordinate system defined by the calibration plate in the second image (figure 9.12). This transformation must be known beforehand from a precise measurement of the calibration object.

From these two transformations, it is easy to derive the transformation that transforms the world coordinate system of the second image such that its origin lies in the upper left corner of the second rectified image. For this, the two transformations have to be combined appropriately (see figure 9.13):

$$
\begin{aligned}
{}^{cp2}\mathbf{H}_{ul2} &= {}^{cp2}\mathbf{H}_{cp1} \cdot {}^{cp1}\mathbf{H}_{ur1} && (9.1)\\
&= ({}^{cp1}\mathbf{H}_{cp2})^{-1} \cdot {}^{cp1}\mathbf{H}_{ur1} && (9.2)
\end{aligned}
$$

This can be implemented in HDevelop as follows:

```
hom_mat3d_invert (cp1Hcp2, cp2Hcp1)
hom_mat3d_compose (cp2Hcp1, cp1Hur1, cp2Hul2)
```

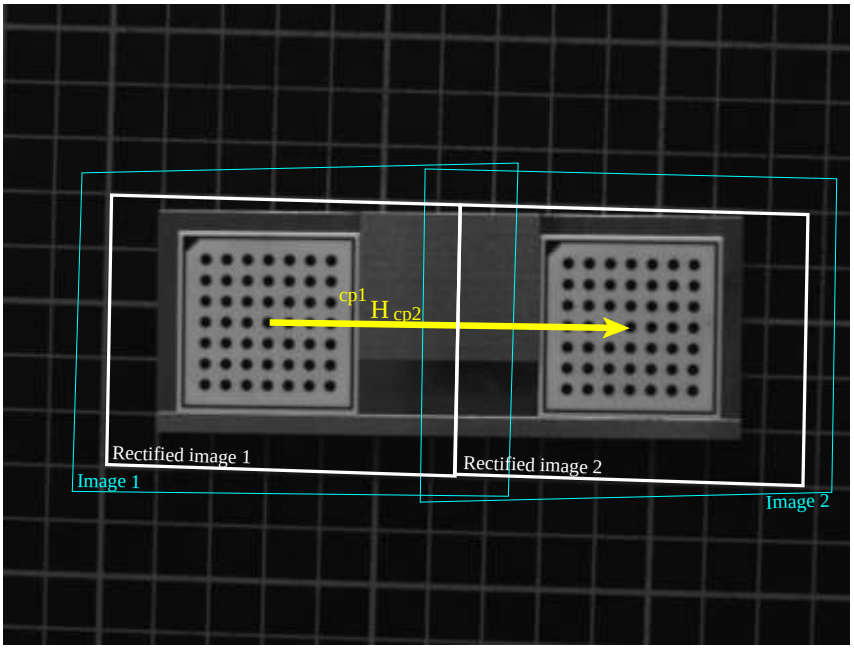


Figure 9.12: Transformation between the two world coordinate systems, each defined by the respective calibration plate.

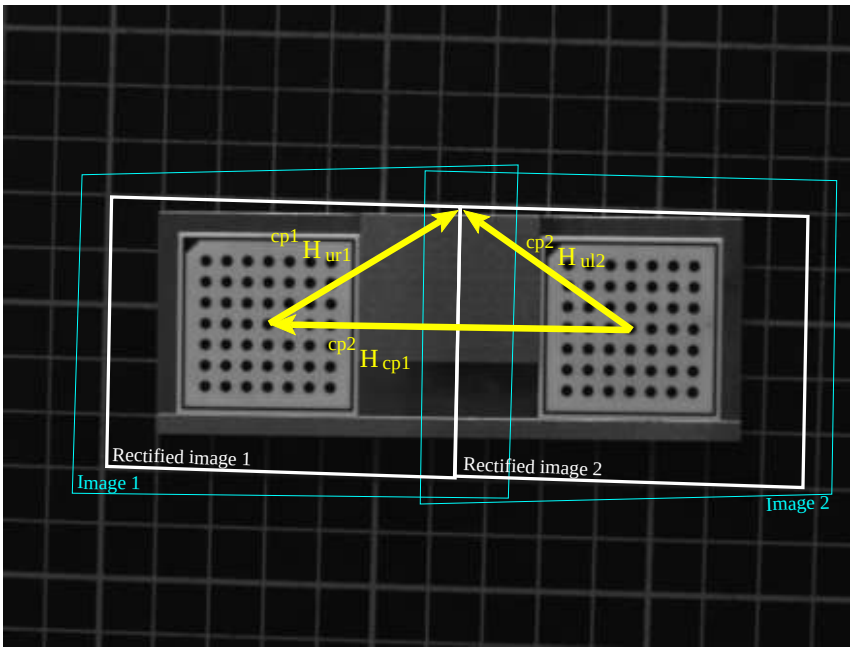


Figure 9.13: Definition of the upper left corner of the second rectified image.

With this, the pose of the calibration plate in the second image can be modified such that the origin of the world coordinate system lies in the upper left corner of the second rectified image.

```
pose_to_hom_mat3d (Pose2, cam2Hcp2)
hom_mat3d_compose (cam2Hcp2, cp2Hul2, cam2Hul2)
hom_mat3d_to_pose (cam2Hul2, PoseNewOrigin2)
```

With the resulting new pose and the size of the rectified image, which can be the same as for the first rectified image, the rectification map for the second image can be derived.

```
gen_image_to_world_plane_map (MapSingle2, CamParam2, PoseNewOrigin2, Width, \
                              Height, WidthRect, HeightRect, PixelSize, \
                              'bilinear')
```

9.3.2.3 Rectification of the Images

Once the rectification maps are created, every image pair from the two-camera setup can be rectified and tiled very efficiently. The resulting mosaic image consists of the two rectified images and covers a part as indicated in figure 9.14.

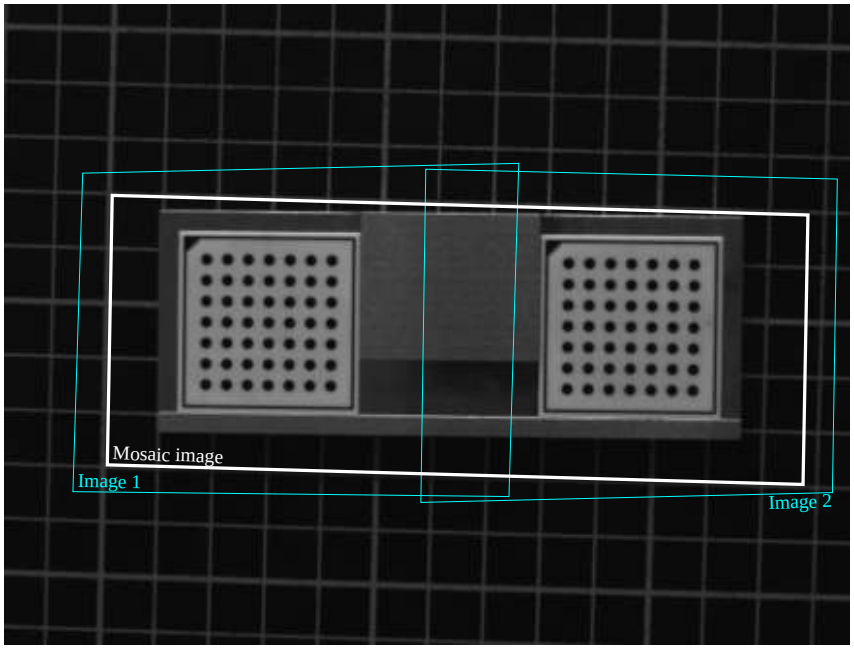


Figure 9.14: The position of the final mosaic image.

The rectification is carried out by the operator `map_image`.

```
map_image (Image1, MapSingle1, RectifiedImage1)
map_image (Image2, MapSingle2, RectifiedImage2)
```

This transforms the two images displayed in figure 9.15, into the two rectified images that are shown in figure 9.16.

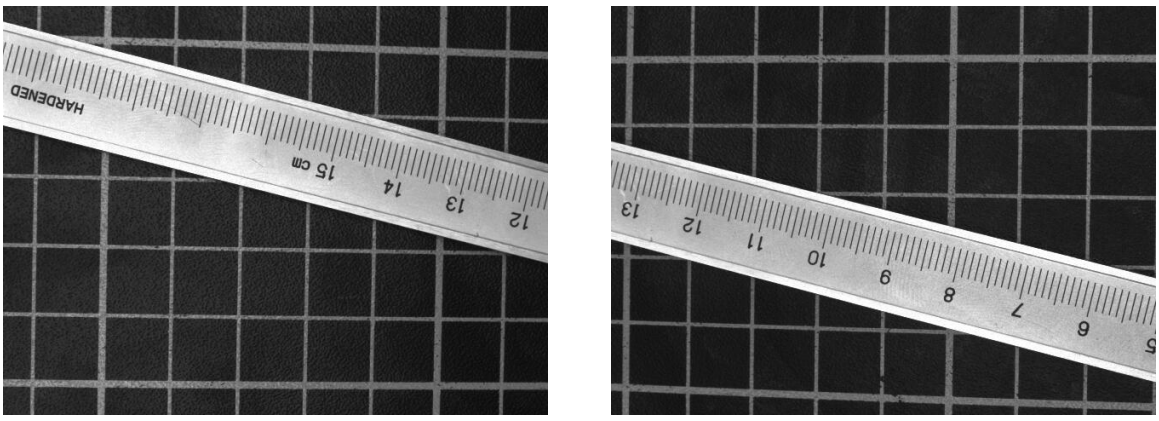


Figure 9.15: Two test images acquired with the two-camera setup.

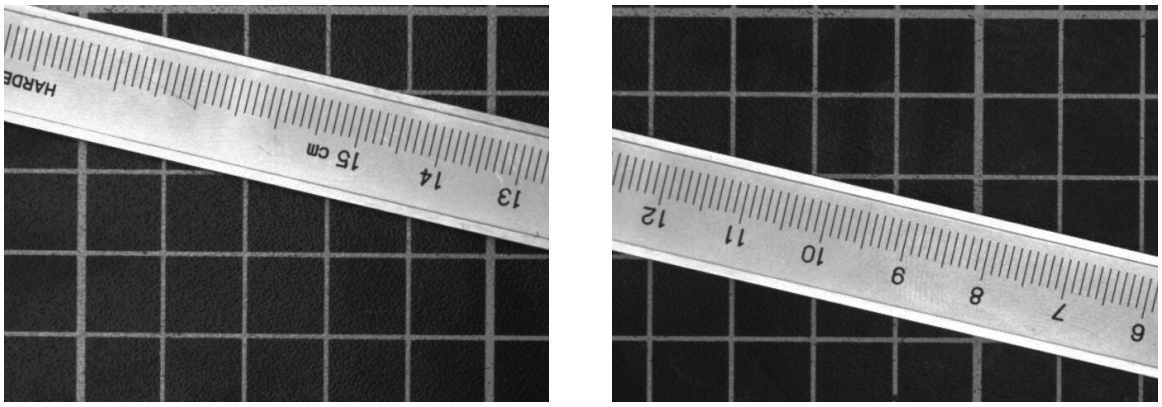


Figure 9.16: Rectified images.

As a preparation for the tiling, the rectified images must be concatenated into one tuple, which then contains both images.

```
concat_obj (RectifiedImage1, RectifiedImage2, Concat)
```

Then the two images can be tiled.

```
tile_images (Concat, Combined, 2, 'vertical')
```

The resulting mosaic image is displayed in figure 9.17.

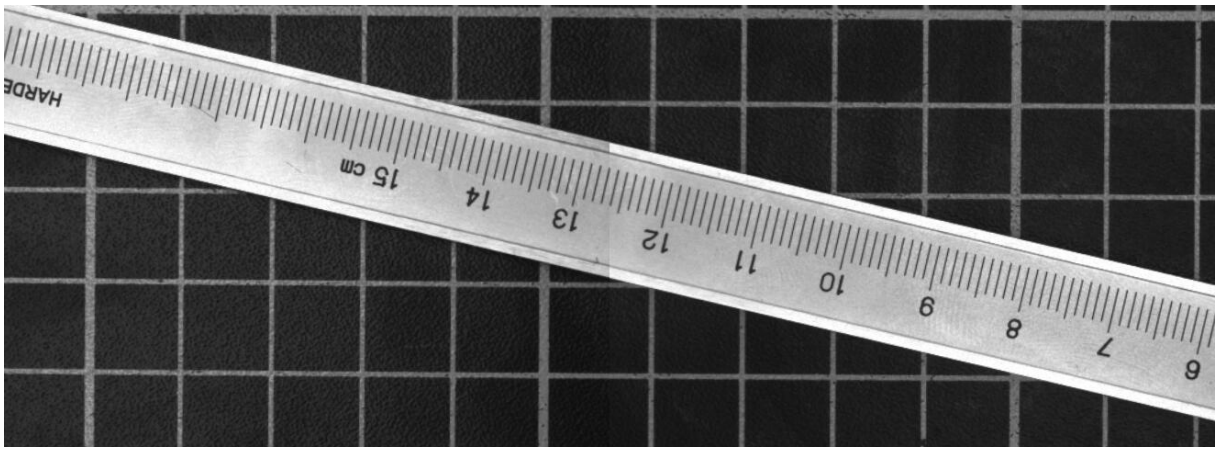


Figure 9.17: Mosaic image.

Chapter 10

Uncalibrated Mosaicking

If you need an image of a large object, but the field of view of the camera does not allow to cover the entire object with the desired resolution, you can use image mosaicking to generate a large image of the entire object from a sequence of overlapping images of parts of the object.

An example for such an application is given in figure 10.1. On the left side, six separate images are displayed stacked upon each other. On the right side, the mosaic image generated from the six separate images is shown. Note that the folds visible in the image do not result from the mosaicking. They are due to some degradations on the PCB, which can be seen already in the separate images.

The mosaicking approach described in this section is designed for applications where it is not necessary to achieve the high-precision mosaic images as described in chapter 9 on page 191. The advantages compared to this approach are that no camera calibration is necessary and that the individual images can be arranged automatically.

The example program `%HALCONEXAMPLES%\solution_guide\3d_vision\mosaicking.hdev` generates the mosaic image displayed in figure 10.7 on page 210. First, the images are read from file and collected in one tuple.

```
gen_empty_obj (Images)
for J := 1 to 10 by 1
    read_image (Image, ImgPath + ImgName + J$'02')
    concat_obj (Images, Image, Images)
endfor
```

Then, the image pairs must be defined, i.e., which image should be mapped to which image.

```
    From := [1, 2, 3, 4, 6, 7, 8, 9, 3]
    To := [2, 3, 4, 5, 7, 8, 9, 10, 8]
```

Now, characteristic points must be extracted from the images, which are then used for the matching between the image pairs. The resulting projective transformation matrices[1] must be accumulated.

[1] A projective transformation matrix describes a perspective projection. It consists of 3×3 values. If the last row contains the values [0,0,1], it corresponds to a homogeneous transformation matrix of HALCON and therefore describes an affine transformation.

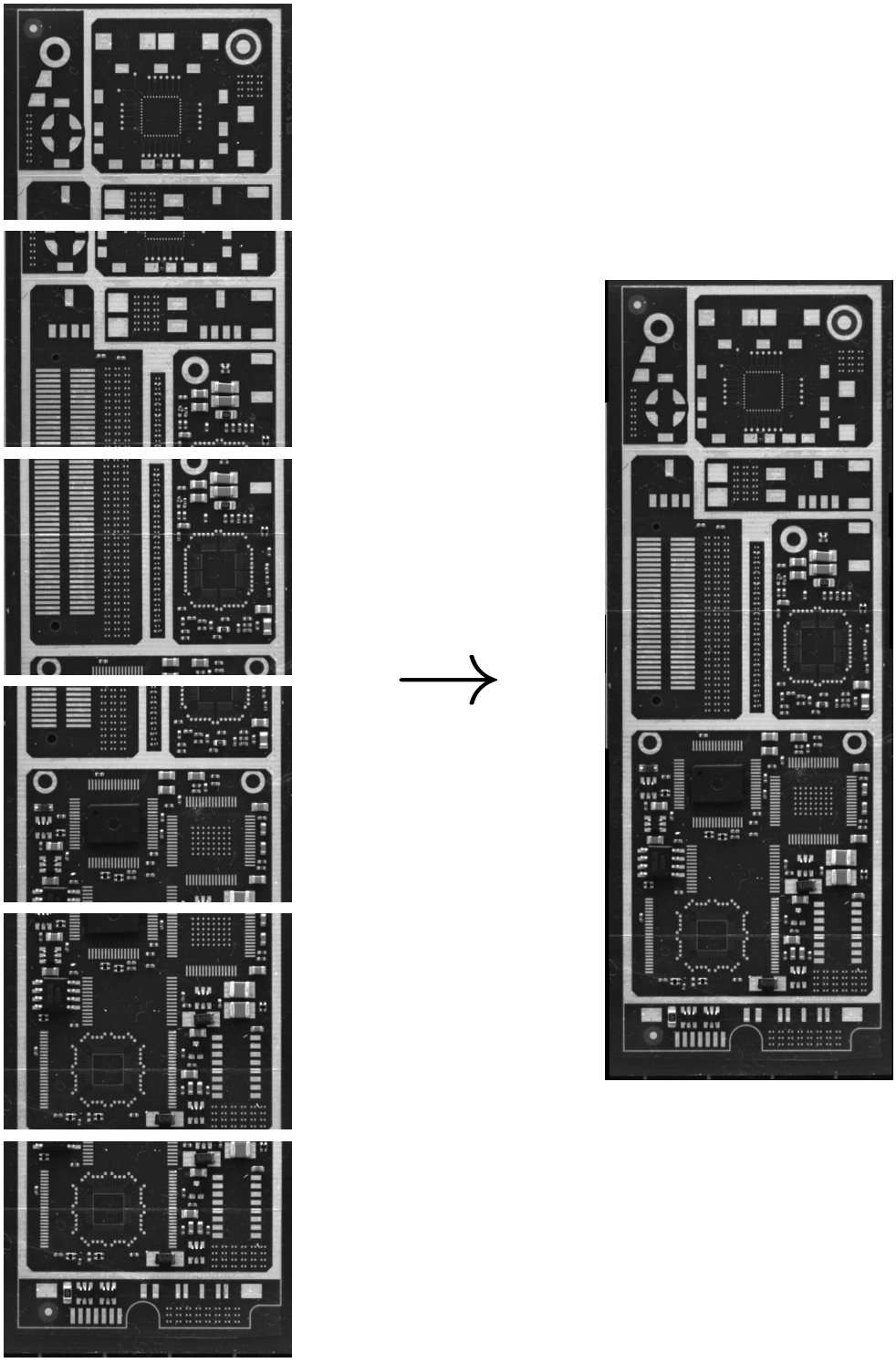


Figure 10.1: A first example for image mosaicking.

```
Num := |From|
ProjMatrices := []
for J := 0 to Num - 1 by 1
    F := From[J]
    T := To[J]
    select_obj (Images, ImageFrom, F)
    select_obj (Images, ImageTo, T)
    points_harris (ImageFrom, SigmaGrad, SigmaSmooth, Alpha, Threshold, \
                   RowFromAll, ColumnFromAll)
    points_harris (ImageTo, SigmaGrad, SigmaSmooth, Alpha, Threshold, \
                   RowToAll, ColumnToAll)
    proj_match_points_ransac (ImageFrom, ImageTo, RowFromAll, ColumnFromAll, \
                               RowToAll, ColumnToAll, 'sad', MaskSize, \
                               RowMove, ColumnMove, RowTolerance, \
                               ColumnTolerance, Rotation, MatchThreshold, \
                               'gold_standard', DistanceThreshold, RandSeed, \
                               ProjMatrix, Points1, Points2)
    ProjMatrices := [ProjMatrices,ProjMatrix]
disp_message (WindowHandle1, 'Point matches', 'window', 12, 12, 'black', \
              'true')
```

Finally, the image mosaic can be generated.

```
gen_projective_mosaic (Images, MosaicImage, StartImage, From, To, \
                       ProjMatrices, StackingOrder, 'false', \
                       MosaicMatrices2D)
```

Note that image mosaicking is a tool for a quick and easy generation of large images from several overlapping images. For this task, it is not necessary to calibrate the camera. If you need a high-precision image mosaic, you should use the method described in chapter 9 on page 191.

In the following sections, the individual steps for the generation of a mosaic image are described.

10.1 Rules for Taking Images for a Mosaic Image

The following rules for the acquisition of the separate images should be considered:

- The images must overlap each other.
- The overlapping area of the images must be textured in order to allow the automatic matching process to identify identical points in the images. The lack of texture in some overlapping areas may be overcome by an appropriate definition of the image pairs (see section 10.2). If the whole object shows little texture, the overlapping areas should be chosen larger.
- Overlapping images must have approximately the same scale. In general, the scale differences should not exceed 5-10 %.
- The images should be radiometrically similar, at least in the overlapping areas, as no radiometric adaption of the images is carried out. Otherwise, i.e., if the brightness differs heavily between neighboring images, the seams between them will be clearly visible as can be seen in figure 10.2.

The images are mapped onto a common image plane using a projective transformation. Therefore, to generate a geometrically accurate image mosaic from images of non-flat objects, the separate images must be acquired from approximately the same point of view, i.e., the camera can only be rotated around its optical center (see figure 10.3).

When dealing with flat objects, it is possible to acquire the images from arbitrary positions and with arbitrary orientations if the scale difference between the overlapping images is not too large (figure 10.4).

The lens distortions of the images are not compensated by the mosaicking process. Therefore, if lens distortions are present in the images, they cannot be mosaicked with high accuracy, i.e., small distortions at the seams between

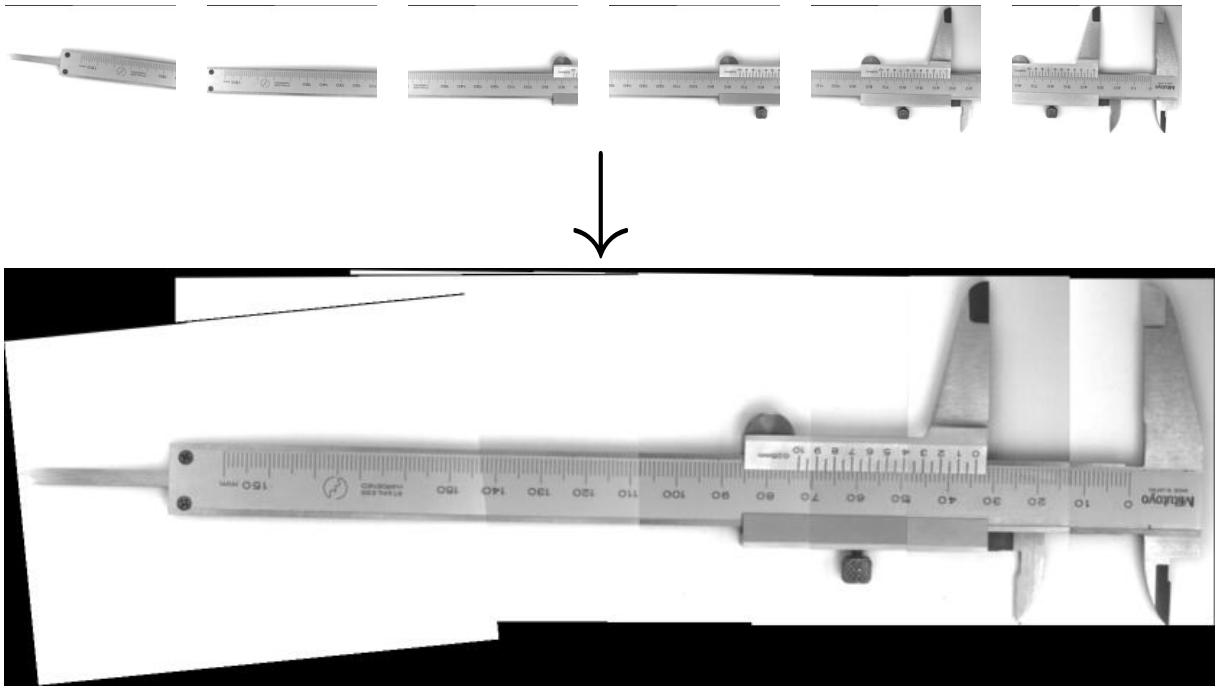


Figure 10.2: A second example for image mosaicking.

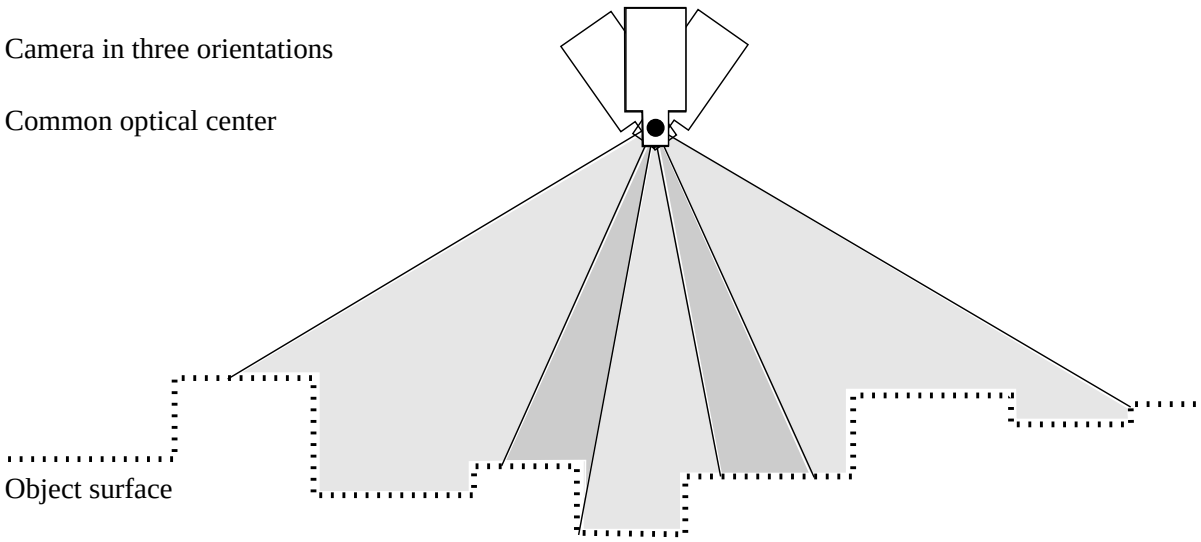


Figure 10.3: Image acquisition for non-flat objects.

neighboring images cannot be prevented (see figure 10.8 on page 211). To eliminate this effect, the lens distortions can be compensated before starting the mosaicking process (see section 3.4.2 on page 86).

If processing time is an issue, it is advisable to acquire the images in the same orientation, i.e., neither the camera nor the object should be rotated around the optical axis, too much. Then, the rotation range can be restricted for the matching process (see section 10.4 on page 213).

10.2 Definition of Overlapping Image Pairs

As shown in the introductory example, it is necessary to define the overlapping image pairs between which the transformation is to be determined. The successive matching process will be carried out for these image pairs only.

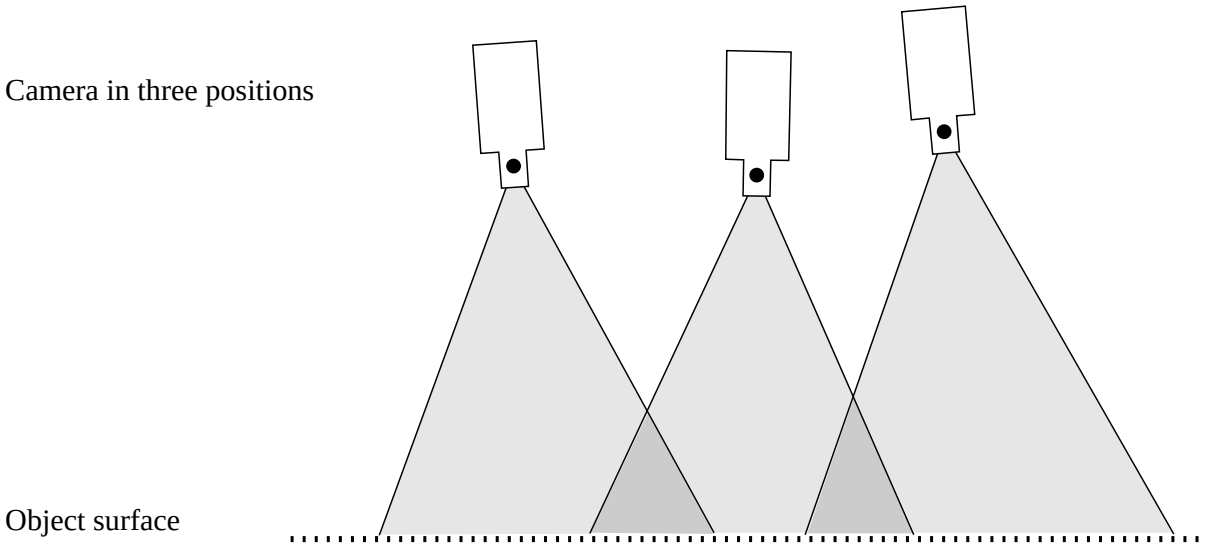


Figure 10.4: Image acquisition for flat objects.

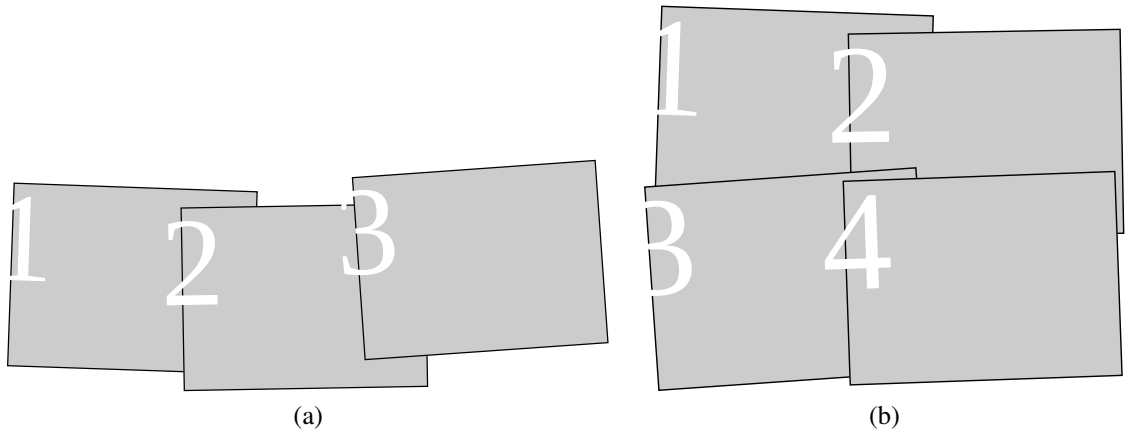


Figure 10.5: Two configurations of overlapping images.

Figure 10.5 shows two configurations of separate images. For configuration (a), the definition of the image pairs is simply (1,2) and (2,3), which can be defined in HDevelop as:

```
From := [1,2]
To := [2,3]
```

In any case, it is important to ensure that each image must be "connected" to all the other images. For example, for configuration (b) of figure 10.5, it is not possible to define the image pairs as (1,2) and (3,4), only, because images 1 and 2 would not be connected to images 3 and 4. In this case, it would, e.g., be possible to define the three image pairs (1,2), (1,3), and (2,4).

```
From := [1,1,2]
To := [2,3,4]
```

Assuming there is no texture in the overlapping area of image two and four, the matching could be carried out between images three and four instead.

```
From := [1,1,3]
To := [2,3,4]
```

If a larger number of separate images are mosaicked, or, e.g., an image configuration similar to the one displayed in figure 10.6, where there are elongated rows of overlapping images, it is important to thoroughly arrange the image

pair configuration. Otherwise it is possible that some images do not fit together precisely. This happens since the transformations between the images cannot be determined with perfect accuracy because of very small errors in the point coordinates due to noise. These errors are propagated from one image to the other.

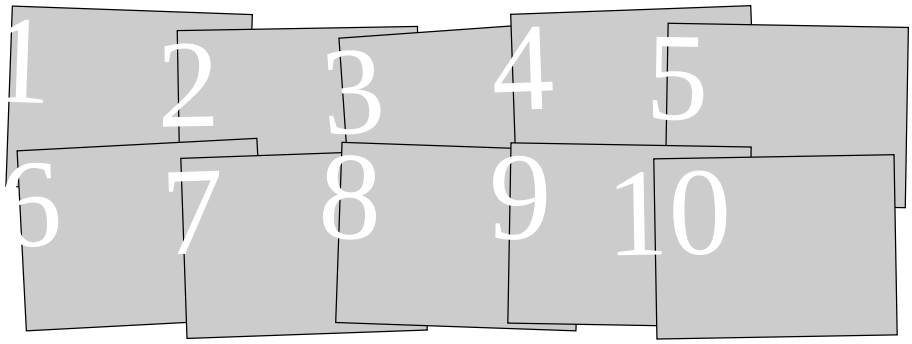


Figure 10.6: A configuration of ten overlapping images.

Figure 10.7 shows such an image sequence of ten images of a BGA and the resulting mosaic image. Figure 10.8 shows a cut-out of that mosaic image. It depicts the seam between image 5 and image 10 for two image pair configurations, using the original images and the images where the lens distortions have been eliminated, respectively. The position of the cut-out is indicated in figure 10.7 by a rectangle.

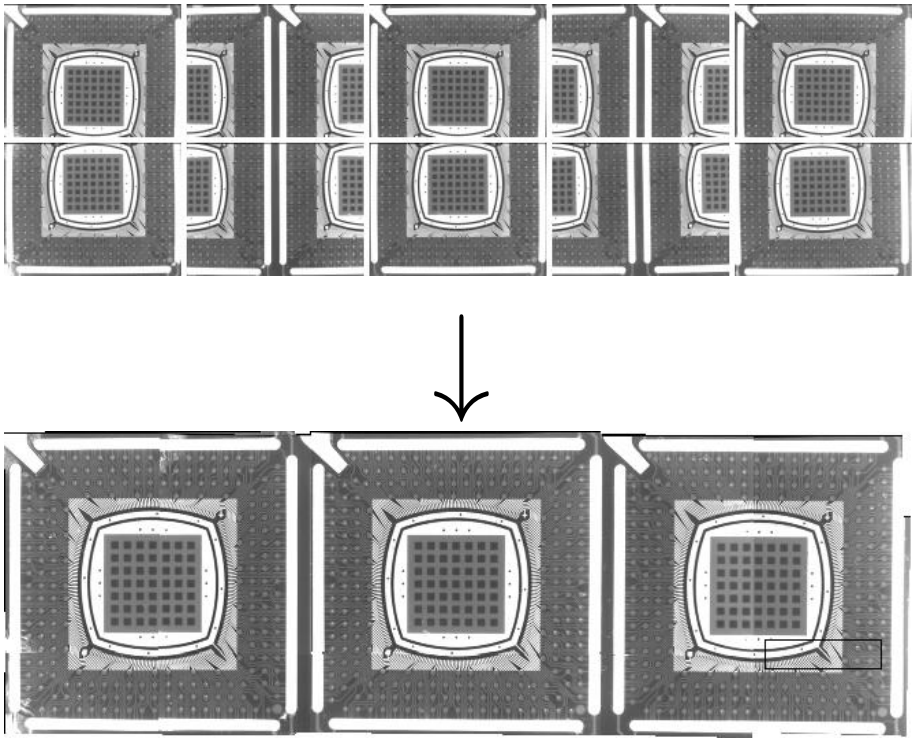


Figure 10.7: Ten overlapping images and the resulting (rigid) mosaic image.

First, the matching has been carried out in the two image rows separately and the two rows are connected via image pair $1 \rightarrow 6$.

```
From := [1,2,3,4,6,7,8,9,1]
To :=  [2,3,4,5,7,8,9,10,6]
```

In this configuration the two neighboring images 5 and 10 are connected along a relatively long path (figure 10.9).

To improve the geometrical accuracy of the image mosaic, the connections between the two image rows could be established by the image pair (3,8), as visualized in (figure 10.10)).

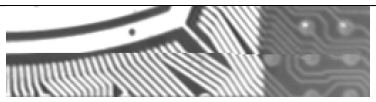
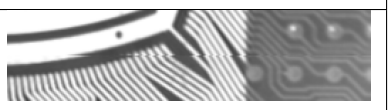
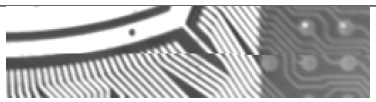
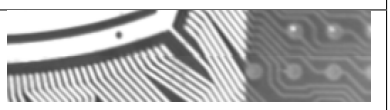
	with lens distortions	lens distortions eliminated
unfavorable configuration		
good configuration		

Figure 10.8: Seam between image 5 and image 10 for various configurations.

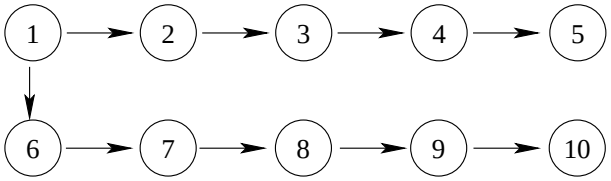


Figure 10.9: Unfavorable configuration of image pairs.

This can be achieved by defining the image pairs as follows.

```
From := [1,2,3,4,6,7,8,9,3]
To := [2,3,4,5,7,8,9,10,8]
```

As can be seen in figure 10.8, now the neighboring images fit better.

Recapitulating, there are three basic rules for the arrangement of the image pairs:

Take care that

1. each image is connected to all the other images.
2. the path along which neighboring images are connected is not too long.
3. the overlapping areas of image pairs are large enough and contain enough texture to ensure a proper matching.

In principle, it is also possible to define more image pairs than required (number of images minus one). However, then it cannot be controlled which pairs are actually used. Therefore, we do not recommend this approach.

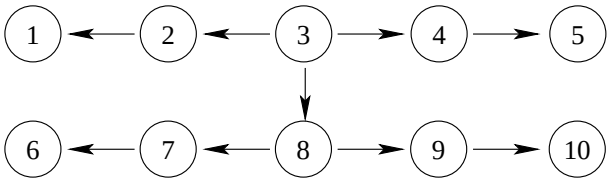


Figure 10.10: Good configuration of image pairs.

10.3 Detection of Characteristic Points

HALCON provides you with various operators for the extraction of characteristic points (interest points). The most important of these operators are

- points_foerstner
- points_harris and points_harris_binomial
- points_lepetit
- points_sojka
- saddle_points_sub_pix

All of these operators can determine the coordinates of interest points with subpixel accuracy.

In figure 10.11, a test image together with typical results of these interest operators is displayed.

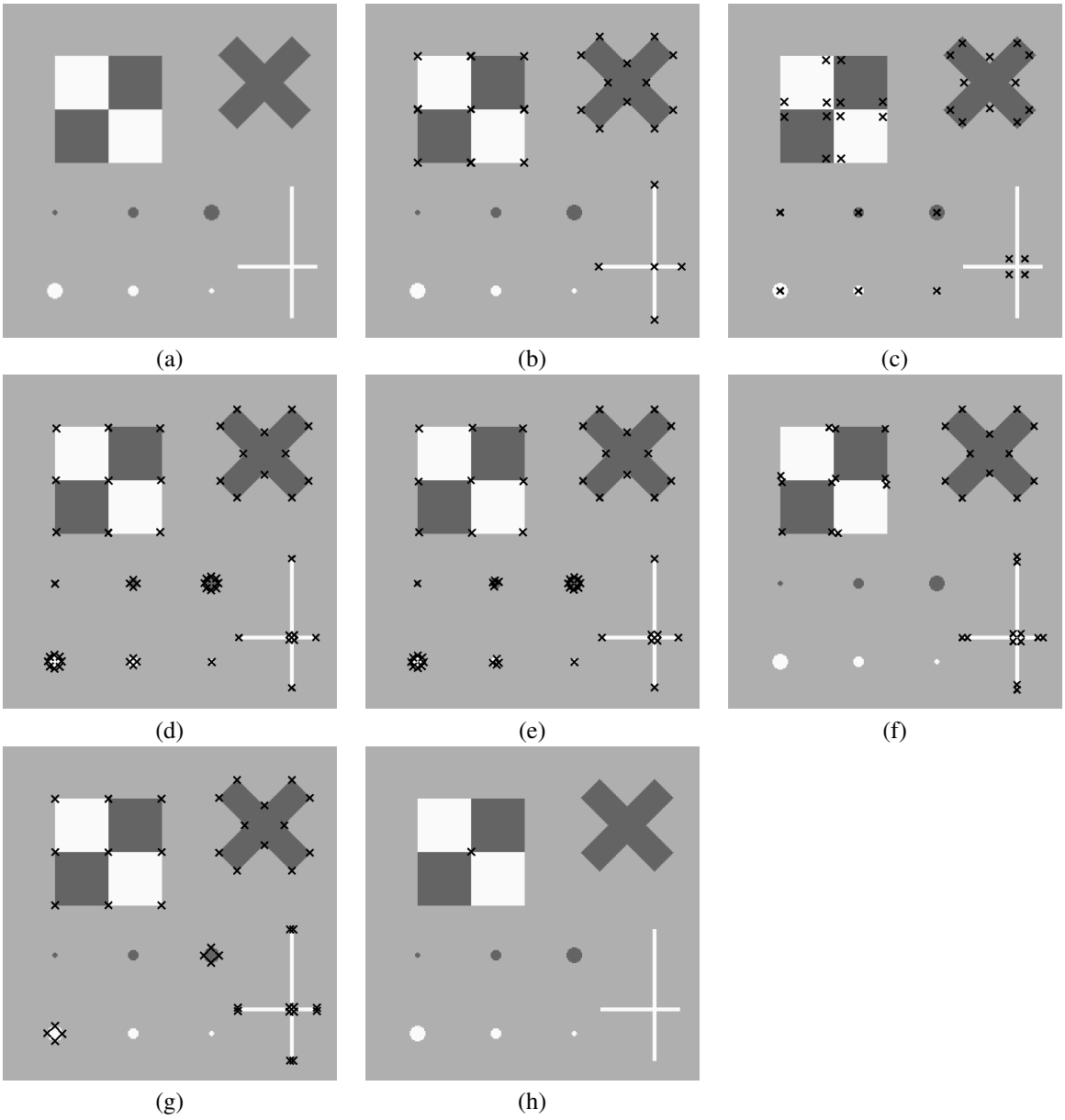


Figure 10.11: Comparison of typical results of interest operators. a) Test image; b) Förstner, junctions; c) Förstner, area; d) Harris; e) Harris, binomial f) Lepetit; g) Sojka; h) Saddle points.

The operator points_foerstner classifies the interest points into two categories: junction-like features and area-like features. The results are very reproducible even in images taken from a different point of view. Therefore, it is very well suited for the extraction of points for the subsequent matching. It is very accurate but computationally the most expensive operator out of the interest operators presented in this section.

The results of the operator points_harris are very reproducible, too. Admittedly, the points extracted by the operator points_harris are sometimes not meaningful to a human, e.g., they often lie slightly beside a corner or an eye-catching image structure. Nevertheless, it is faster than the operator points_foerstner. The operator points_harris_binomial detects points of interest using the binomial approximation of the points_harris operator. It is therefore faster than points_harris.

The operator points_lepetit extracts points of interest like corners or blob-like structures from the image. This operator can especially be used for very fast interest point extraction. It is the fastest out of the six operators presented in this section.

The operator points_sojka is specialized in the extraction of corner points.

The operator saddle_points_sub_pix is designed especially for the extraction of saddle points, i.e., points whose image intensity is minimal along one direction and maximal along a different direction.

The number of interest points influence the execution time and the result of the subsequent matching process. The more interest points are used, the longer the matching takes. If too few points are used the probability of an erroneous result increases.

In most cases, the default parameters of the interest operators need not be changed. Only if too many or too few interest points are found adaptations of the parameters might be necessary. For a description of the parameters, please refer to the respective pages of the Reference Manual (points_foerstner, points_harris, points_harris_binomial, points_lepetit, points_sojka, saddle_points_sub_pix).

10.4 Matching of Characteristic Points in Overlapping Areas and Determination of the Transformation between the Images

The most demanding task during the generation of an image mosaic is the matching process. The operator proj_match_points_ransac is able to perform the matching even if the two images are shifted and rotated arbitrarily.

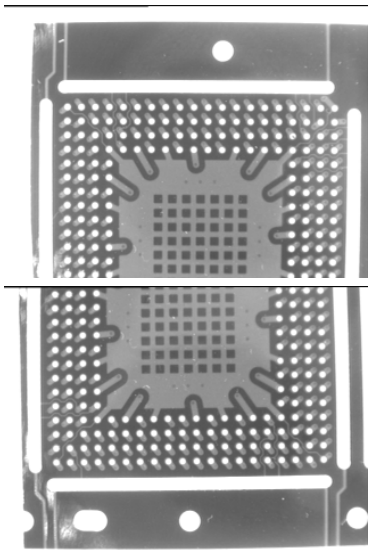
```
proj_match_points_ransac (ImageFrom, ImageTo, RowFromAll, ColumnFromAll, \
                          RowToAll, ColumnToAll, 'sad', MaskSize, RowMove, \
                          ColumnMove, RowTolerance, ColumnTolerance, \
                          Rotation, MatchThreshold, 'gold_standard', \
                          DistanceThreshold, RandSeed, ProjMatrix, Points1, \
                          Points2)
```

The only requirement is that the images should have approximately the same scale. If information about shift and rotation is available it can be used to restrict the search space, which speeds up the matching process and makes it more robust.

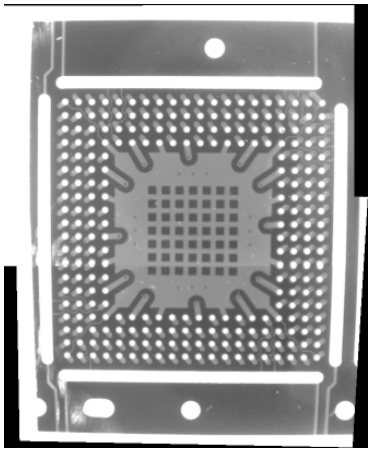
In case the matching fails, ensure that there are enough characteristic points and that the search space and the maximum rotation are defined appropriately.

If the images that should be mosaicked contain repetitive patterns, like the two images of a BGA shown in figure 10.12a, it may happen that the matching does not work correctly. In the resulting erroneous mosaic image, the separate images may not fit together or may be heavily distorted. To achieve a correct matching result for such images, it is important to provide initial values for the shift between the images with the parameters RowMove and ColMove. In addition, the search space should be restricted to an area that contains only one instance of the repetitive pattern, i.e., the values of the parameters RowTolerance and ColTolerance should be chosen smaller than the distance between the instances of the repetitive pattern. With this, it is possible to obtain proper mosaic images, even for objects like BGAs (see figure 10.12b).

For a detailed description of the other parameters, please refer to the Reference Manual (proj_match_points_ransac).



(a)



(b)

Figure 10.12: Separate images (a) and mosaic image (b) of a BGA.

The results of the operator proj_match_points_ransac are the projective transformation matrix and the two tuples Points1 and Points2 that contain the indices of the matched input points from the two images.

The projective transformation matrices resulting from the matching between the image pairs must be accumulated.

```
ProjMatrices := [ProjMatrices,ProjMatrix]
```

Alternatively, if it is known that the mapping between the images is a rigid 2D transformation, the operator proj_match_points_ransac can be used to determine the point correspondences only, since it returns the indices of the corresponding points in the tuples Points1 and Points2. With this, the corresponding point coordinates can be selected.

```
RowFrom := subset(RowFromAll,Points1)
ColumnFrom := subset(ColumnFromAll,Points1)
RowTo := subset(RowToAll,Points2)
ColumnTo := subset(ColumnToAll,Points2)
```

Then, the rigid transformation between the image pair can be determined with the operator vector_to_rigid. Note that we have to add *0.5* to the coordinates to make the extracted pixel positions fit the coordinate system that is used by the operator gen_projective_mosaic.

```
vector_to_rigid (RowFrom + 0.5, ColumnFrom + 0.5, RowTo + 0.5, \
                 ColumnTo + 0.5, HomMat2D)
```

Because gen_projective_mosaic expects a 3×3 transformation matrix, but vector_to_rigid returns a 2×3 matrix, we have to add the last row *[0,0,1]* to the transformation matrix before we can accumulate it.

```
ProjMatrix := [HomMat2D,0, 0, 1]
ProjMatrices := [ProjMatrices,ProjMatrix]
```

Furthermore, the operator proj_match_points_ransac_guided is available. Like proj_match_points_ransac, it can be used to calculate the projective transformation matrix between two images by finding correspondences between points. But in contrast to proj_match_points_ransac, it is based on a known approximation of the projective transformation matrix. Thus, it can be used, for example, to speed up the matching of very large images by implementing an image-pyramid-based projective matching algorithm. The HDevelop example program %HALCONEXAMPLES%\hdevelop\Tools\Mosaicking\mosaicking_pyramid.hdev shows how to implement the image-pyramid-based approach and compares the runtime for different numbers of pyramid levels.

10.5 Generation of the Mosaic Image

Once the transformations between the image pairs are known the mosaic image can be generated with the operator gen_projective_mosaic.

```
gen_projective_mosaic (Images, MosaicImage, StartImage, From, To, \
                       ProjMatrices, StackingOrder, 'false', \
                       MosaicMatrices2D)
```

It requires the images to be given in a tuple. All images are projected into the image plane of a so-called start image. The start image can be defined by its position in the image tuple (starting with *1*) with the parameter StartImage.

Additionally, the image pairs must be specified together with the corresponding transformation matrices.

The order in which the images are added to the mosaic image can be specified with the parameter StackingOrder. The first index in this array will end up at the bottom of the image stack while the last one will be on top. If *'default'* is given instead of an array of integers, the canonical order (the order in which the images are given) will be used.

If the domains of the images should be transformed as well, the parameter TransformRegion must be set to *'true'*.

The output parameter MosaicMatrices2D contains the projective 3×3 transformation matrices for the mapping of the separate images into the mosaic image. These matrices can, e.g., be used to transform features extracted from the separate images into the mosaic image by using the operators projective_trans_pixel, projective_trans_region, projective_trans_contour_xld, or projective_trans_image.

10.6 Bundle Adjusted Mosaicking

It is also possible to generate the mosaic based on the matching results of all overlapping image pairs. The transformation matrices between the images are then determined together within one bundle adjustment. For this, the operators bundle_adjust_mosaic and gen_bundle_adjusted_mosaic are used.

The main advantage of the bundle adjusted mosaicking compared with the mosaicking based on single image pairs is that the bundle adjustment determines the geometry of the mosaic as robustly as possible. Typically, this leads to more accurate results. Another advantage is that there is no need to figure out a good pair configuration, you simply pass the matching results of all overlapping image pairs. What is more, it is possible to define the class of transformations that is used for the transformation between the individual images. A disadvantage of the bundle adjusted mosaicking is that it takes more time to perform the matching between all overlapping image pairs instead of just using a subset. Furthermore, if the matching between two images was erroneous, sometimes the respective image pair is difficult to find in the set of all image pairs.

With this, it is obvious that the bundle adjustment is worthwhile if there are multiple overlaps between the images, i.e., if there are more than $n-1$ overlapping image pairs, with n being the number of images. Another reason for using the bundle adjusted mosaicking is the possibility to define the class of transformations.

The example program %HALCONEXAMPLES%\solution_guide\3d_vision\bundle_adjusted_mosaicking.hdev shows how to generate the bundle adjusted mosaic from the ten images of the BGA displayed in figure 10.7 on page 210. The design of the program is very similar to that of the example program %HALCONEXAMPLES%\solution_guide\3d_vision\mosaicking.hdev, which is described in the introduction of chapter 10 on page 205. The main differences are that

- the matching is carried out between all overlapping images,
- in addition to the projective transformation matrices also the coordinates of the corresponding points must be accumulated, and
- the operator gen_projective_mosaic is replaced with the operators bundle_adjust_mosaic and gen_bundle_adjusted_mosaic.

First, the matching is carried out between all overlapping image pairs, which can be defined as follows:

```
From := [1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5]
To := [6, 7, 2, 6, 7, 8, 3, 7, 8, 9, 4, 8, 9, 10, 5, 9, 10]
```

In addition to the accumulation of the projective transformation matrices, as described in section 10.4 on page 213, also the coordinates of the corresponding points as well as the number of corresponding points must be accumulated.

```
Rows1 := [Rows1,subset(RowFAll,Points1)]
Cols1 := [Cols1,subset(ColFAll,Points1)]
Rows2 := [Rows2,subset(RowTAll,Points2)]
Cols2 := [Cols2,subset(ColTAll,Points2)]
NumCorrespondences := [NumCorrespondences,|Points1|]
```

This data is needed by the operator bundle_adjust_mosaic, which determines the bundle adjusted transformation matrices.

```
bundle_adjust_mosaic (10, StartImage, From, To, ProjMatrices, Rows1, Cols1, \
                      Rows2, Cols2, NumCorrespondences, Transformation, \
                      MosaicMatrices2D, Rows, Cols, Error)
```

The parameter Transformation defines the class of transformations that is used for the transformation between the individual images. Possible values are *'projective'*, *'affine'*, *'similarity'*, and *'rigid'*. Thus, if you know, e.g., that the camera looks perpendicular onto a planar object and that the camera movement between the images is restricted to rotations and translations in the object plane, you can choose the transformation class *'rigid'*. If translations may also occur in the direction perpendicular to the object plane, you must use *'similarity'* because this transformation class allows scale differences between the images. If the camera looks tilted onto the object, the transformation class *'projective'* must be used, which can be approximated by the transformation class *'affine'*. Figure 10.13 shows cut-outs of the resulting mosaic images. They depict the seam between image 5 and image 10. The mosaic images have been created using the images where the lens distortions have been eliminated. The position of the cut-out within the whole mosaic image is indicated by the rectangle in figure 10.7 on page 210.

Finally, with the transformation matrices MosaicMatrices2D, which are determined by the operator bundle_adjust_mosaic, the mosaic can be generated with the operator gen_bundle_adjusted_mosaic.

```
gen_bundle_adjusted_mosaic (Images, MosaicImage, MosaicMatrices2D, \
                            StackingOrder, TransformRegion, TransMat2D)
```

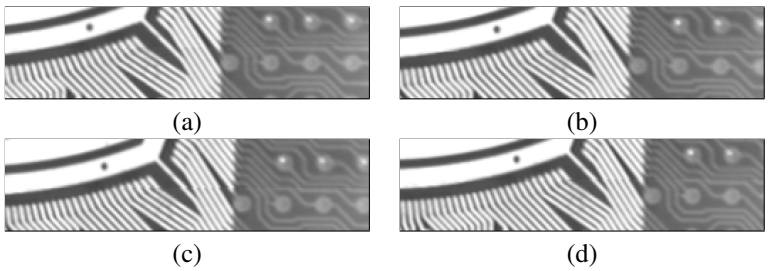


Figure 10.13: Seam between image 5 and image 10 for different classes of transformations: (a) projective, (b) affine, (c) similarity, and (d) rigid.

10.7 Spherical Mosaicking

The methods described in the previous sections arranged the images on a plane. As the name suggests, using spherical mosaicking you can arrange them on a sphere instead. Note that this method can only be used if the camera is only rotated around its optical center or zoomed. If the camera movement includes a translation or if the rotation is not performed exactly around the optical center, the resulting mosaic image will not be accurate and can therefore not be used for high-accuracy applications.

To create a spherical mosaic, you first perform the matching as described in the previous sections to determine the projective transformation matrices between the individual images. This information is the input for the operator stationary_camera_self_calibration, which determines the internal camera parameters of the camera and the rotation matrices for each image. Based on this information, the operator gen_spherical_mosaic

then creates the mosaic image. Please have a look at the HDevelop example program `%HALCONEXAMPLES%\hdevelop\Calibration\Self-Calibration\stationary_camera_self_calibration.hdev` for more information about how to use these operators.

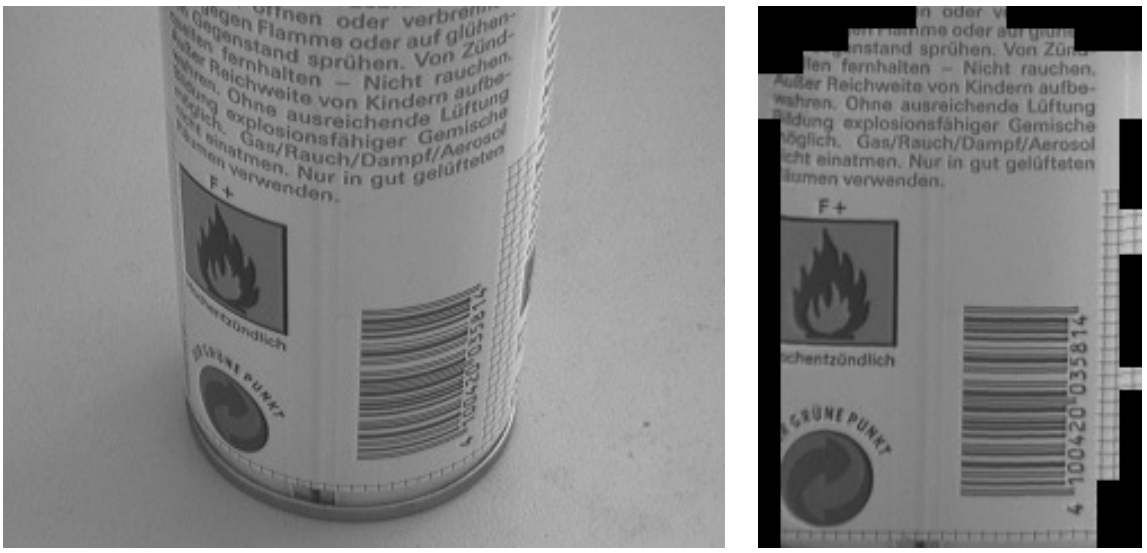
As an alternative, you can map the images on the six sides of a cube using `gen_cube_map_mosaic`. Cube maps are especially useful in computer graphics.

Chapter 11

Rectification of Arbitrary Distortions

For many applications like OCR or bar code reading, distorted images must be rectified prior to the extraction of information. The distortions may be caused by the perspective projection and by the radial lens distortions as well as by the decentering lens distortions, a non-flat object surface, or by any other reason. In the first three cases, i.e., if the object surface is flat and the camera shows only radial or decentering distortions, the rectification can be carried out very precisely as described in section 3.4.1 on page 80. For the remaining cases, a piecewise bilinear rectification can be carried out. In HALCON, this kind of rectification is called grid rectification.

The following example (`%HALCONEXAMPLES%\hdevelop\Tools\Grid-Rectification\grid_rectification.hdev`) shows how the grid rectification can be used to rectify the image of a cylindrically shaped object (figure 11.1).



a) b)

Figure 11.1: Cylindrical object: a) Original image; b) rectified image.

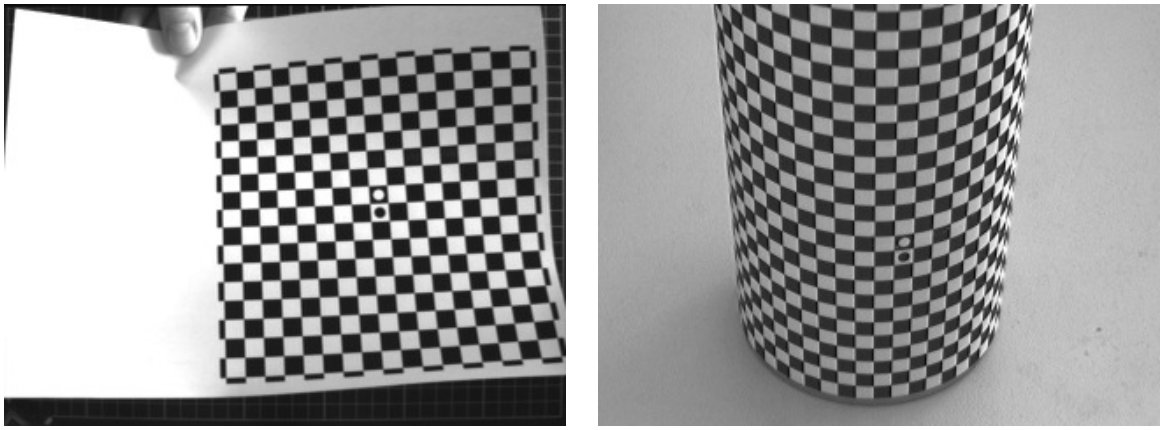
The main idea of the grid rectification is that the mapping for the rectification is determined from an image of the object, where the object is covered by a known pattern.

First, this pattern, which is called rectification grid, must be created with the operator `create_rectification_grid`.

```
create_rectification_grid (WidthOfGrid, NumSquares, 'rectification_grid.ps')
```

The resulting PostScript file must be printed. An example for such a rectification grid is shown in figure 11.2a.

Now, the object must be wrapped with the *rectification grid* and an image of the wrapped object must be taken (figure 11.2b).



a) b)

Figure 11.2: a) Example of a rectification grid. b) Cylindrical object wrapped with the rectification grid.

From this image, the mapping that describes the transformation from the distorted image into the rectified image can be derived. For this, first, the rectification grid must be extracted. Then, the rectification map is derived from the distorted grid. This can be achieved by the following lines of code:

```
find_rectification_grid (Image, GridRegion, MinContrast, Radius)
reduce_domain (Image, GridRegion, ImageReduced)
saddle_points_sub_pix (ImageReduced, 'facet', SigmaSaddlePoints, Threshold, \
                       Row, Col)
connect_grid_points (ImageReduced, ConnectingLines, Row, Col, \
                     SigmaConnectGridPoints, MaxDist)
gen_grid_rectification_map (ImageReduced, ConnectingLines, Map, Meshes, \
                            GridSpacing, 0, Row, Col, 'bilinear')
```

Using the derived map, any image that shows the same distortions can be rectified such that the parts that were covered by the rectification grid appear undistorted in the rectified image (figure 11.1b). This mapping is performed by the operator `map_image`.

```
map_image (ImageReduced, Map, ImageMapped)
```

In the following section, the basic principle of the grid rectification is described. Then, some hints for taking images of the rectification grid are given. In section 11.3 on page 223, the use of the involved HALCON operators is described in more detail based on the above example application. Finally, it is described briefly how to use self-defined grids for the generation of rectification maps.

11.1 Basic Principle

The basic principle of the grid rectification is that a mapping from the distorted image into the rectified image is determined from a distorted image of the rectification grid whose undistorted geometry is well known: The black and white fields of the printed rectification grid are squares (figure 11.3).

In the distorted image, the black and white fields do not appear as squares (figure 11.4a) because of the non-planar object surface, the perspective distortions, and the lens distortions.

To determine the mapping for the rectification of the distorted image, the distorted rectification grid must be extracted. For this, first, the corners of the black and white fields must be extracted with the operator `saddle_points_sub_pix` (figure 11.4b). These corners must be connected along the borders of the black and white fields with the operator `connect_grid_points` (figure 11.4c). Finally, the connecting lines must be combined into meshes (figure 11.4d) with the operator `gen_grid_rectification_map`, which also determines the mapping for the rectification of the distorted image.

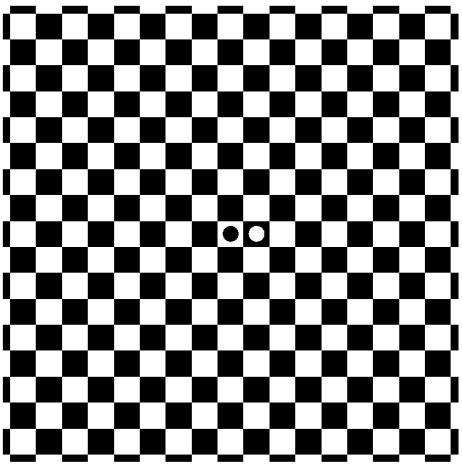


Figure 11.3: Rectification grid.

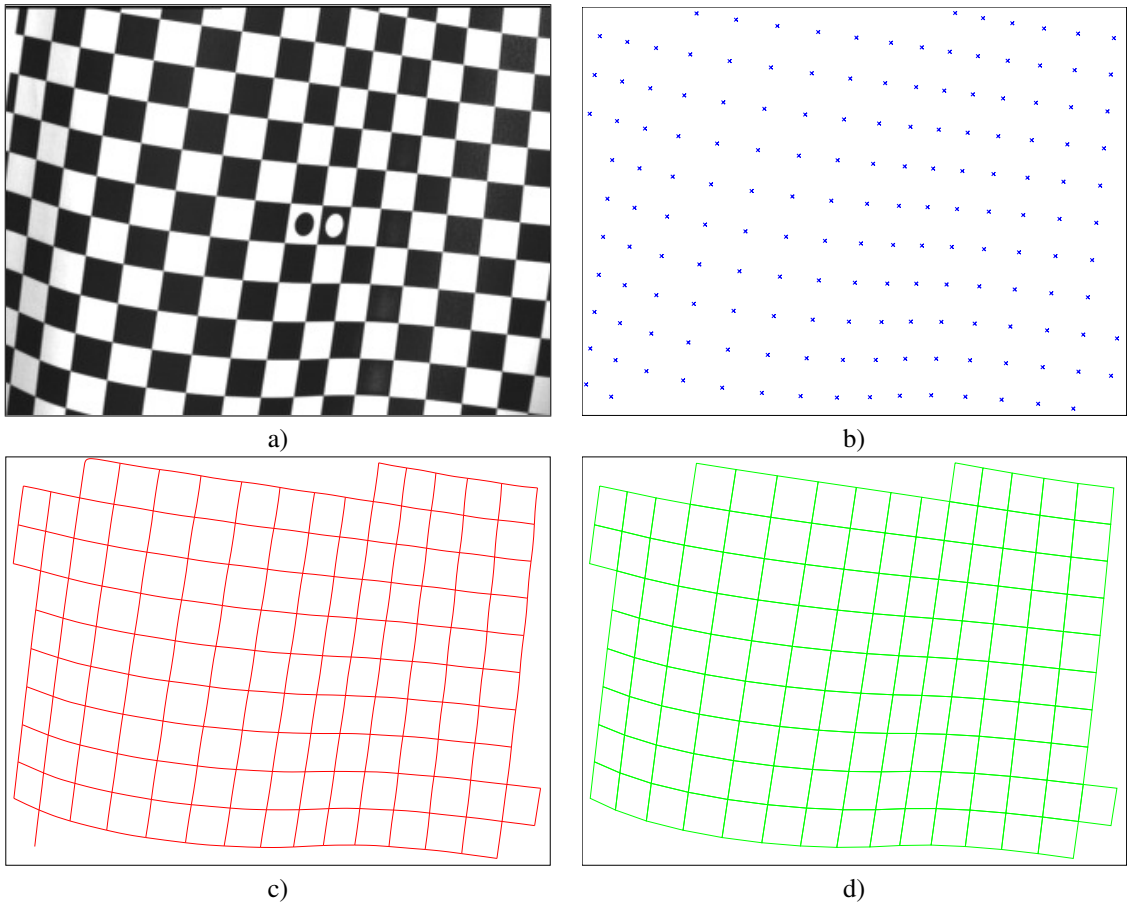


Figure 11.4: Distorted rectification grid: a) Image; b) extracted corners of the black and white fields; c) lines that connect the corners; d) extracted rectification grid.

If you want to use a self-defined grid, the grid points must be defined by yourself. Then, the operator `gen_arbitrary_distortion_map` can be used to determine the mapping (see section 11.4 on page 225 for an example).

The mapping is determined such that the distorted rectification grid will be mapped into its original undistorted geometry (figure 11.5). With this mapping, any image that shows the same distortions can be rectified easily with the operator `map_image`. Note that within the meshes a bilinear interpolation is carried out. Therefore, it is important to use a rectification grid with an appropriate grid size (see section 11.2 for details).

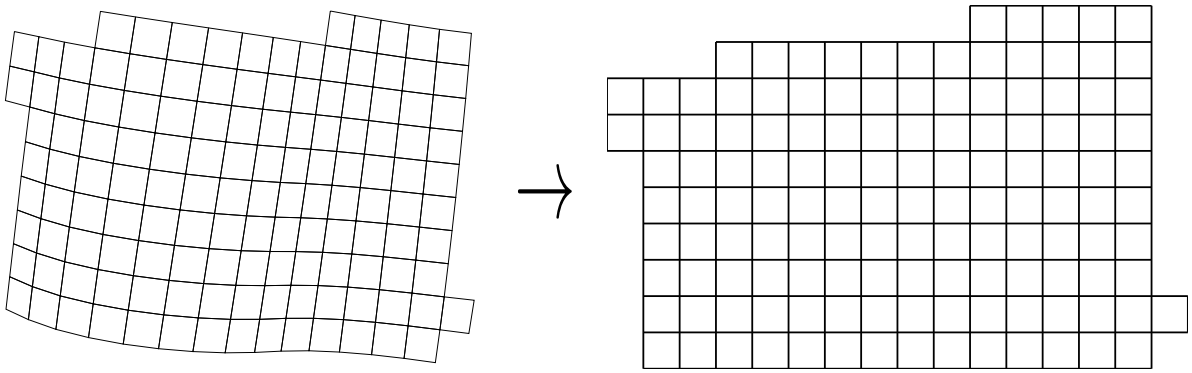


Figure 11.5: Mapping of the distorted rectification grid (a) into the undistorted rectification grid (b).

11.2 Rules for Taking Images of the Rectification Grid

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

- The image must not be overexposed or underexposed: otherwise, the extraction of the corners of the black and white fields of the rectification grid may fail.
- The contrast between the bright and the dark fields of the rectification grid should be as high as possible.
- Ensure that the rectification grid is homogeneously illuminated.
- The images should contain as little noise as possible.
- The border length of the black and white fields should be at least 10 pixels.

In addition to these few rules for the taking of the images of the rectification grid, it is very important to use a rectification grid with an appropriate grid size because the mapping is determined such that within the meshes of the rectification grid a bilinear interpolation is applied. Because of this, non-linear distortions within the meshes cannot be eliminated.

The use of a rectification grid that is too coarse (figure 11.6a), i.e., whose grid size is too large, leads to errors in the rectified image (figure 11.6b).

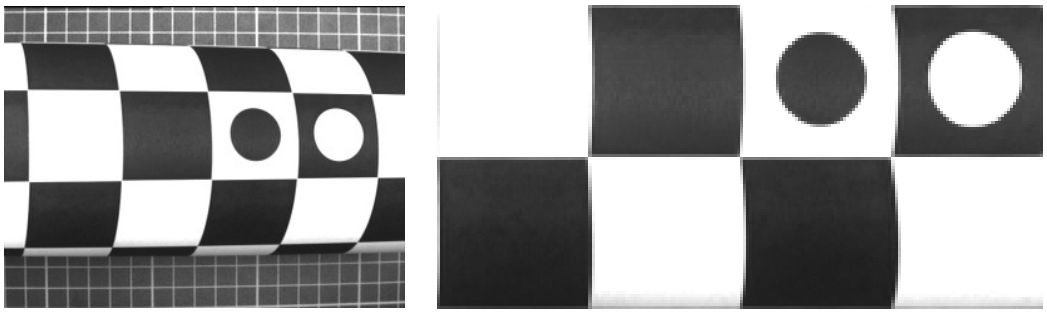
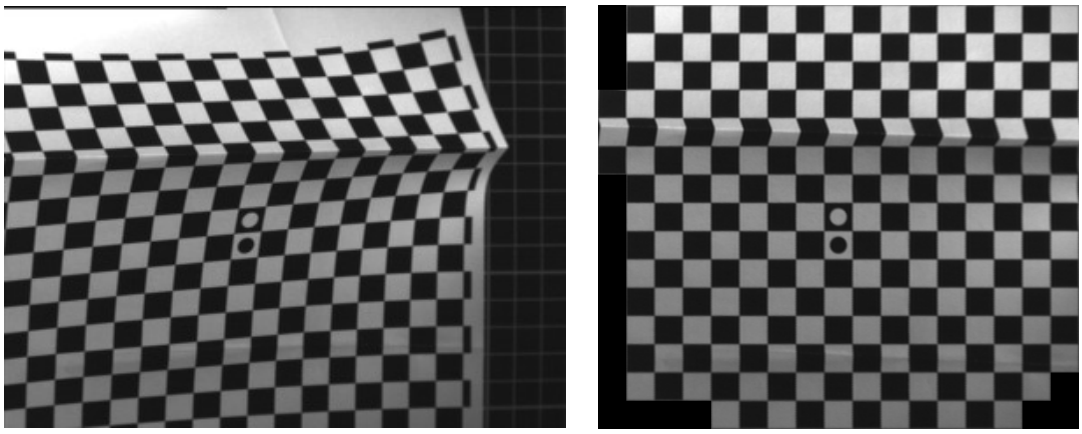


Figure 11.6: Cylindrical object covered with a very coarse rectification grid: a) Distorted image; b) rectified image.

If it is necessary to fold the rectification grid, it should be folded along the borders of the black and white fields. Otherwise, i.e., if the fold crosses these fields (figure 11.7a), the rectified image (figure 11.7b) will contain distortions because of the bilinear interpolation within the meshes.



a) b)

Figure 11.7: Rectification grid folded across the borders of the black and white fields: a) Distorted image; b) rectified image.

11.3 Machine Vision on Ruled Surfaces

In this section, the rectification of images of ruled surfaces is described in detail. Again, the example of the cylindrically shaped object (`%HALCONEXAMPLES%\hdevelop\Tools\Grid-Rectification\grid_rectification.hdev`) is used to explain the involved operators.

First, the operator `create_rectification_grid` is used to create a suitable rectification grid.

```
create_rectification_grid (WidthOfGrid, NumSquares, 'rectification_grid.ps')
```

The parameter `WidthOfGrid` defines the effectively usable size of the rectification grid in meters and the parameter `NumSquares` sets the number of squares (black and white fields) per row. The rectification grid is written to the PostScript file that is specified by the parameter `GridFile`.

To determine the mapping, an image of the rectification grid, wrapped around the object, must be taken as described in section 11.2 on page 222. Figure 11.8a shows an image of a cylindrical object and figure 11.8b shows the same object wrapped by the rectification grid.

Then, the rectification grid is searched in this image with the operator `find_rectification_grid`.

```
find_rectification_grid (Image, GridRegion, MinContrast, Radius)
```

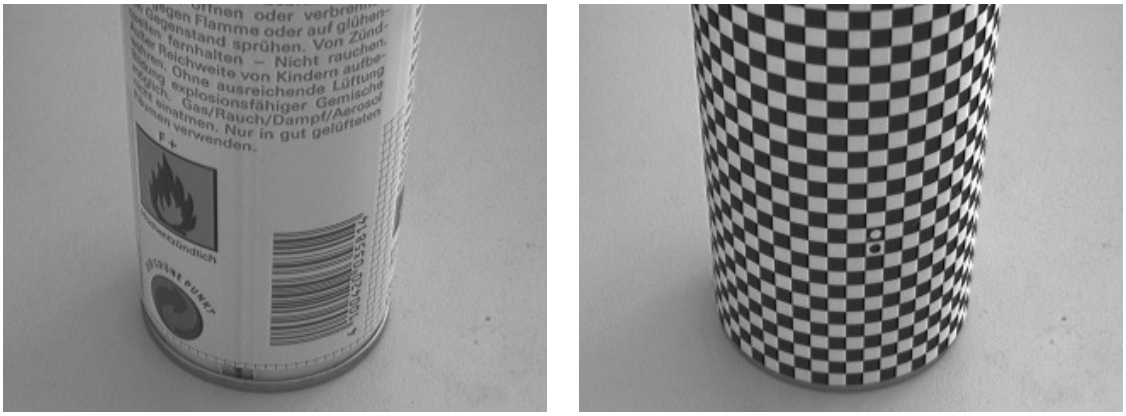
The operator `find_rectification_grid` extracts image areas with a contrast of at least `MinContrast` and fills up the holes in these areas. Note that in this case, contrast is defined as the gray value difference of neighboring pixels in a slightly smoothed copy of the image (Gaussian smoothing with $\sigma = 1.0$). Therefore, the value for the parameter `MinContrast` must be set significantly lower than the gray value difference between the black and white fields of the rectification grid. Small areas of high contrast are then eliminated by an opening with the radius `Radius`. The resulting region is used to restrict the search space for the following steps with the operator `reduce_domain` (see figure 11.9a).

```
reduce_domain (Image, GridRegion, ImageReduced)
```

The corners of the black and white fields appear as saddle points in the image. They can be extracted with the operator `saddle_points_sub_pix` (see figure 11.9b).

```
saddle_points_sub_pix (ImageReduced, 'facet', SigmaSaddlePoints, Threshold, \
                       Row, Col)
```

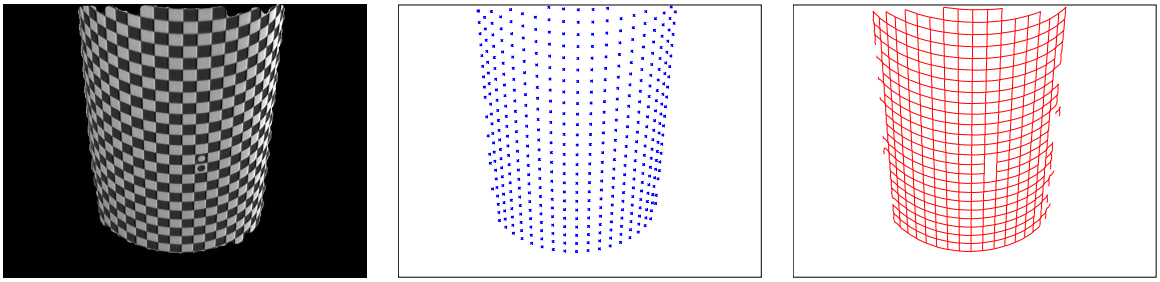
The parameter `Sigma` controls the amount of Gaussian smoothing that is carried out before the actual extraction of the saddle points. Which point is accepted as a saddle point is based on the value of the parameter `Threshold`. If



a) b)

Figure 11.8: Cylindrical object: a) Without and b) with rectification grid.

`Threshold` is set to higher values, fewer but more distinct saddle points are returned than if `Threshold` is set to lower values. The filter method that is used for the extraction of the saddle points can be selected by the parameter `Filter`. It can be set to *'facet'* or *'gauss'*. The method *'facet'* is slightly faster. The method *'gauss'* is slightly more accurate but tends to be more sensitive to noise.



a) b) c)

Figure 11.9: Distorted rectification grid: a) Image reduced to the extracted area of the rectification grid; b) extracted corners of the black and white fields; c) lines that connect the corners.

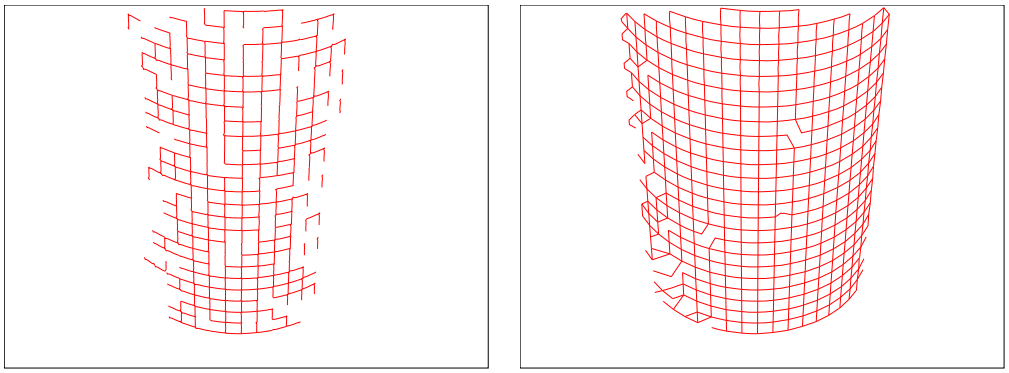
To generate a representation of the distorted rectification grid, the extracted saddle points must be connected along the borders of the black and white fields (figure 11.9c). This is done with the operator `connect_grid_points`.

```
connect_grid_points (ImageReduced, ConnectingLines, Row, Col, \
                     SigmaConnectGridPoints, MaxDist)
```

Again, the parameter `Sigma` controls the amount of Gaussian smoothing that is carried out before the extraction of the borders of the black and white fields. When a tuple of three values [*sigma_min*, *sigma_max*, *sigma_step*] is passed instead of only one value, the operator `connect_grid_points` tests every sigma within the given range from *sigma_min* to *sigma_max* with a step size of *sigma_step* and chooses the sigma that causes the largest number of connecting lines. The same happens when a tuple of only two values *sigma_min* and *sigma_max* is passed. However, in this case a fixed step size of 0.05 is used. The parameter `MaxDist` defines the maximum distance with which an edge may be linked to the respectively closest saddle point. This helps to overcome the problem that edge detectors typically return inaccurate results in the proximity of edge junctions. Figure 11.10 shows the connecting lines if the parameter `MaxDist` has been selected inappropriately: In figure 11.10a, `MaxDist` has been selected to small, whereas in figure 11.10b, it has been selected too large.

Then, the rectification map is determined from the distorted grid with the operator `gen_grid_rectification_map`.

```
gen_grid_rectification_map (ImageReduced, ConnectingLines, Map, Meshes, \
                            GridSpacing, 0, Row, Col, 'bilinear')
```



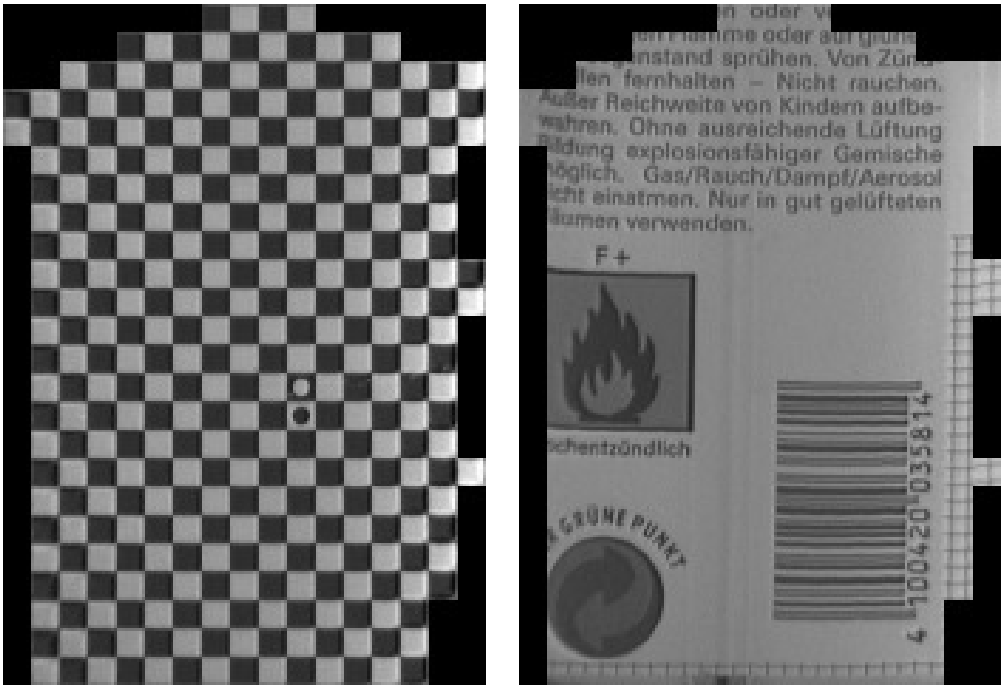
a) b)

Figure 11.10: Connecting lines: Parameter `MaxDist` selected a) too small and b) too large.

The parameter `GridSpacing` defines the size of the grid meshes in the rectified image. Each of the black and white fields is projected onto a square of `GridSpacing` × `GridSpacing` pixels. The parameter `Rotation` controls the orientation of the rectified image. The rectified image can be rotated by *0*, *90*, *180*, or *270* degrees, or it is rotated such that the black circular mark is left of the white circular mark if `Rotation` is set to *'auto'*.

Using the derived rectification map, any image that shows the same distortions can be rectified very fast with the operator `map_image` (see figure 11.11). Note that the objects must appears at exactly the same position in the distorted images.

```
map_image (ImageReduced, Map, ImageMapped)
```



a) b)

Figure 11.11: Rectified images: a) Rectification grid; b) object.

11.4 Using Self-Defined Rectification Grids

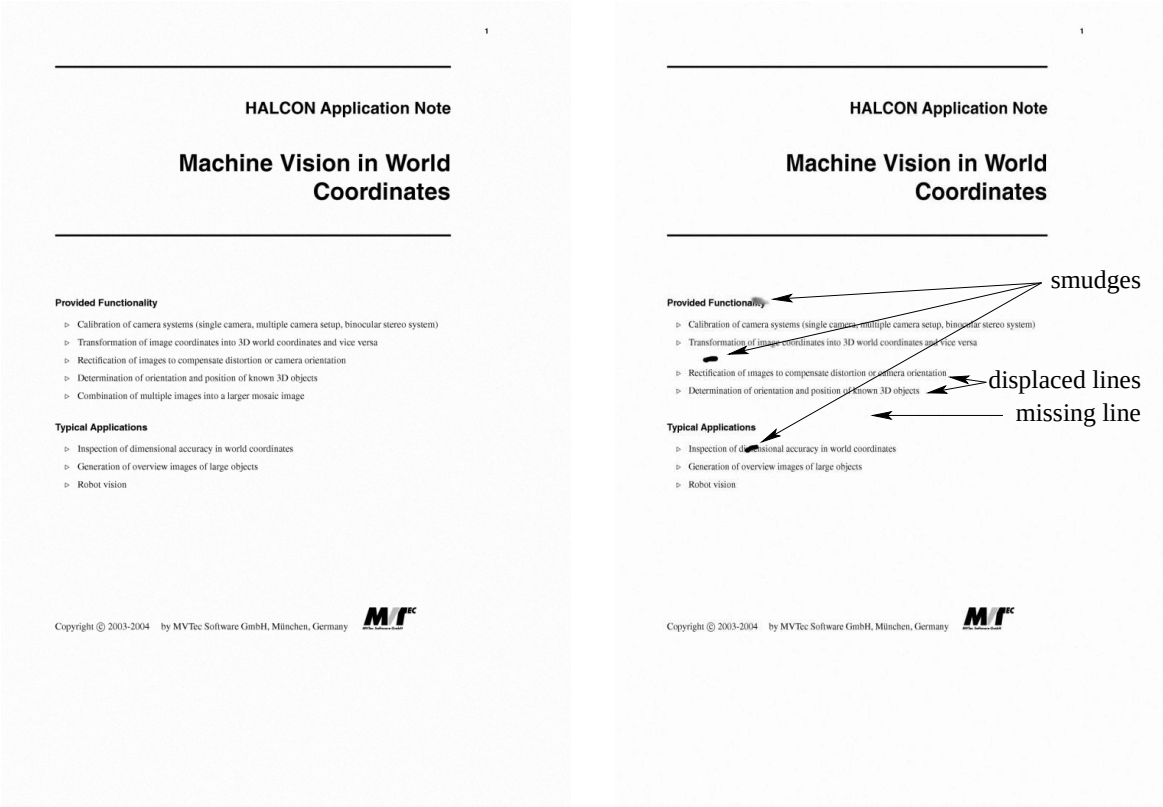
Up to now, we have used the predefined rectification grid together with the appropriate operators for its segmentation. In this section, an alternative to this approach is presented. You can arbitrarily define the rectification grid by

yourself, but note that in this case you must also carry out the segmentation by yourself.

This example shows how the grid rectification can be used to generate arbitrary distortion maps based on self-defined grids.

The example application is a print inspection. It is assumed that some parts are missing and that smudges are present. In addition, lines may be vertically shifted, e.g., due to an inaccurate paper transport, i.e., distortions in the vertical direction of the printed document may be present. These distortions should not result in a rejection of the tested document. Therefore, it is not possible to simply compute the difference image between a reference image and the image that must be checked.

Figure 11.12a shows the reference image and figure 11.12b the test image that must be checked.



a) b)

Figure 11.12: Images of one page of a document: a) Reference image; b) test image that must be checked.

In a first step, the displacements between the lines in the reference document and the test document are determined. With this, the rectification grid is defined. The resulting rectification map is applied to the reference image to transform it into the geometry of the test image. Now, the difference image of the mapped reference image and the test image can be computed.

The HDevelop example program `%HALCONEXAMPLES%\solution_guide\3d_vision\grid_rectification_arbitrary_distortion.hdev` uses the shape-based matching to determine corresponding points in the reference image and the test image. First, the different matching models are generated for every region in the reference image. Then, the corresponding points are searched in the test image by searching the matches for every model.

Based on the corresponding points of the reference and the test image (`RowRef`, `ColRef`, `RowTest`, and `ColTest`), the coordinates of the grid points of the distorted grid are determined. In this example, the row and column coordinates can be determined independently from each other because only the row coordinates are distorted. Note that the upper left grid point of the undistorted grid is assumed to have the coordinates *(-0.5, -0.5)*. This means that the corresponding grid point of the distorted grid will be mapped to the point *(-0.5, -0.5)*. Because there are only vertical distortions in this example, the column coordinates of the distorted grid are equidistant, starting at the value *-0.5*.

```
GridSpacing := 10
ColShift := mean(ColTest - ColRef)
RefGridColValues := []
for HelpCol := -0.5 to WidthTest + GridSpacing by GridSpacing
    RefGridColValues := [RefGridColValues,HelpCol + ColShift]
endfor
```

The row coordinates of the distorted grid are determined by a linear interpolation between the above determined pairs of corresponding row coordinates.

```
MinValue := 0
MaxValue := HeightTest + GridSpacing
sample_corresponding_values (RowTest, RowRef - 0.5, MinValue, MaxValue, \
                             GridSpacing, RefGridRowValues)
```

The interpolation is performed within the procedure which is part of the HDevelop example program %HALCONEXAMPLES%\solution_guide\3d_vision\grid_rectification_arbitrary_distortion.hdev.

```
procedure sample_corresponding_values (Values, CorrespondingValues,
                                       MinValue, MaxValue,
                                       InterpolationInterval,
                                       SampledCorrespondingValues):::
```

Now, the distorted grid is generated row by row.

```
RefGridRow := []
RefGridCol := []
Ones := gen_tuple_const(|RefGridColValues|,1)
for r := 0 to |RefGridRowValues| - 1 by 1
    RefGridRow := [RefGridRow,RefGridRowValues[r] * Ones]
    RefGridCol := [RefGridCol,RefGridColValues]
endfor
```

The operator gen_arbitrary_distortion_map uses this distorted grid to derive the rectification map that maps the reference image into the geometry of the test image[1].

```
gen_arbitrary_distortion_map (Map, GridSpacing, RefGridRow, RefGridCol, \
                              |RefGridColValues|, WidthRef, HeightRef, \
                              'bilinear')
```

With this rectification map, the reference image can be transformed into the geometry of the test image. Note that the size of the mapped image depends on the number of grid cells and on the size of one grid cell, which must be defined by the parameter GridSpacing. Possibly, the size of the mapped reference image must be adapted to the size of the test image.

```
map_image (ImageRef, Map, ImageMapped)
crop_part (ImageMapped, ImagePart, 0, 0, WidthTest, HeightTest)
```

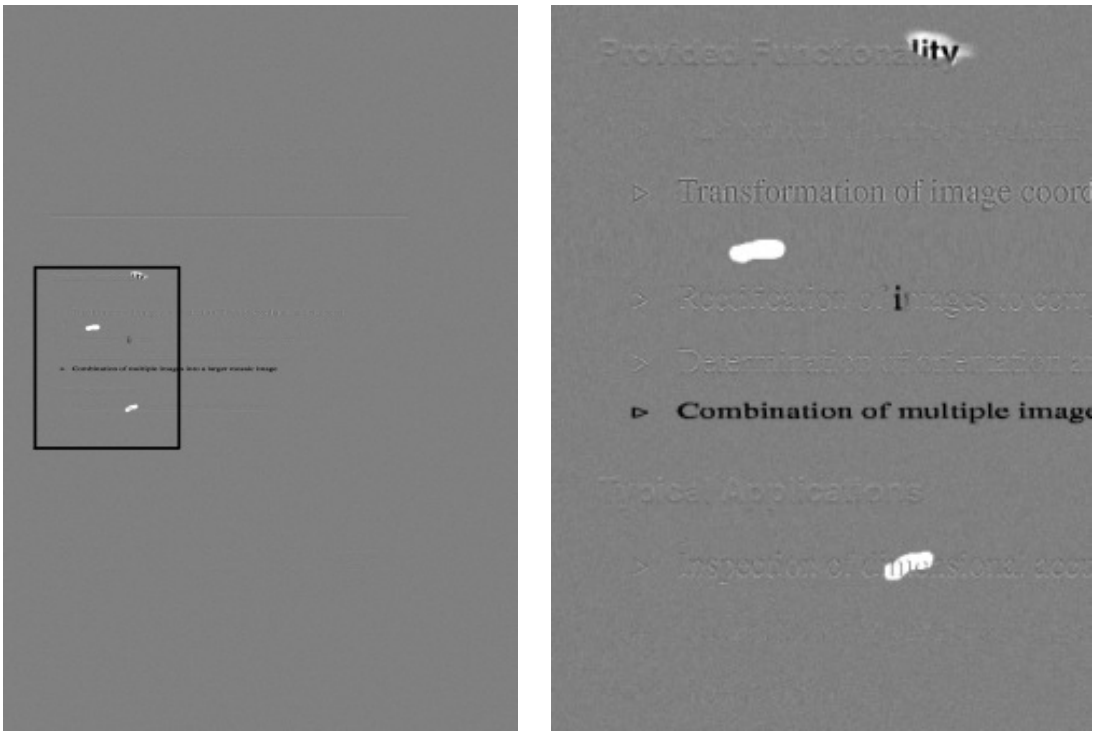
Finally, the test image can be subtracted from the mapped reference image.

```
sub_image (ImagePart, ImageTest, ImageSub, 1, 128)
```

Figure 11.13 shows the resulting difference image. In this case, missing parts appear dark while the smudges appear bright.

The differences between the test image and the reference image can now be extracted easily from the difference image with the operator threshold. If the difference image is not needed, e.g., for visualization purposes, the differences can be derived directly from the test image and the reference image with the operator dyn_threshold.

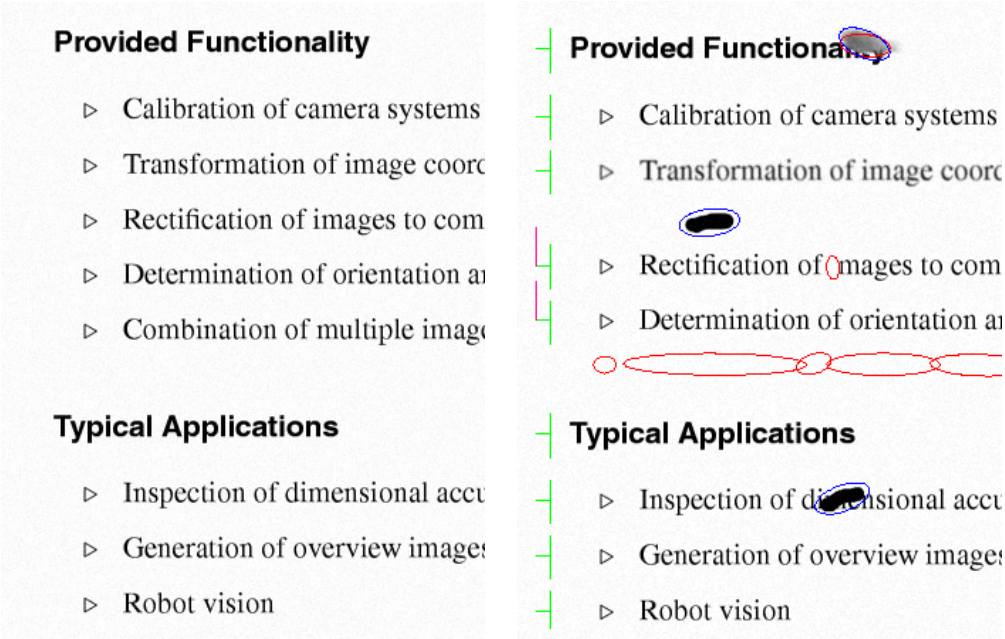
[1] In this case, the reference image is mapped into the geometry of the test image to facilitate the marking of the differences in the test image. Obviously, the rectification grid can also be defined such that the test image is mapped into the geometry of the reference image.



a) b)

Figure 11.13: Difference image: a) The entire image overlaid with a rectangle that indicates the position of the cut-out. b) A cut-out.

Figure 11.14 shows the differences in a cut-out of the reference image (figure 11.14a) and of the test image (figure 11.14b). The calibration marks near the left border of figure 11.14b indicate the vertical position of the components that were used for the determination of the corresponding points. Vertical shifts of the components with respect to the reference image are indicated by a vertical line of the respective length that is attached to the respective calibration mark. All other differences that could be detected between the test image and the reference image are encircled.



a) b)

Figure 11.14: Cut-out of the reference and the checked test image with the differences marked in the test image: a) Reference image; b) checked test image.

Appendix A

HDevelop Procedures Used in this Solution Guide

A.1 gen_hom_mat3d_from_three_points

```
procedure gen_hom_mat3d_from_three_points (Origin, PointOnXAxis,
                                           PointInXYPlane, HomMat3d):::
XAxis := [PointOnXAxis[0] - Origin[0],PointOnXAxis[1] - Origin[1], \
          PointOnXAxis[2] - Origin[2]]
XAxisNorm := XAxis / sqrt(sum(XAxis * XAxis))
VectorInXYPlane := [PointInXYPlane[0] - Origin[0], \
                    PointInXYPlane[1] - Origin[1], \
                    PointInXYPlane[2] - Origin[2]]
cross_product (XAxisNorm, VectorInXYPlane, ZAxis)
ZAxisNorm := ZAxis / sqrt(sum(ZAxis * ZAxis))
cross_product (ZAxisNorm, XAxisNorm, YAxisNorm)
HomMat3d_WCS_to_RectCCS := [XAxisNorm[0],YAxisNorm[0],ZAxisNorm[0], \
                            Origin[0],XAxisNorm[1],YAxisNorm[1], \
                            ZAxisNorm[1],Origin[1],XAxisNorm[2], \
                            YAxisNorm[2],ZAxisNorm[2],Origin[2]]
hom_mat3d_invert (HomMat3d_WCS_to_RectCCS, HomMat3d)
return ()
```

This procedure uses the procedure

```
procedure cross_product (V1, V2, CrossProduct)
CrossProduct := [V1[1] * V2[2] - V1[2] * V2[1], \
                 V1[2] * V2[0] - V1[0] * V2[2], \
                 V1[0] * V2[1] - V1[1] * V2[0]]
return ()
```

A.2 parameters_image_to_world_plane_centered

```
procedure parameters_image_to_world_plane_centered (CamParam, Pose,
                                                    CenterRow, CenterCol,
                                                    WidthMappedImage,
                                                    HeightMappedImage,
                                                    ScaleForCenteredImage,
                                                    PoseForCenteredImage):::
* Determine the scale for the mapping
* (here, the scale is determined such that in the
*   surroundings of the given point  the image scale of the
*   mapped image is similar to the image scale of the original image)
Dist_ICS := 1
image_points_to_world_plane (CamParam, Pose, CenterRow, CenterCol, 1, \
                             CenterX, CenterY)
image_points_to_world_plane (CamParam, Pose, CenterRow + Dist_ICS, \
                             CenterCol, 1, BelowCenterX, BelowCenterY)
image_points_to_world_plane (CamParam, Pose, CenterRow, \
                             CenterCol + Dist_ICS, 1, RightOfCenterX, \
                             RightOfCenterY)
distance_pp (CenterY, CenterX, BelowCenterY, BelowCenterX, \
             Dist_WCS_Vertical)
distance_pp (CenterY, CenterX, RightOfCenterY, RightOfCenterX, \
             Dist_WCS_Horizontal)
ScaleVertical := Dist_WCS_Vertical / Dist_ICS
ScaleHorizontal := Dist_WCS_Horizontal / Dist_ICS
ScaleForCenteredImage := (ScaleVertical + ScaleHorizontal) / 2.0
* Determine the parameters for set_origin_pose such
* that the point given via get_mbutton will be in the center of the
* mapped image
DX := CenterX - ScaleForCenteredImage * WidthMappedImage / 2.0
DY := CenterY - ScaleForCenteredImage * HeightMappedImage / 2.0
DZ := 0
set_origin_pose (Pose, DX, DY, DZ, PoseForCenteredImage)
return ()
```

A.3 parameters_image_to_world_plane_entire

```
procedure parameters_image_to_world_plane_entire (Image, CamParam, Pose,
                                                  WidthMappedImage,
                                                  HeightMappedImage,
                                                  ScaleForEntireImage,
                                                  PoseForEntireImage):::
* Transform the image border into the WCS (scale = 1)
full_domain (Image, ImageFull)
get_domain (ImageFull, Domain)
gen_contour_region_xld (Domain, ImageBorder, 'border')
contour_to_world_plane_xld (ImageBorder, ImageBorderWCS, CamParam, Pose, 1)
smallest_rectangle1_xld (ImageBorderWCS, MinY, MinX, MaxY, MaxX)
* Determine the scale of the mapping
ExtentX := MaxX - MinX
ExtentY := MaxY - MinY
ScaleX := ExtentX / WidthMappedImage
ScaleY := ExtentY / HeightMappedImage
ScaleForEntireImage := max([ScaleX,ScaleY])
* Shift the pose by the minimum X and Y coordinates
set_origin_pose (Pose, MinX, MinY, 0, PoseForEntireImage)
return ()
```

A.4 tilt_correction

```
procedure tilt_correction (DistanceImage, RegionDefiningReferencePlane,
                           DistanceImageCorrected):::
* Reduce the given region, which defines the reference plane
* to the domain of the distance image
get_domain (DistanceImage, Domain)
intersection (RegionDefiningReferencePlane, Domain, \
              RegionDefiningReferencePlane)
* Determine the parameters of the reference plane
moments_gray_plane (RegionDefiningReferencePlane, DistanceImage, MRow, MCol, \
                    Alpha, Beta, Mean)
* Generate a distance image of the reference plane
get_image_pointer1 (DistanceImage, Pointer, Type, Width, Height)
area_center (RegionDefiningReferencePlane, Area, Row, Column)
gen_image_surface_first_order (ReferencePlaneDistance, Type, Alpha, Beta, \
                               Mean, Row, Column, Width, Height)
* Subtract the distance image of the reference plane
* from the distance image of the object
sub_image (DistanceImage, ReferencePlaneDistance, DistanceImageWithoutTilt, \
           1, 0)
* Determine the scale factor for the reduction of the distance values
CosGamma := 1.0 / sqrt(Alpha * Alpha + Beta * Beta + 1)
* Reduce the distance values
scale_image (DistanceImageWithoutTilt, DistanceImageCorrected, CosGamma, 0)
return ()
```

A.5 calc_calplate_pose_movingcam

```
procedure calc_calplate_pose_movingcam (CalibObjInBasePose, ToolInCamPose,
                                        ToolInBasePose, CalibObjInCamPose):::
* CalibObjInCamPose = cam_H_calplate
*                   = cam_H_tool * tool_H_base * base_H_calplate
*                   = ToolInCamPose * BaseInToolPose * CalibrationPose
pose_invert (ToolInBasePose, BaseInToolPose)
pose_compose (ToolInCamPose, BaseInToolPose, BaseInCamPose)
pose_compose (BaseInCamPose, CalibObjInBasePose, CalibObjInCamPose)
return ()
```

A.6 calc_calplate_pose_stationarycam

```
procedure calc_calplate_pose_stationarycam (CalObjInToolPose, BaseInCamPose,
                                            ToolInBasePose,
                                            CalObjInCamPose):::
* CalObjInCamPose = cam_H_calplate = cam_H_base * base_H_tool *  \
* tool_H_calplate
*                                 =  \
* BaseInCamPose*ToolInBasePose*CalObjInToolPose
pose_compose (BaseInCamPose, ToolInBasePose, ToolInCamPose)
pose_compose (ToolInCamPose, CalObjInToolPose, CalObjInCamPose)
return ()
```

A.7 define_reference_coord_system

```
procedure define_reference_coord_system (ImageName, CamParam, CalplateFile,
                                         WindowHandle, PoseCamRef):::
read_image (RefImage, ImageName)
dev_display (RefImage)
caltab_points (CalplateFile, X, Y, Z)
* parameter settings for find_caltab and find_marks_and_pose
SizeGauss := 3
MarkThresh := 100
MinDiamMarks := 5
StartThresh := 128
DeltaThresh := 10
MinThresh := 18
Alpha := 0.9
MinContLength := 15
MaxDiamMarks := 100
find_caltab (RefImage, Caltab, CalplateFile, SizeGauss, MarkThresh, \
             MinDiamMarks)
find_marks_and_pose (RefImage, Caltab, CalplateFile, CamParam, StartThresh, \
                     DeltaThresh, MinThresh, Alpha, MinContLength, \
                     MaxDiamMarks, RCoord, CCoord, PoseCamRef)
disp_3d_coord_system (WindowHandle, CamParam, PoseCamRef, 0.01)
return ()
```

Index