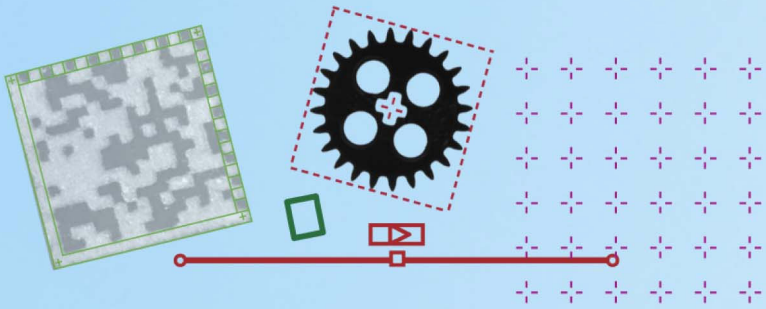
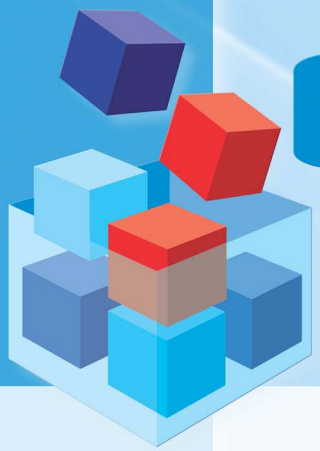


Functional Guide

Open eVision™ 1.2



Open eVision™

Image Analysis Tools

DownloadDevelopDeploy

on any platform

EURESYS™
Excellence in vision

www.euresys.com

Copyright © EURESYS s.a. 1989-2011

Disclaimer

EURESYS s.a. shall retain all property rights, title and interest of the documentation of the hardware and the software, and of the trademarks of EURESYS s.a.

All the names of companies and products mentioned in the documentation may be the trademarks of their respective owners.

The licensing, use, leasing, loaning, translation, reproduction, copying or modification of the hardware or the software, brands or documentation of EURESYS s.a. contained in this book, is not allowed without prior notice.

EURESYS s.a. may modify the product specification or change the information given in this documentation at any time, at its discretion, and without prior notice.

EURESYS s.a. shall not be liable for any loss of or damage to revenues, profits, goodwill, data, information systems or other special, incidental, indirect, consequential or punitive damages of any kind arising in connection with the use of the hardware or the software of EURESYS s.a. or resulting of omissions or errors in this documentation.

Open eVision Functional Guide

Open eVision Libraries

The **Open eVision** libraries are a set of powerful image processing tools, tailored for use in computer vision applications. They cover most state-of-the-art techniques in digital image processing, from classical algorithms to advanced solutions ready-made for specific tasks.

Even though many of the available tools are designed to be self-consistent and easy to use, the advanced user should find everything he needs to build his own workflow by combining the numerous building blocks provided.

Open eVision is made of a set of C++, ActiveX and .NET classes designed to be integrated into your application. The libraries are in no way a closed solution and do require to be integrated in your own application, leaving you all the freedom to deal with all other aspects of the automation not related to image processing.

General-Purpose Libraries

EasyImage includes gray-level image processing functions for linear or non-linear filtering, geometric transformations for registration, histogram analysis for thresholding, edges and corners detection based on Canny and Harris algorithms... These operations are usually performed as pre-processing steps to improve the image quality and obtain a good contrast between the background and the objects of interest.

EasyColor includes color level image processing functions and support for alternate color systems (such as HSV, YUV, L*a*b*...). It provides efficient means to convert images between these systems and transform between color and gray-level images. This library is complementary to EasyImage.

EasyObject is used for blob analysis, that is obtaining information about the distinct objects that are present in an image. The processing is based on connected component labeling (after a segmentation step by thresholding), to isolate and identify the objects. Then, various geometric features of the objects, such as the area or ellipse of inertia parameters, can be computed and the objects can be sorted and selected with respect to these features.

EasyMatch locates patterns or image templates (previously shown to the system) arbitrarily positioned in an image. It can be used for instance for image registration or component placement inspection.

EasyFind is a library aimed at rapidly locating patterns in an image down to the sub-pixel precision. In a one-sentence definition: EasyFind finds instances similar to a model in a search field and reports information about these instances. It is robust against noise, occlusion, blurring and illumination variations.

EasyGauge is a sub-pixel measurement tool that enables accurate dimensional assessment of objects. It relies on a robust algorithm for edge detection and can be used to locate points or fit geometric models. When measurement are performed in a calibrated field of view, EasyGauge provides results in physical units (mm, inch, ...) rather than as a number of pixels.

Mark-Inspection Libraries

EasyOCR is a printed character reader for use in applications such as serial number reading or printed label verification.

EasyOCV is able to check the printing quality of labels against a good quality template. Defects such as low contrast, misalignment, scratches or incorrect marking can be reliably detected. Two complementary tools are available: one to model the marking as a set of independent shapes; another, based on image comparison, to inspect globally.

EasyBarCode is a library dedicated to the reading of bar codes. By contrast with printed characters, bar codes are machine-readable only. They are generally used to ensure traceability of goods.

EasyMatrixCode is a library dedicated to the reading of Data Matrix codes.

Installing Open eVision

1. Supported Platforms and Requirements

For the latest supported operating systems, processor architectures, integrated development environments and required system resources, please refer to the releases notes coming with your Open eVision packages on the Euresys website.

2. Installation under WES 2009

Instead of creating a component, it is possible with Windows Embedded Standard 2009 to install drivers and applications after FBA completes.

It is recommend to install Open eVision in the following method:

- **Add mandatory components to the Run-Time Image using Target Designer**
  - .Net framework 3.0 setup component is needed to run Open eVision license manager
  - Windows Installer Service component. It is required for Open eVision to install its C/C++ run-time libraries.
  - To activate the Open eVision licenses on the final system, the Open eVision license manager application needs either the internet (http/https) support component and the network support component for an online activation or a USB flash disk support for an offline activation
  - To reseal the "**Master Target**" before deploying the image to multiple devices, you have to add either **Sysprep** (Windows System preparation) **Component** or **System Cloning Tool Component**.

Note: To prevent **System Cloning Tool** from executing **FBRESEAL** automatically when FBA finishes. In the System Cloning Tool Settings, set **Reseal Phase** to **Manual**, or go into the **Advanced Settings** of the **System Cloning Tool** settings and change **cmiResealPhase** from **12000** to **0**.

- **Open eVision 1.1 and further versions installation on the "Master Target" device**
  - Once the Pre FBA OS Image has been successfully build, boot the "Master Target" device and allow FBA to complete.
  - When the "**Master Target**" device has been booted for the second time, install the Open eVision libraries using the standard installer provided.
  - Optionally, install your own final application based on Open eVision. Note that you have to include all the run-time libraries needed by your application.
- **Resealing of the master package**
  - When you are ready to mass deploy the image, run **FBRESEAL** or **Sysprep**. Once the computer shutdowns, the image is the master.
  - Each time you deploy the "**Master**" in a new device, the Open eVision libraries needs to be activated as described in the license manager documentation.

3. Installation Options

3.1 Detection of Other Versions and Uninstallation

Open eVision Detection and Uninstall

As far as the removal of Open eVision is concerned, the Open eVision Libraries and Open eVision Development Tools installers of Open eVision 1.1 and further versions remove the same version of Open eVision Libraries and Development Tools when revision differs.

The existing installations of Open eVision Libraries and Open eVision Development Tools having a different version number are left unchanged.

The Open eVision Eval installer removes Open eVision Eval installations of the same Open eVision version. Open eVision Eval installations are never removed by other installers.

The existing installations of Open eVision Eval having a different version number are left unchanged.

eVision Detection and Uninstall

As far as eVision is concerned, the Open eVision Libraries and Open eVision Development Tools installers of Open eVision 1.1 and further versions simply ignore existing installations of eVision.

The existing installations of eVision are left unchanged.

The dongle drivers are also left unchanged.

3.2 Packages

Two installation packages are available for Open eVision:

- **Open eVision Libraries**: contains all libraries and Open eVision License Manager.
- **Open eVision Development Tools**: which contains Open eVision Studio, Open eVision Documentation, sample images and Open eVision License Manager.

Open eVision 1.1 and further versions can be installed alongside previous versions of Open eVision or eVision; it is now possible to use several versions of the same Open eVision libraries in an application.

Note. To install Open eVision, make sure you have administrator rights on your platform.

Deployment

To deploy Open eVision, refer to **Open eVision License Manager Documentation**.

3.3 Customized Setup

When installing **Open eVision Libraries**, choose the relevant components to install. (Open eVision License Manager is always installed.)

Setup type	Description
Full	Installs everything, including optional components. Detected IDEs are automatically configured (*).
Typical	Installs the mostly used Open eVision components. Includes the C++ headers and libraries, the .NET assemblies and the ActiveX controls. By default, detected IDEs are automatically configured.
Custom	Allows customizing the selected components.

When installing **Open eVision Development Tools**, choose the relevant components to install. (Open eVision License Manager is always installed.)

Setup type	Description
Full	Installs everything: Open eVision Studio, Open eVision Documentation, sample images.
Custom	Allows customizing the selected components.

(*) Up to Visual Studio 2008

4. Command-Line Installation

You may want to integrate the Open eVision tools installation into your own application distribution.

Open eVision setup program can be called in command-line mode with your installation options. In this mode, Open eVision setup program is run in silent mode, and no dialog box is displayed.

You need to specify the "/S" flag (note that "/S" is case-sensitive) when installing Open eVision products in silent mode. If Open eVision products are already installed, running Open eVision setup program will remove the version installed on your machine, and install the version contained inside the installer, even if the installed version on your machine is newer than the one contained inside the installer.

Changing the Default Location

By default, the installers will place the files in the folder **C:\Program Files\Euresys\Open eVision X.Y**.(*)

To override this default location, enter the flag "/D=", followed by the desired installation folder. This flag is case-sensitive. It must be the last parameter used in the command line and must not contain any quotes, even if the path contains spaces. Moreover, only absolute paths are supported.

(*) Locations may differ according to your installation, and whether you are using a 64-bit system

Adding More Options

By default, only the options that cannot be deselected (read-only) in the graphical mode will be installed. For Open eVision 1.0.1 and Open eVision 1.1, this corresponds to the Open eVision License Manager, always installed by the Open eVision Libraries and Open eVision Development Tools installation files.

To add more options, you need to specify them on the command line (before the "/D=" flag, if it is used). Such options are case-insensitive, unlike the "/S" and "/D=" flags. Available options are described below. For example:

Open_eVision_64bit_Libraries_1_1_6_6403.exe /S /DotNet /D=E:\Euresys Open eVision

Available Options for Open eVision Libraries

All these options are case-insensitive.

- /Full: Install everything, including optional components. Detected IDEs are automatically configured.
- /Typical: Install the mostly used Open eVision components. Includes the C++ headers and libraries for the detected IDEs, the .NET assemblies and the ActiveX controls. By default, detected IDEs are automatically configured.
- /DotNet: Use Open eVision with .NET enabled development environments.
- /CppHeaders: Install C++ headers.
- /CppDlls: Install the Open eVision C++ dynamic-link libraries for use with C++ IDE.
- /MsVc90Config: Automatically configure Visual C++ 2008 to work with Open eVision.
- /MsVc80Config: Automatically configure Visual C++ 2005 to work with Open eVision.
- /MsVc71Config: Automatically configure Visual C++ .NET 2003 to work with Open eVision.
- /MsVc60Config: Automatically configure Visual C++ 6.0 to work with Open eVision.
- /ActiveXControls: Install the Open eVision ActiveX controls for use with Microsoft Visual Basic.

Note: From Visual Studio 2010 on, the settings are set per-project and are not configured by the Open eVision installer

Available Options for Open eVision Development Tools

All these options are case-insensitive.

- /Full: Install everything.
- /Studio: Install the Open eVision development environment.
- /Documentation: Install the Open eVision documentation.
- /Images: Install images to use with Open eVision Studio and the sample programs.

Error Reporting

After a command-line installation, the following registry key is updated and holds the installation status: **[HKEY_LOCAL_MACHINE\SOFTWARE\Euresys\Common\LastInstallError]**.

- The **ErrorCode** DWORD identifies the error:
  - **0: No error**. Installation successful.
  - 1: Reserved.
  - 2: Reserved.
  - 3: Reserved.
  - **4. Newer Version Found**. A newer version is already installed. Use the force flag to update it anyway.
  - 5: Reserved.
  - **6: Unsupported Version Found**. Unsupported eVision version detected. Please uninstall it before installing this product.
  - **7: Installation failure**. Indicates that a fatal problem was encountered that prevents the installation from pursuing (e.g. impossibility to register the uninstaller).
  - **8: Dongle Driver Removal**. Dongle drivers installed on the machine. Use the force flag to remove them.
  - **9: Failure during the installation of the .NET Framework 2.0 SP1**.
  - **10: Problem in the installation of the Euresys License Manager**.
  - 11: Reserved.
  - **12: Preliminary reboot required**. This value is set if the exit code of the installer was 3.
  - **13: Another instance of a Euresys installer is already running**.
  - **14: Removal not possible**. The currently installed version cannot be removed automatically. Please cancel the current installation and remove the installed version manually.
  - **15: Open eVision does not support virtual machines. Please install on a physical machine**.
- The **Cause** STRING gives a wording of the error.
- The **Source** STRING identifies the installer that caused the error.
- The **ErrorTime** STRING gives the time and date when the error occurred.

Installed Version Management

If the installer encounters a product version already installed on the platform having the first two numeric fields of the version number matching exactly, it does the following with that installation::

- The product version is the same one: the installer first uninstalls the product, then reinstalls it.

- The product version on the platform is older than the one contained inside the installer: the old version is removed, and the new one is installed.
- The product version on the platform is newer than the one contained inside the installer: the installation is aborted. The LastInstallError registry keys are set accordingly (see Error Reporting above). The Cause string mentions the installed version in the message.

The other installations are left unchanged.

However, when the "/Force" flag (case-insensitive) is specified, the found version is *always* removed, and the product contained inside the installer is installed.

Uninstallation

It is possible to silently remove an installed Open eVision product. To this end, the user has to run again the corresponding installer, with the "/S" flag together with the "/Remove" flag (note that the "/S" flag is case-sensitive).

Similarly to the silent installation, a silent uninstallation returns an exit code. The uninstallation process will never leave the computer in its initial state (thus, the "Installation failure" exit code of the installers has no equivalent for uninstallers). In the case of an error, the uninstaller will continue gracefully until its completion.

The possible exit codes are the following:

- **0: Uninstallation fully successful**.
- **2: Incomplete uninstallation**. Certain parts of the installation could not have been removed (for instance, the ActiveX controls were not unregistered or some IDE could not be unconfigured).
- **3: Reboot needed**. Certain parts of the installation could not have been immediately removed, but are scheduled for deletion on the next reboot. It is recommended to reboot the computer as soon as possible.
- **4: The user is not an administrator**. It is impossible to carry on the uninstallation. Run the uninstaller using an administrator account.
- **5: The product is not installed**. The uninstallation cannot be achieved, as the product is not installed on the computer.

5. Licenses Activation

Once you have installed Open eVision Libraries, or Open eVision Development Tools, you must activate licenses in order to use the products you purchased. **Open eVision License Manager** is the application that allows you to activate the purchased Open eVision licenses, with an online or an offline procedure. It is automatically launched at the end of the setup, so that you can directly finalize all the steps before using Open eVision.

Anyway, if you didn't activate your licenses yet, access Open eVision License Manager through the Euresys Open eVision shortcut created in the Windows start menu. Click the help button to access the **Open eVision License Manager Documentation**. You will find there complete assistance, especially about offline licenses activation.

Open eVision License Manager home page

6. Third-Party Libraries or Components

FreeImage

FreeImage is a library that allows loading and saving images in several standard formats.

This software uses the FreeImage open source image library. See http://freeimage.sourceforge.net for details. FreeImage is used under the FIPL, version 1.0. For more information about the licensing refer to the following text: freeimage-license.

Intel IPP and Intel MKL

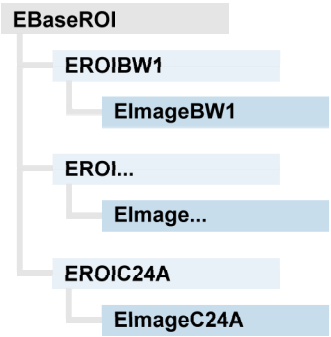
- This software uses the Intel® Integrated Performance Primitives.
  See http://software.intel.com/en-us/articles/intel-ipp/ for details.
- This software uses the Intel® Math Kernel Library.
  See http://software.intel.com/en-us/articles/intel-mkl/ for details.

Basic Types and Operations

To work with Open eVision, a good understanding of the basic types is necessary. These basic types are used in any of the other Open eVision components.

- **Images**, used to handle images,
- **ROIs**, used to delimit rectangular areas from images,
- **Vectors**, used to store rows of pixels.
- **Pixels**
- **Colors**

Image and ROI classes derive from an abstract class named **EBaseROI**, and they inherit from all its properties.



1. Images

The Open eVision building blocks are handling images, or parts of images (ROIs). The image data information is stored in an Open eVision image object. These image objects are used by all other components of Open eVision. They are the main objects you will be dealing with for processing and analysis. The Open eVision image objects encapsulate all the functionalities needed to represent rectangular images.

1.1 Image Main Properties

An Open eVision image object holds a rectangular array of pixels. It is characterized by a few parameters inherited from the abstract class EBaseROI.

The width of the image is the number of pixels per row (per line) of the image.

The height of the image is the number of rows (number of lines) of the image.

The maximum value for each image dimension (width and height) is 32,767 (2^{15}-1) in Open eVision 32-bit, and 2,147,483,647 (2^{31}-1) in Open eVision 64-bit.

Their **depth** tells how many bits are used to encode the value of each pixel.

The number of planes indicates how many spectral components are considered. For gray-level images, this number is always 1. For color images, it is always 3.

The **size of an image** is specified as a number of columns (**width**) and rows (**height**).

Additional information can be associated to an image, such as a title, a creation date, an author name and a comment.

An image object has an associated data **buffer**, accessible via a pointer, where the pixel values are stored contiguously, row by row.

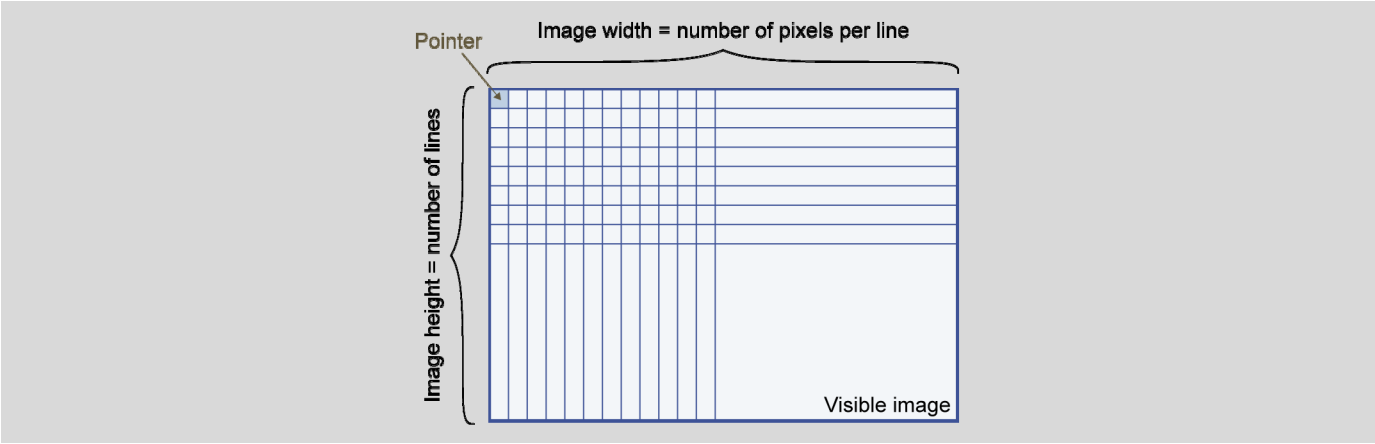


Image size

1.2 Image Types

Several images types are supported according to their pixel types (number of bytes per pixel): black and white, gray levels, color, etc.

Image type	Image class	Description
BW1	**EImageBW1**	1-bit black and white images (8 pixels are stored in one byte)
BW8	**EImageBW8**	8-bit grayscale images (each pixel is stored in one byte)
BW16	**EImageBW16**	16-bit grayscale images (each pixel is stored in two bytes)
BW32	**EImageBW32**	32-bit grayscale images (each pixel is stored in four bytes)
C15	**EImageC15**	15-bit color images (each pixel is stored in two bytes) This format is compatible with the Microsoft® Windows RGB15 color images and MultiCam RGB15 format.
C16	**EImageC16**	16-bit color images (each pixel is stored in two bytes) This format is compatible with the Microsoft® Windows RGB16 color images and MultiCam RGB16 format.
C24	**EImageC24**	C24 images store 24-bit color images (each pixel is stored in three bytes). This format is compatible with the Microsoft® Windows RGB24 color images and MultiCam RGB24 format.
C24A	**EImageC24A**	C24A images store 32-bit color images (each pixel is stored in four bytes). This format is compatible with the Microsoft® Windows RGB32 color images and MultiCam RGB32 format.

Note. Easy.GetBestMatchingImageType( ) returns the best matching image type for a given file on disk.

1.3 Image Files

Supported Image File Types

BMP

The **BMP image file type** uses an uncompressed image data format also known as the Windows Bitmap Format.

JPEG

The **JPEG image file** type uses a data compression standard issued by the Joint Photographic Expert Group and registered as ISO/IEC 10918-1. The compression method is lossy, meaning that some visual quality is lost in the process and cannot be restored.

The Open eVision implementation uses the **JPEG File Interchange Format – JFIF** – file format. The **JFIF** file format is widely accepted by many applications.

JPEG-2000

The **JPEG-2000 image file** type uses a data compression standard issued by the Joint Photographic Expert Group in the year 2000 and registered as ISO/IEC 15444-1 and ISO/IEC 15444-2.

The JPEG-2000 data compression algorithm supports both lossless and lossy compression. The Open eVision implementation supports the lossy compression exclusively.

JPEG2000 defines both a **file format** and a code **stream**: The code stream describes the image samples. The **file format** includes additional meta-information such as the resolution of the image or the color space that has been used to encode the image. The Open eVision implementation supports both variants.

PNG

The **Portable Network Graphics – PNG – image file type** uses a lossless data compression method. The PNG format supersedes the GIF format. Unlike GIF, PNG doesn't require any license.

TIFF

The **Tag Image File Format – TIFF – image file type** uses a file format currently controlled by Adobe Systems.

The current implementation uses the version of the LibTIFF third-party library. The library is capable of dealing with images that are written to follow the 5.0 or 6.0 TIFF specification.

File save operations uses CCITT 1D compression for 1-bit binary pixel types and LZW compression for other pixel types. Both compressions are lossless.

File load operations supports the TIFF variants listed in the LIBTIFF specification coverage.

Serialized

The serialized image file type uses a Euresys proprietary image file format obtained from the **serialization** of the Open eVision image objects.

Image File Access - Save, Load

You can save an image to disk and/or load it from the disk, using different formats described above. When loading, the image size is automatically adjusted.

Saving an Image into a File

Save saves the image data of an image object into a file. It is applicable to all image types. Two arguments are available:

- The first argument specifies the path: a string of characters including the path, filename, and file name extension.
- The second argument specifies the type of the image file. When the second argument is omitted, the file type is automatically determined by the filename extension given in the first argument.

The type of the image file is determined according to the value of the second argument:

Value of second argument	Image file type
EImageFileType_Auto**(*)**	Automatically determined by the filename extension. See below.
EImageFileType_Euresys	Open eVision Serialization format.

EImageFileType_Bmp	Windows bitmap format -BMP
EImageFileType_Jpeg	JPEG File Interchange Format - JFIF
EImageFileType_Jpeg2000	JPEG 2000 File format -JPEG2000
EImageFileType_Png	Portable Network Graphics - PNG
EImageFileType_Tiff	Tagged Image File Format - TIFF

(*) Default value.

The following table shows the list of automatically assigned image file types when the second argument is **ImageFileType_Auto** or missing:

File name extension(*)	Automatically assigned image file type
BMP	Windows Bitmap Format
JPEG, JPG	JPEG File Interchange Format - JFIF
JP2	JPEG 2000 file format
J2K, J2C	JPEG 2000 Code Stream
PNG	Portable Network Graphics
TIFF, TIF	Tagged Image File Format

(*) Case-insensitive.

Save throws an exception when:

- The requested image file format is incompatible with the image pixel types
- The Auto file type selection method and the file name extension is not supported

Save uses exclusively default compression quality settings for both JPEG and JPEG2000 lossy compressions.

SaveJpeg and SaveJpeg2K provide the capability to specify the compression quality when saving images into a compressed file format. They have two arguments:

- The first argument specifies the path: a string of characters including the path, filename, and file name extension.
- The second argument specifies the compression quality of the image file. The **SaveJpeg2K** function saves the image data using the JPEG File Interchange Format – JFIF.

The compression quality argument is an integer value in range [0: 100].

Representative JPEG compression quality values

JPEG compression quality value	Description
JPEG_DEFAULT_QUALITY (-1)	Default quality (*)
100	Superb image quality, lowest compression factor
75	Good image quality (*)
50	Normal image quality
25	Average image quality

10	Bad Image quality

(*) The default quality corresponds to the good image quality (75).

The SaveJpeg2K function saves the image data using the JPEG 2000 File format. The compression quality argument is an integer value in range [1:512].

Representative JPEG 2000 compression quality values

JPEG 2000 compression quality value	Description
-1	Default quality (*)
1	Highest image quality, lowest compression factor
16	Good Image Quality (*) (16:1 rate)
512	Lowest image quality, highest compression factor

(*) The default quality corresponds to the good image quality (16:1 rate).

Being given the size limitations of the file formats listed above, the only way to load or save images bigger than 65,536 (either width or height) is to use the Open eVision proprietary format.

Loading Images from Files

The Load method loads image data into an image object from a file. It is applicable to all image types. It has one argument: the path: a string of characters including the path, filename, and file name extension.

The Load method determines automatically the file type using a two-step procedure: As a first step, it searches for a known file format signature in the file data. If a valid signature is not found, it uses the file name extension to identify the image file according to the following table:

File name extension(*)	Automatically assigned image file type
BMP	Windows Bitmap Format
JPEG, JPG	JPEG File Interchange Format - JFIF
JP2	JPEG 2000 File Format
J2K, J2C	JPEG 2000 Code Stream
PNG	Portable Network Graphics
TIFF, TIF	Tagged Image File Format
<Any other extension>	Open eVision proprietary serialized format

(*) Case-insensitive.

The destination image is automatically resized according to the size of the image on disk.

The Load method throws an exception when:

- The file type identification fails
- The identified file type is incompatible with the pixel types of the image object

Note. The serialized image files of Open eVision 1.1 and further version are not compatible with the serialized image files of previous Open eVision and eVision versions.

1.4 Image Pixel Structures and Image File Types Compatibility

Most of the combinations of Image Pixel Types and Image File Types are supported by Open eVision during Image Load or Image Save operations. The following table provides an overview of the compatibility and reports possible compatibility issues:

Image file types vs. pixel types

Pixel format	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
BW1	Ok	N/A	N/A	Ok	Ok	Ok
BW8	Ok	Ok	Ok	Ok	Ok	Ok
BW16	N/A	N/A	Ok	Ok	Ok (***)	Ok
BW32	N/A	N/A	N/A	N/A	Ok (***)	Ok
C15	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C16	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C24	Ok	Ok	Ok	Ok	Ok (**)	Ok
C24A	Ok	N/A	N/A	Ok	N/A	Ok

N/A: this combination of pixel format/file type in not supported. An exception occurs if you use the combination.

Ok: this combination of pixel format/file type is allowed for both save and file load operations, the image integrity is preserved with no data loss. However, for JPEG and JPEG2000, the image integrity is not preserved, resulting in some lost visual quality during the file save process and it can not be restored.

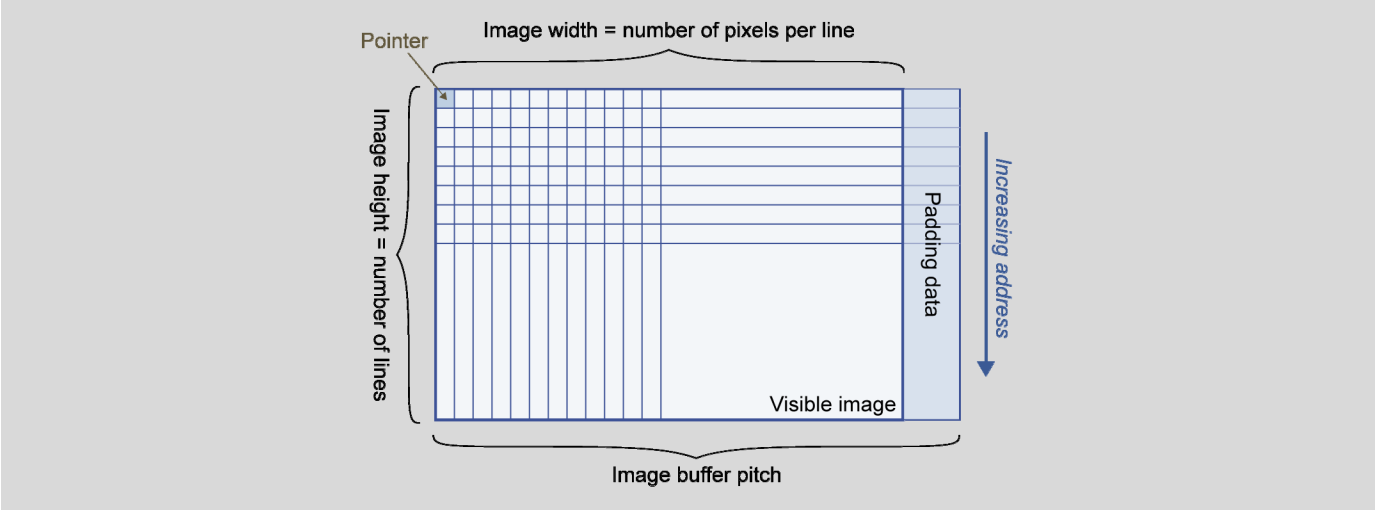
JPEG and JPEG2000 use lossy compression, the loss of the information depends upon the selected quality.

(**) C15 and C16 pixel formats are automatically converted into C24 during the file save operation.

(***) BW16 and BW32 are not supported by Baseline TIFF readers.

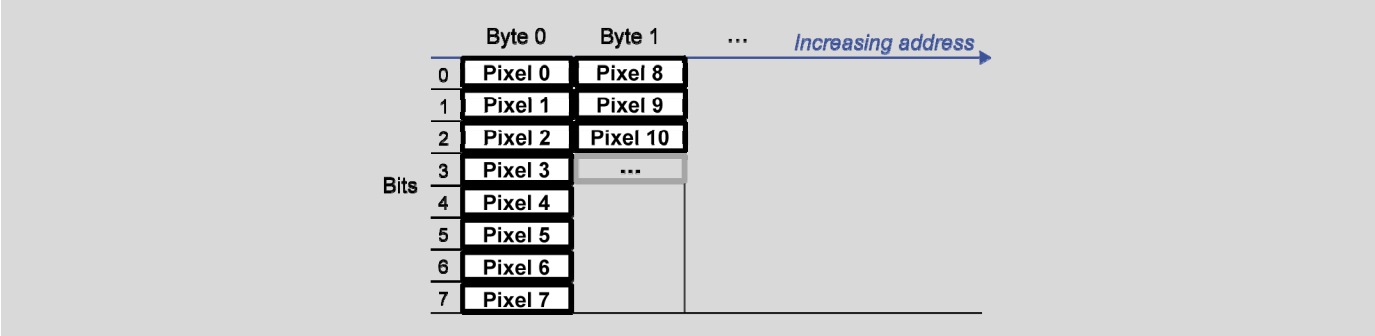
1.5 Image Buffer

The address of the buffer is a pointer pointing to the beginning (lowest) address of the buffer, which contains the top left pixel of the image. The next pixels are stored contiguously, row by row, from left to right then from top to bottom.



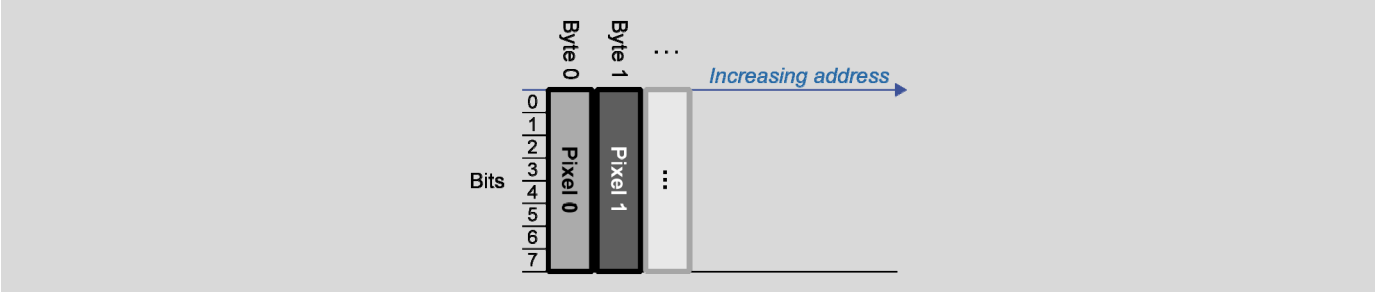
Memory Layout

- For **EImageBW1**, 8 pixels are stored in one byte.



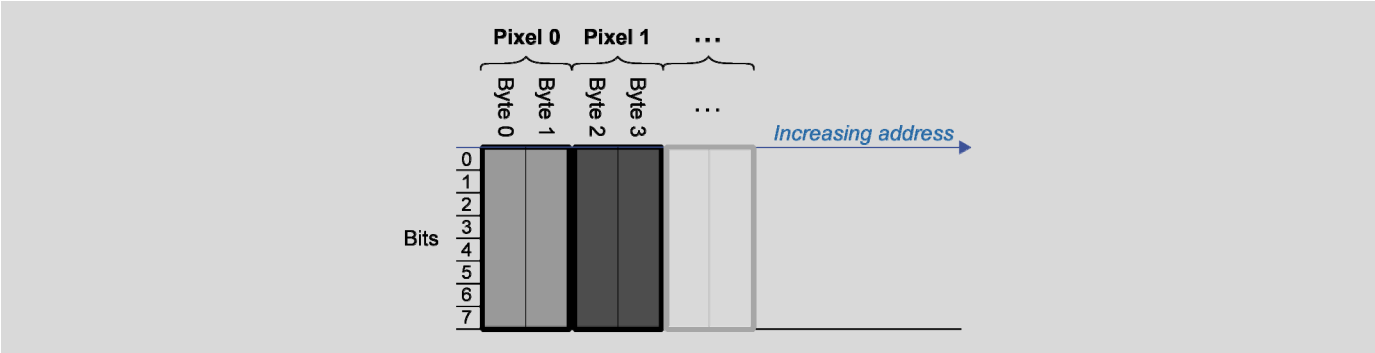
Memory layout of the first 2 pixels of a BW1 image buffer

- For **EImageBW8**, each pixel is stored in one byte.



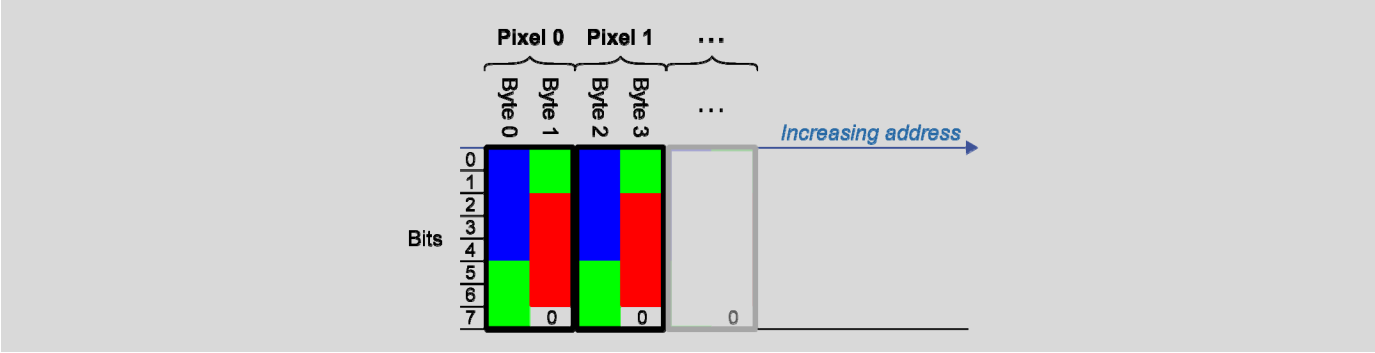
Memory layout of the first pixels of a BW8 image buffer

- For **EImageBW16**, each pixel is stored in a 16-bit word (two bytes).



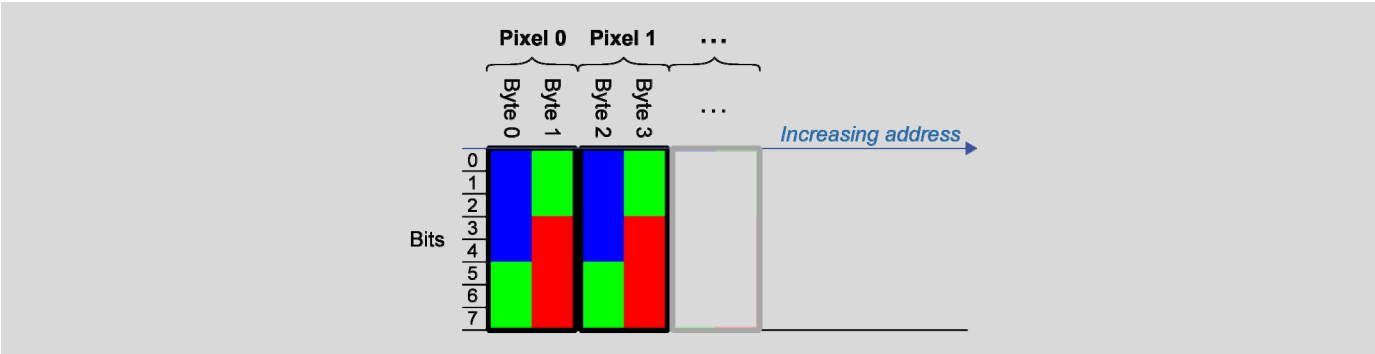
Memory layout of the first pixels of a BW16 image buffer

- For **EImageC15**: the C15 format is a packed three-channel image format that requires 2 bytes per pixel. Each color component is coded with 5-bits. The 16th bit is left unused.



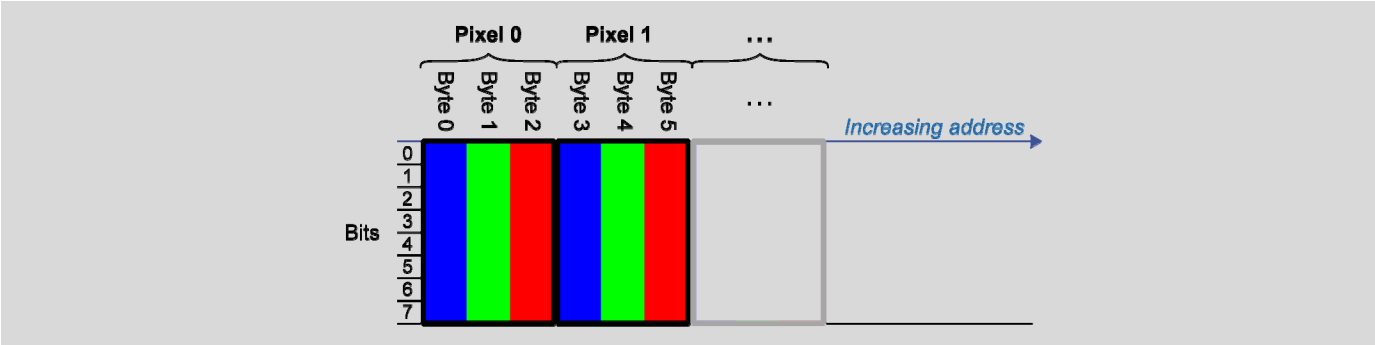
Memory layout of the first pixels of a C15 image buffer

- **EImageC16**: The C16 format is a packed three-channel image format that requires 2 bytes per pixel. The first and third color components are coded with 5-bits. The second color component is coded with 6-bits.



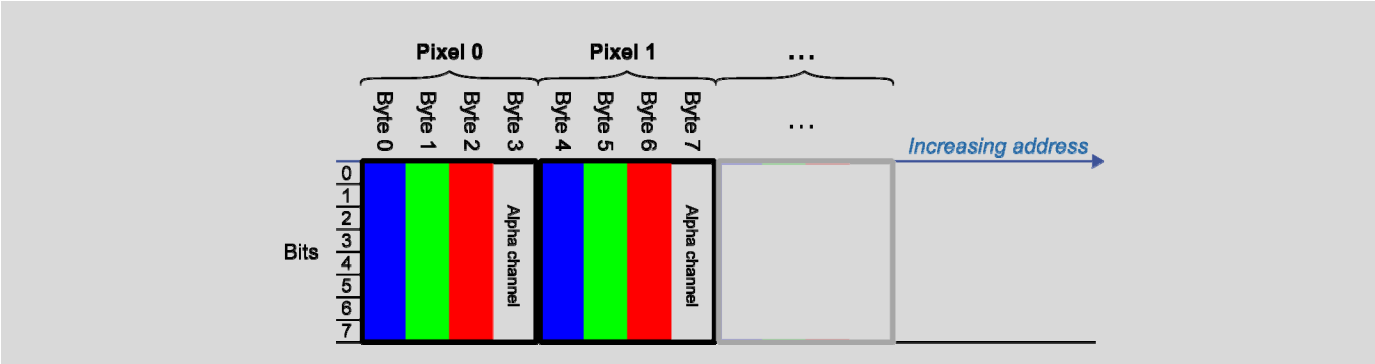
Memory layout of the first pixels of a C16 image buffer

- **EImageC24**: The C24 format is a packed three-channel image format that requires 3 bytes per pixel. Each color component is coded with 8-bits.



Memory layout of the first pixels of a C24 image buffer

- **EImageC24A**: The C24A format is a packed four-channel image format (the fourth channel commonly stores the alpha channel) that requires 4 bytes per pixel. Each color component is coded with 8-bits. The alpha channel is also coded with 8-bits.



Memory layout of the first pixels of a C24A image buffer

Pitch of an Image Buffer

From Open eVision 1.2 onwards, the default line alignment has been set to 32 bytes (it was set to 8 bytes in Open eVision 1.1.5). It has been made for optimization purposes regarding SSE2 acceleration.

Although Open eVision will support external buffers with a smaller alignment, the best performance is reached when using a 32-byte alignment. Please note that the alignment must always be a multiple of 4 bytes.

1.6 Image Construction and Memory Allocation

An image has an associated pixel data area in which the pixels are stored contiguously, row by row, from top to bottom and from left to right, following the Windows bitmap format definition (top-down DIB).

To construct an image with reading memory allocation, two strategies can be used: with an internal or with an external memory allocation.

Construct and Access an Image with an Internal Memory Allocation

The internal allocation lets the image object dynamically allocate and de-allocate the required buffer.

When the image size is changed, reallocation occurs. When the image object is destroyed, the buffer is de-allocated as well. Memory management is transparent.

There are two ways to construct an image with an internal memory allocation:

- Declare an image object with the size as an argument for instance **EImageBW8**.
- Or declare an image object **EImageBW8** and set the size with the **SetSize** function.

To access a particular pixel of the image, several functions are available. GetImagePtr returns a pointer to the first byte of the pixel at the given coordinates.

Alternatively, **Get/SetPixel** allows you to read/write the pixel value at the given coordinates.

To scan the contents of an image (to implement a custom image processing function), an easy way is to run a double loop on the X and Y values and call the **GetPixel** or **SetPixel** function each time.

This is not recommended for compute-intensive tasks because of the overhead incurred by each function call.

Construct and Access an Image with External Memory Allocation

The External Memory allocation allows the user to control the buffer allocation himself or to link a third-party image stored in the PC memory buffer to an Open eVision image.

In order for the image object to access the user buffer appropriately, the image size and buffer address must be specified. When external allocation is used, destroying the image object has no effect on the buffer allocation.

The address of the buffer is a pointer pointing to the beginning (lowest) address of the buffer, which contains the top left pixel of the image. The next pixels are stored contiguously, row by row, from left to right then from top to bottom.

- Declare an image object, for instance **EImageBW8**
- Create a buffer with the suitable size. Do not forget the alignment requirements (see "Pitch of an Image Buffer")
- Set the image size with the **SetSize** function
- Via a pointer, access the data buffer associated to the image with the **SetImagePtr** function. See Image Buffer for details about the data buffer.

1.7 Image Drawing and Overlay

Image drawing is based on the Windows GDI (Graphics Device Interface) system calls. This provides total device independence and ease of use. If needed, the system color palette is adjusted (in 256 color display mode) to allow optimal rendering of gray-level images as well as color images.

Optionally, gray-level images can be displayed using a gray-level or color lookup table for improved results by using histogram stretching techniques or pseudo-coloring.

The drawing can be zoomed by arbitrary factors (in the range 1/16..16), possibly different horizontally and vertically.

In an MFC application, drawing normally occurs in an OnDraw event handler, where a pointer to a device context is available. (In Borland/CodeGear's OWL or VCL, the event handler is named Paint.)

Drawing Functions

The drawing functions set a color automatically.

For each Draw method that uses pen drawing, a DrawFrameWithCurrentPen method allow to use a custom pen (for custom color, patterns or pen widths, for instance).

Destructive Overlaying

After an image has been drawn, any other drawing operation can be used, such as the **MoveTo**/**LineTo** GDI functions, or other Open eVision drawing functions. This is called **non-destructive overlaying**, since it does not alter the image contents at all.

On the other hand, **destructive overlaying** alters the image contents by drawing inside the image. Of course, a gray-level [color] image can only receive a gray-level [color] overlay.

The Easy::OpenImageGraphicContext function allows drawing in a gray-level or a color image, using any of the Windows device context functions (the image content is altered, allowing destructive overlays).

The function returns a handle to a device context associated to the image pixel data. When the device context is no more needed, call function Easy::CloseImageGraphicContext with the same argument.

Graphical Interaction with Images

The **MouseDown**, **MouseMove**, **MouseUp** events can be handled to implement graphical interaction, such as adjusting an ROI position.

Note. In case of VB, these functions return the current cursor position in twips rather than pixels. A unit conversion is mandatory.

The EBaseROI::HitTest method informs if the cursor is placed over a dragging handle. Once the handle is known, the cursor shape can be changed. EROIXXX::Drag allows updating the ROI position based on the new handle position.

2. ROIs

A **region of interest (ROI)** of an image is a rectangular zone within the image. The processing of an image can be accelerated by focalizing on such a Region of Interest avoiding interferences from the remainder of the image. The number of pixels to consider is then reduced.

The processing of all Open eVision functions can be restricted to a Region of Interest (ROI). Open eVision supports nested rectangular ROIs, which are organized in a hierarchical way in each image.

When the area of interest has a disconnected or complex shape or if it has a shape different from a rectangle, it is recommended to mask the image using the Open eVision flexible masks. They are supported for selected functions of the EasyObject and EasyImage libraries.

2.1 ROI Main Properties

The Open eVision ROI is characterized by a few parameters inherited from the abstract class EBaseROI.

ROIs are defined by a width, a height, and origin x and y coordinates. The position of the origin point is specified with respect to the top left corner in the parent image. The ROI must be wholly contained in its parent image.

It is also possible to save or load an ROI of an image rather than a full image. When loading in an ROI, the image size in the file must match the ROI size (automatic resizing of the ROI might come out with unwanted results). The image contents around the ROI remain unchanged.

ROIs can be used anywhere and passed to processing functions, as if they were full images.

2.2 ROI Types

There are several ROI types, according to their pixel type: black and white, gray levels, color, etc. They have the same characteristics than the corresponding image types.

ROI class
EROIBW1
EROIBW8
EROIBW16
EROIBW32
EROIC15
EROIC16
EROIC24
EROIC24A

Note. The processing/analysis time of a BW1 ROI is faster if your ROI properties, **OrgX** and, **Width** are multiple of 8.

2.3 ROI Construction and Attachment

An ROI is attached to a parent (image or ROI).

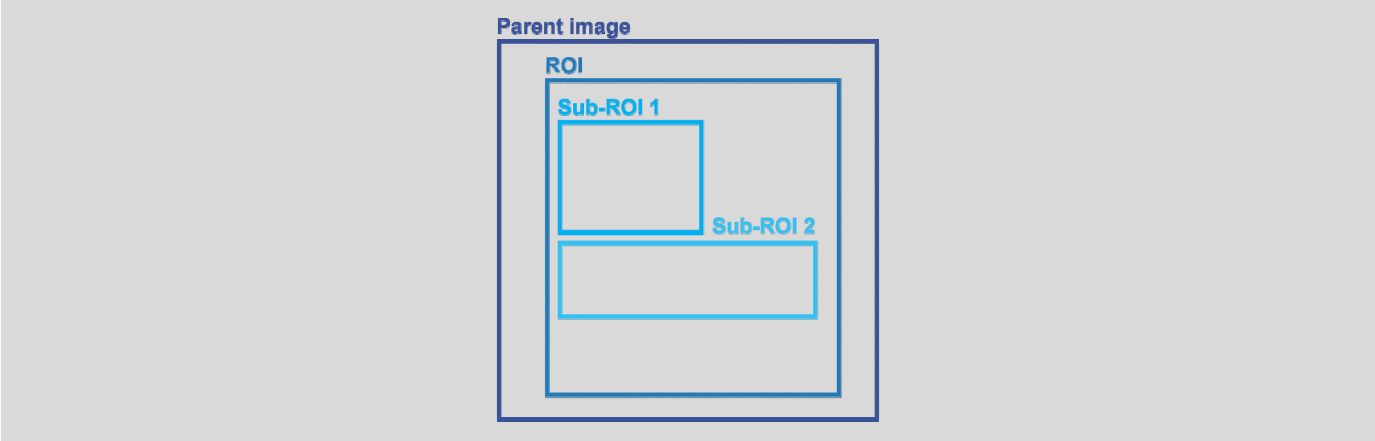
Attach methods allow to attach an ROI to an image (or another ROI). It has parameters allowing to set the parent, position and size in one shot.

Only ROIs can be attached. An image can never be attached to another image or ROI.

When attaching ROIs to another, the image/ROI classes update the links between them in a transparent way, so that dangling pointers are avoided, and the operation is always safe.

Nested ROIs

An image may accommodate an arbitrary number of ROIs, which can be nested in a hierarchical way. Moving the ROI will cause the embedded ROIs to move accordingly.



Nested ROIs. Two sub-ROIs are attached to an ROI, itself attached to the parent image

Setters and Getters functions allow you to change or query the width, height and position of the origin of an **ROI**, with respect to its immediate parent or, with respect to the topmost parent image.

The image/ROI classes provide several methods to traverse the hierarchy of ROIs associated with an image.

To crop an ROI which is partially out of its image, call CropToImage explicitly. The resized ROI never grows. An exception will be thrown by any function attempting to use an ROI having limits that extend outside of the parents.

Note. In Open eVision 1.0.1 and in earlier versions, an ROI was silently resized or repositioned when placed out of its image; in some cases, when automatically resizing, the ROI could grow. If one specifies ROI limits that extend outside of the parents, these were silently resized or repositioned to remain within the parent limits.

The ROIs perform **no memory allocation** at all and never duplicate parts of their parent image. On the opposite, the parent image provides them with suitable access to its own image data.

2.4 Graphical Interaction

ROIs can easily be resized and positioned by means of dragging handles. Two functions are provided:

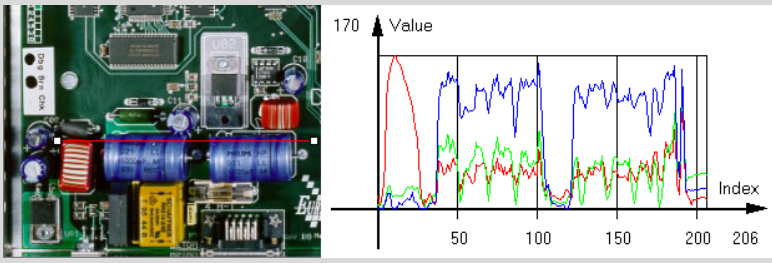
- **HitTest** informs if the cursor is placed over a dragging handle. Once the handle is known, the cursor shape can be changed.
- **Drag** adjusts the ROI coordinates while the cursor moves.

HitTest can be used in an OnSetCursor MFC event handler to change the cursor shape. It should also be used at the start of a dragging operation, such as in OnLButtonDown. (Under Borland/CodeGear's OWL, corresponding event handlers are EvSetCursor and EvLButtonDown. Under Borland/CodeGear's VCL, use FormMouseMove and FormMouseDown.)

Note. HitTest may not be called while dragging is in progress, otherwise the inside handle will not operate as expected.
Note. When using Visual Basic 6, the MouseDown, MouseMove and MouseUp events return the current cursor position in twips rather than pixels. A unit conversion is mandatory.

3. Vectors

A vector is a one-dimensional array of pixels taken from an image (profile or contour). Memory allocation is handled in a transparent way, so that the vectors can be resized dynamically.



Index	Red	Green	Blue
0	15	5	3
1	7	4	0
2	5	8	0
3	9	5	0
4	29	1	0
5	55	6	9
6	120	15	9
7	139	24	17
8	157	26	18
9	161	17	6
10	165	13	0
11	170	14	1

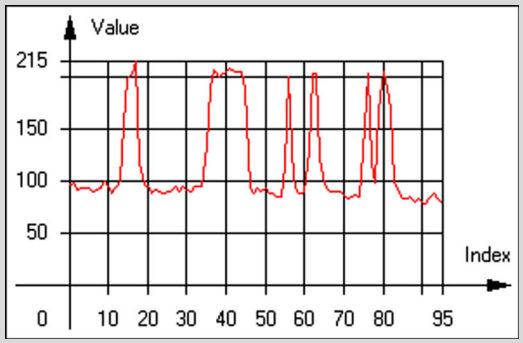
Profile in a C24 image. Plot of the R, G, B pixel values along the profile. Array of R, G, B values (EC24Vector**)**

There are several vector classes/controls, according to their pixel type: black and white, gray levels, color, etc. EVector is the base class for all types of vectors. This class contains all methods that are not type-specific, mainly methods to handle elements count and serialization.

3.1 Vectors Types and Main Properties

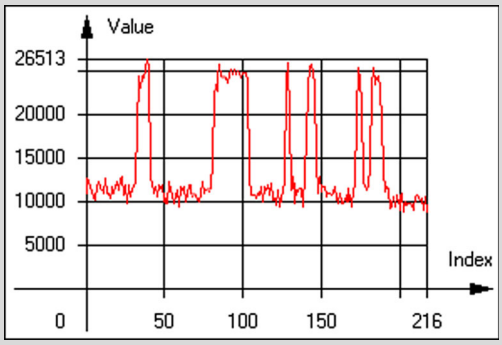
Each vector class has a specific purpose, as follows:

- **EBW8Vector** represents a sequence of gray-level pixel values, often extracted from an image profile (used by **EasyImage::Lut**, **EasyImage::SetupEqualize**, **EasyImage::ImageToLineSegment**, **EasyImage::LineSegmentToImage**, **EasyImage::ProfileDerivative**, ...).



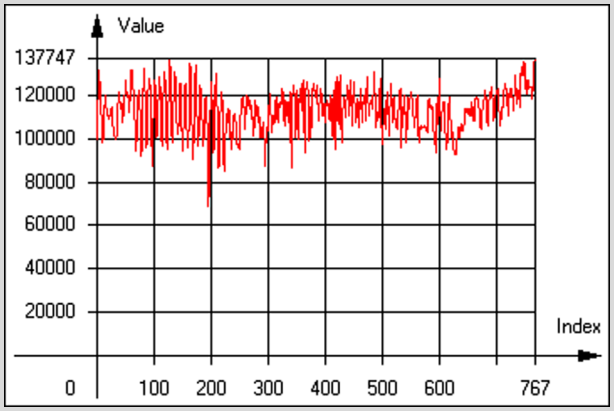
Graphical representation of an EBW8Vector **(see** Draw **method)**

- **EBW16Vector** represents a sequence of gray-level pixel values, using an extended range (16 bits), mainly for intermediate computations.



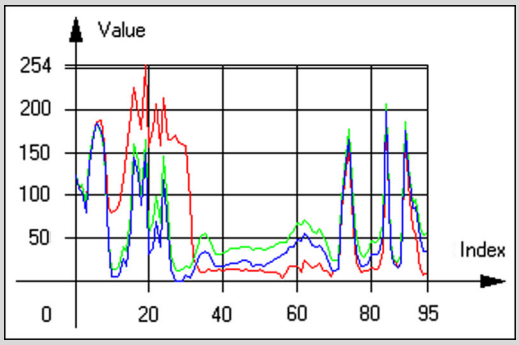
Graphical representation of an EBW16Vector **(see** Draw **method)**

- **EBW32Vector** represents a sequence of gray-level pixel values, using an extended range (32 bits), mainly for intermediate computations (used in **EasyImage::ProjectOnARow**, **EasyImage::ProjectOnAColumn**, ...).



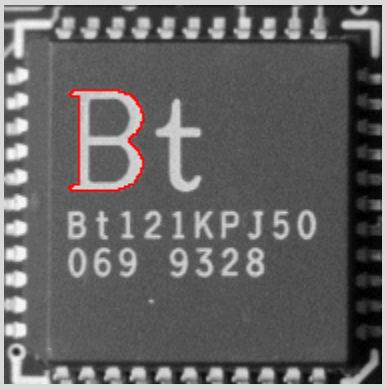
Graphical representation of an EBW32Vector **(see** Draw **method)**

- **EC24Vector** represents a sequence of color pixel values, often extracted from an image profile (used by **EasyImage::ImageToLineSegment**, **EasyImage::LineSegmentToImage**, **EasyImage::ProfileDerivative**, ...).



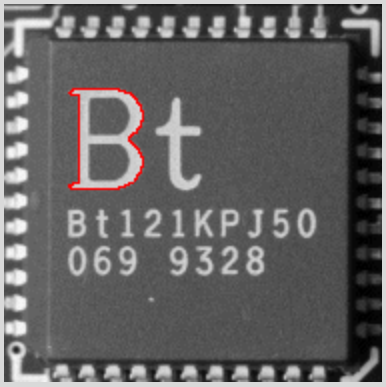
Graphical representation of an EC24Vector **(see** Draw **method)**

- **EBW8PathVector** represents a sequence of gray-level pixel values, extracted from an image profile or contour, along with the corresponding pixel coordinates (used by **EasyImage::ImageToPath**, **EasyImage::PathToImage**, ...).



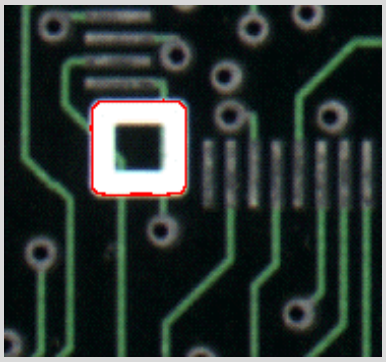
Graphical representation of an EBW8PathVector **(see** Draw **method)**

- **EBW16PathVector** represents a sequence of gray-level pixel values, extracted from an image profile or contour, along with the corresponding pixel coordinates (used by **EasyImage::ImageToPath**, **EasyImage::PathToImage**, ...).



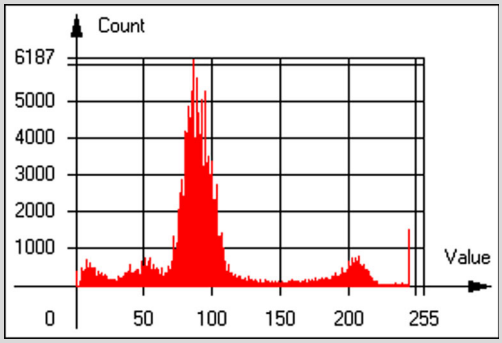
Graphical representation of an EBW16PathVector **(see** Draw **method)**

- **EC24PathVector** represents a sequence of color pixel values, extracted from an image profile or contour, along with the corresponding pixel coordinates (used by **EasyImage::ImageToPath**, **EasyImage::PathToImage**, ...).



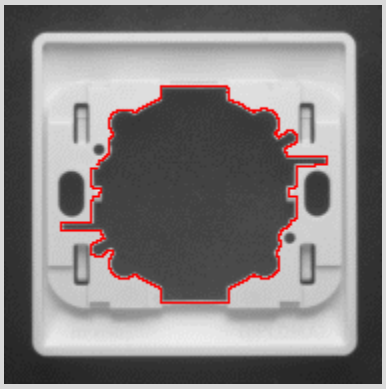
Graphical representation of an EC24PathVector **(see** Draw **method)**

- **EBWHistogramVector** represents a sequence of frequency counts of pixels in a BW8 or BW16 image (used by **EasyImage::IsodataThreshold**, **EasyImage::Histogram**, **EasyImage::AnalyseHistogram**, **EasyImage::SetupEqualize**, ...).



Graphical representation of an EBWHistogramVector **(see** Draw **method)**

- **EPathVector** represents a sequence of pixel coordinates. The corresponding pixels need not be contiguous (used by **EasyImage::PathToImage** and **EasyImage::Contour**).



Graphical representation of an EPathVector **(see** Draw **method)**

- **EPeakVector** represents a description of peaks found in an image profile (used by **EasyImage::GetProfilePeaks**).
- **EColorVector** represents a description of colors (used by **EasyColor::ClassAverages** and **EasyColor::ClassVariances**).

The use of vectors is quite straightforward: create a vector of the appropriate type, possibly pre-allocating a number of elements. You can add elements, or remove all of them. You can access any element by means of indexing.

Note. 32-bit vectors are used for intermediate computations.

3.2 Vector Construction and Usage

A vector manages an array of elements. Memory allocation is handled in a transparent way, so that the vectors can be resized dynamically.

- Create a vector of the appropriate type, using its constructor and possibly pre-allocating a number of elements
- Fill a vector with values, first empty it, using the **EVector::Empty** member, and then add elements one at a time by calling the **EC24Vector::AddElement** member.
- Access a vector element, either for reading or writing, use the brackets operator, for instance, **EC24Vector::operator[]**.
- Enquire for the current number of elements, use member **EVector::NumElements**.

Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs, if necessary.

Note. 32-bit vectors are used for intermediate computations.

3.3 Vector Drawing

A vector is able to draw itself in a window. Its graphical appearance depends on its type. In the case of pixel vectors, it is a plot of the element values as a function of the element index.

The following parameters can be defined: graphicContext, width, height, origin, origin, color0, color1, color2.

For legibility, the drawing should appear on a neutral background.

Drawing is done in the device context associated to the desired window. By default, the curves are drawn using a blue pen. To draw the annotations, a black pen is used instead.

In the special case of the EC24Vector, three curves are drawn instead of one, each corresponding to a color component. By default, red, blue and green pens are used respectively.

4. Pixels

4.1 Pixel Types

Note. Before eVision 6.5, the image pixel types were defined using typedef of integral types.

Pixel structures
EBW1
EBW8
EBW32
EC15 (*)
EC16 (*)
EC24 (*)
EC24A

(*) These color image formats have been introduced to support the Windows formats known as RGB15 (5-5-5 bit packing), RGB16 (5-6-5 bit packing) and RGB32 (RGB + alpha channel). These can be displayed and converted to/from EC24 using EasyImage::Convert. Beware that no processing functions apply to such image formats, for efficiency reasons. Explicit conversion is mandatory.

Note. The pixel values are always treated as unsigned (non-negative) numbers.
Note. Implicit conversion to/from the previous integral pixel types is provided, so that the transition should be seamless. In case you are using pointers to integral pixel types, you can safely cast the newly provided pointer to the structure(s) to a pointer to the integral type.

4.2 Pixel Data Access

Get/SetPixel allows you to read or write the pixel value at the given coordinates.

To **scan the contents of an image** (to implement a custom image processing function), an easy way is to run a double loop on the X and Y values and use **GetPixel** or **SetPixel** each time.

- Anyway, this is not recommended for compute-intensive tasks because of the overhead incurred by each function call.

5. Colors

Color types

Color type	Description
EISH	Intensity, Saturation, Hue color system.
ELAB	CIE Lightness, a*, b* color system.
ELCH	Lightness, Chroma, Hue color system.
ELSH	Lightness, Saturation, Hue color system.
ELUV	CIE Lightness, u*, v* color system.
ERGB	NTSC/PAL/SMPTE Red, Green, Blue color system.
EVSH	Value, Saturation, Hue color system.
EXYZ	CIE XYZ color system.
EYIQ	CCIR Luma, Inphase, Quadrature color system.
EYSH	CCIR Luma, Saturation, Hue color system.
EYUV	CCIR Luma, U Chroma, V Chroma color system.

EasyImage

1. What Is EasyImage?

EasyImage includes operations extracting global information and also operations usually performed as pre-processing steps to improve the image quality and obtain a good contrast between the background and the objects to be inspected. For instance, the pre-processing step will remove noise, compensate variable lighting conditions, and correct geometric distortions. These operations help make higher-level tasks more robust, repeatable and accurate. Typical applications are enhancement of poorly contrasted images by linear or non-linear filtering, defect isolation by thresholding, compensation of perspective effects for non-flat surfaces, filtering for visual aspect improvement. EasyImage supports gray level and color images. Selected morphology functions are also optimized for binary (1-bit per pixel) and bi-level images.

EasyImage supports the restriction of the processing to rectangular ROIs. For selected functions, it supports also the restriction of the processing to complex or disconnected-shape regions of the image through the flexible masks.

The EasyImage library contains many general functions for image processing such as:

- **Intensity scale transformation functions**:
  - **Gain/Offset** change: To adjust the contrast of an image, the gray value at each pixel is simply multiplied by a gain coefficient and offset by a constant value.
  - **Normalization**: Automate the contrast adjustment; i.e. let the average gray level and its standard deviation be equal to pre-defined values.
  - **Uniformization**: A linear method to compensate for non-uniform illumination and/or camera sensitivity
  - **Lookup mapping**: For more complex transformations, a correspondence table is filled once with the new pixel values to be substituted to the current ones. This way, the transformation of a whole image can be done very efficiently.
- **Thresholding:**
  - Automatic thresholding
    - Min residue
    - Max entropy
    - Isodata
  - Manual thresholding
    - Single threshold (absolute and relative)
    - Double threshold
  - Histogram-based threshold
- **Arithmetic and logic operations**: pixel-wise arithmetical and logical combinations between two images or between an image and a constant.
  - Arithmetic operations:
    - Add, subtract
    - Multiply, divide
    - Copy
    - Invert, module, shift
  - Logical and bitwise operations:
    - AND
    - OR
    - XOR
    - NOT

- Minimum, maximum
- Pixel compare
- Histogram equalization

- **Convolution**: linear combination of neighboring pixels using a convolution kernel.
  - Pre-defined filters for
    - Edge detection: Laplacian, Gradient, Prewitt, Sobel, Roberts
    - Sharpening: several high-pass filters
    - Smoothing: several low-pass including Gaussian filter and uniform filters
  - Custom kernel filtering: Kernel creation and management functions
- **Non-linear filtering**: non-linear combinations of neighboring pixels
  - Morphological operators
    - Erosion, dilation
    - Opening, closing
    - Thinning, thickening
    - Top-hat filters
    - Morphological distance
    - Hit-and-miss transform: detects a particular pattern of foreground and background pixels in an image
  - Median filter
- **Geometric transformations**: displacement of the image pixels
  - Image registration (alignment)
  - Horizontal and vertical mirroring
  - Translation, scaling and rotation with optional interpolation
  - LUT-based (un)warping
- **Vector operations:** extraction of 1-dimensional data from an image. It is sometimes convenient to extract 1-dimensional data from an image. Think of extracting a profile along a given line segment to detect peaks, drawing iso-level curves to describe the shape of an enclosed object, count the frequencies of every gray value to detect the presence or absence of a given object... These operations generate linear sets of data that are handled as so-called vectors. Subsequent operations on vectors are fast because of the reduced amount of data.
  - Projection: sum of all gray-level values in a given direction vector
  - Profile: sampling(line segment, path, contour) and analysis
- **Statistics**
  - Measurement of:
    - Area, binary moments
    - Weighted moments
    - Gravity center
    - Pixel count and pixel statistics
    - Minimum and maximum gray-level value
    - Average, variance and standard deviation
    - Histogram computation and analysis
    - Image focus

- **Noise reduction and estimation**
  - Spatial noise reduction:
    - Convolution
    - Median filters
  - Temporal noise reduction:
    - Recursive average
    - Moving average
    - Average
  - Noise estimation:
    - Root-mean-square noise
    - Signal-to-noise ratio
- **Operation on interfaced video frames**: Elimination of the interlaced images artifacts by rebuilding or re-aligning fields
- **Feature points detectors**
  - Harris corner detector
  - Canny edge detector

2. Flexible Masks in EasyImage

EasyImage supports flexible masks as an argument for selected functions.

Automatic Thresholding

- AutoThreshold

Histograms

- Histogram

Note. The function HistogramThreshold has no overload with mask argument

Vector Operations

Projection

- ProjectOnAColumn
- ProjectOnARow

Profile

- ImageToLineSegment
- ImageToPath

Statistics Operations

- Area
- AreaDoubleThreshold
- BinaryMoments
- GravityCenter
- PixelAverage
- PixelCompare
- PixelCount
- PixelMax
- PixelMin
- PixelStat
- PixelStdDev
- PixelVariance
- WeightedMoments

Note. The function PixelCompare has no overload with mask argument for BW1 source images.

Noise Reduction by Integration

- RmsNoise

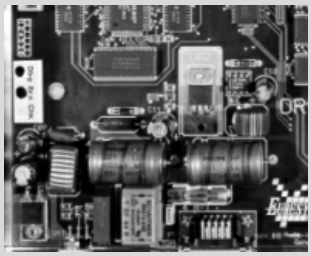
- SignalNoiseRatio

Other Operations – Overlays

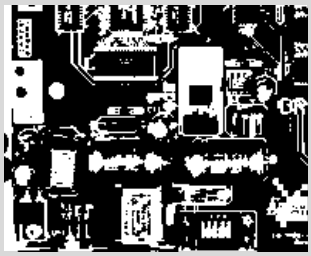
- Overlay

Note. The function Overlay has no overload with mask argument for BW8 source images.

2.1 Using Flexible Masks in EasyImage

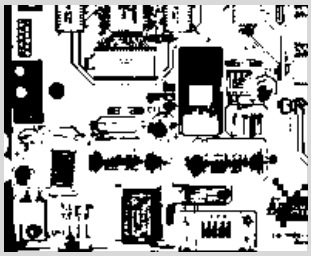


Source image



Mask variable

- **Call the functions from EasyImage that take an input mask as an argument**. For instance, one can evaluate the average value of the pixels in the white layer and after in the black layer.

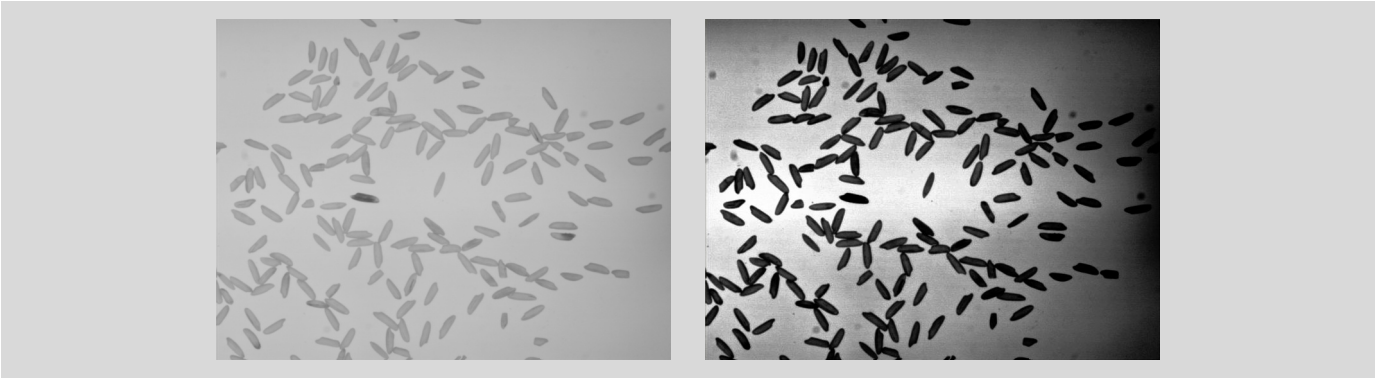


Resulting image

- **Display the results.**

3. Intensity Scale Transformation Functions

The intensity scale transformations apply pixel-wise to the image, i.e. every pixel is handled separately, and its gray-level value is set to another by some "remapping" rule. This allows several forms of histogram stretching, to improve contrast or change the range of gray levels.



Histogram stretching changes contrast

Color transforms are also possible and form a much richer set, because a color value is described by three numbers (the RGB components) instead of one. For this reason, a whole Open eVision library is devoted to color image transforms: EasyColor.

The simplest form of image processing consists in modifying the values of the pixels, one after another, according to a predefined rule. For instance, to adjust the contrast of an image, the gray value at each pixel is simply multiplied by a gain coefficient and offset by a constant value.

One way to automate the contrast adjustment is to normalize the image, i.e. let the average gray level and its standard deviation be equal to predefined values.

For more complex transformations, the use of a "lookup table" is recommended: a correspondence table is filled once with the new pixel values to be substituted to the current ones. This way, the transformation of a whole image can be done very efficiently.

3.1 Gain/Offset

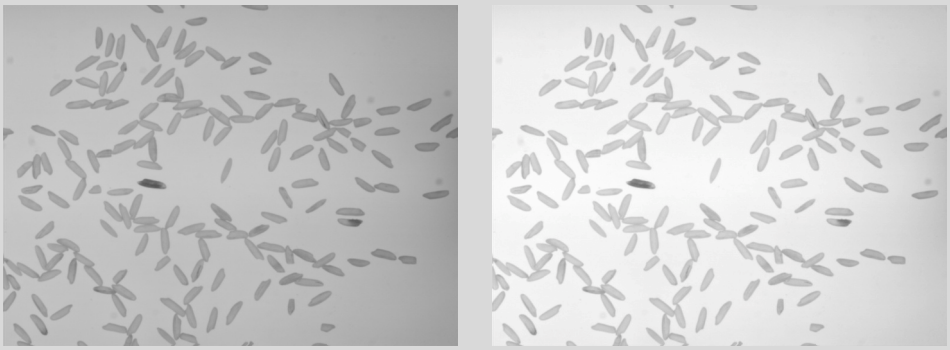
EasyImage::GainOffset is used to adjust the contrast of an image. Each pixel value is simply multiplied by a gain coefficient and offset by a constant value. It amounts to a linear relation of the form:

```
new = Gain * old + Offset
```

Gray Levels

The **gain** allows to adjust the image **contrast**. It should remain close to **1**.

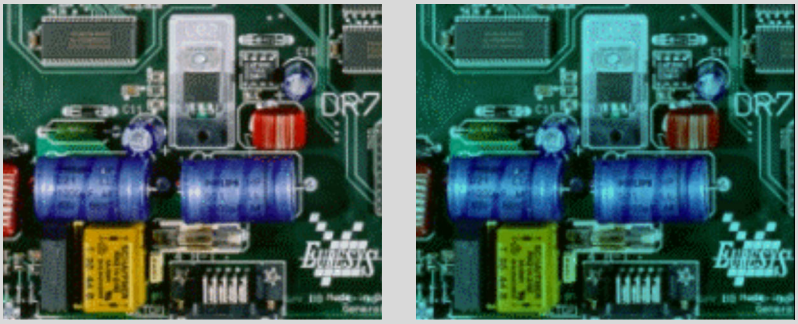
The **offset** allows to adjust the pixels **intensity** (brightness). It can be positive or negative. The resulting values are always saturated to range **[0..255]**.



Source and result images (with gain = 1.2 and offset = +12)
The resulting image is better contrasted and a bit lighter than the source image

Color

When color images are considered, three separate gain and offset values are given, one per color component (red, green, blue).



Example of gain/offset applied on a color image

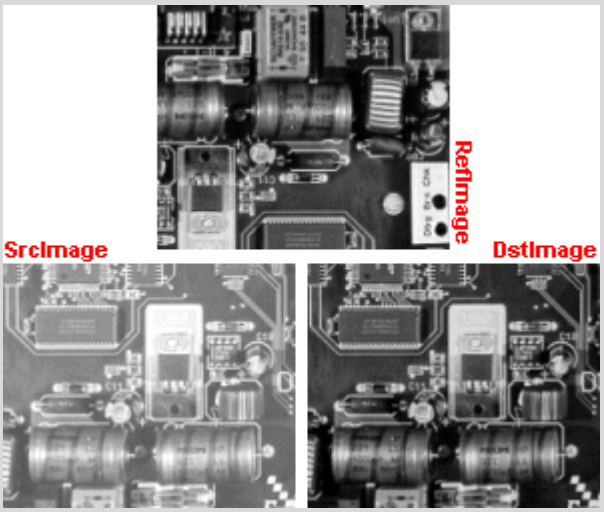
3.2 Normalization

Gray-level normalization is a kind of gain/offset transformation.

It is a convenient way to render images of the same scene comparable, even if the illumination conditions have changed.

From an average and a standard deviation given as parameters, EasyImage::Normalize computes the right gain and offset coefficients and transforms the source image, so that the average of pixel values (brightness) and standard deviation of pixel values (contrast) in the resulting image are the same as the given ones.

Note. This assumes that the linear relation holds, i.e. the camera response is reasonably linear and the image does not saturate.



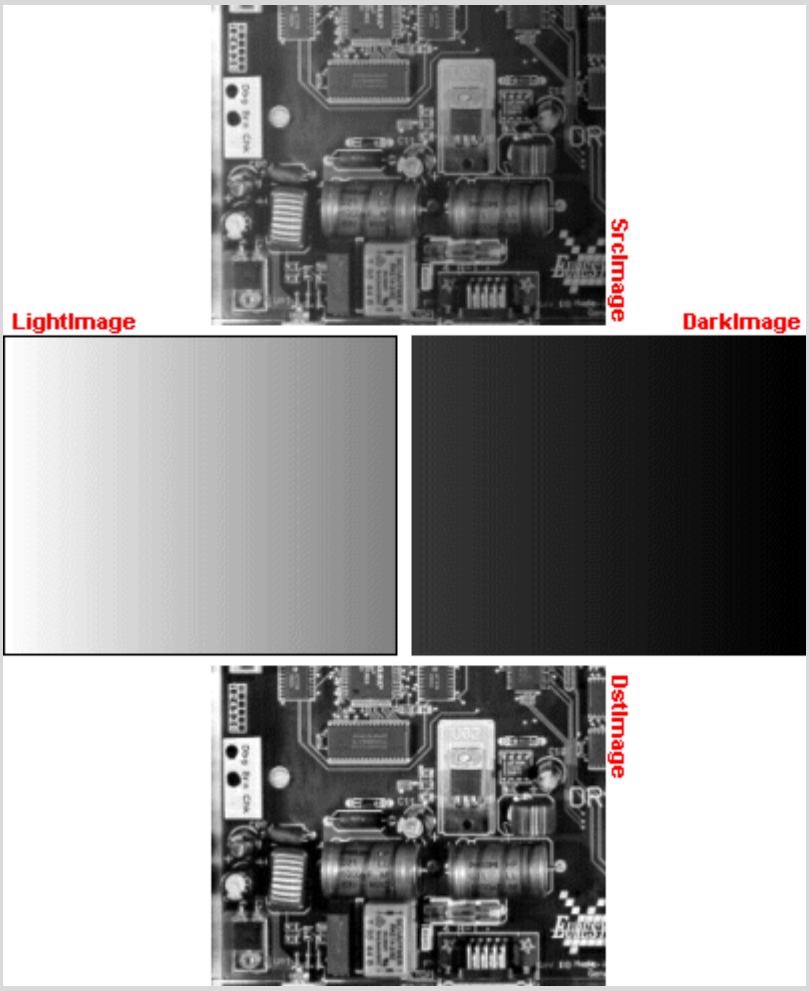
Reference image (from which the average and standard deviation are computed), source image (too bright), and normalized image (contrast and brightness are the same as the reference image)

3.3 Uniformization

Gray-level uniformization (EasyImage::Uniformize) is a linear method to compensate for non-uniform illumination and/or camera sensitivity. It is based on the use of one or two reference images.

- When a single reference image is used, the transform is analog to an adaptive (space-variant) gain; the transform lets the reference image become a constant.
- When two reference images are used, the transform is analog to adaptive gain and offset; the transform let both reference images become a constant.

Note. The reference image(s) should be chosen such that they contain no saturated pixel values (remain in the linear domain) and little (filtered out) noise.

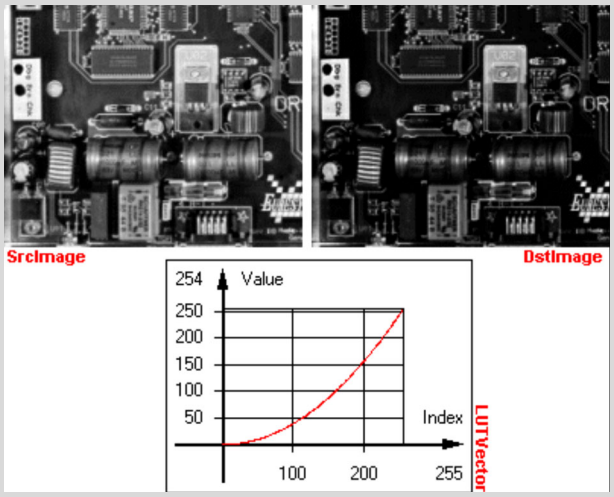


Example of an image uniformized with two reference images

3.4 Lookup Mapping

For complex transformations, a correspondence lookup table (LUT) is filled once with the new pixel values to be substituted to the current ones. This way, the transformation of a whole image can be done very efficiently.

The function EasyImage::Lut provides the capability to perform user-defined LUT-based transformation on BW8 and BW16 images.



Example of a transform function

Note. If the transform function never changes, it is better to construct a lookup table once for all, and use the first method.

4. Thresholding

Thresholding is an image transformation that creates of binary image, a bi-level gray-scale, or a tri-level grayscale image from a source image using a classification method based on the pixel value.

When it operates on BW8 and BW16 images, the EasyImage::Threshold function implements a single-threshold thresholding function that classifies the pixel of the source images into two classes. The destination image can be either a binary image or a bi-level grayscale image of the same type as the source image. Both automatic and manual threshold modes are available.

When it operates on C24 images, the EasyImage::Threshold function implements a range thresholding function that classifies the pixel of the source images into two classes according to their color value. The destination image is a bi-level grayscale BW8 image. There are no automatic threshold modes in that case.

The EasyImage::DoubleThreshold function implements a double-threshold thresholding function that classifies the pixel of the source images into three classes. It operates on BW8 and BW16 images; the destination image is a tri-level grayscale image of the same type as the source image. There are no automatic threshold modes in that case.

4.1 Automatic Thresholding

When the threshold argument of EasyImage::Threshold function is one of the below mentioned value, the function calculates automatically the appropriate threshold value according to the selected mode prior to perform the thresholding operation. There is no need to supply a threshold value!

Min Residue

When threshold = EThresholdMode_MinResidue, the calculated threshold value is such that the quadratic difference between the source and thresholded image is minimized.

This is the default threshold mode. It is applied when invoking EasyImage::Threshold function without the threshold argument.

Max Entropy

When threshold = EThresholdMode_MaxEntropy, the calculated threshold value is such that the entropy (i.e. the amount of information) of the resulting thresholded image is maximized.

Isodata

When threshold = EThresholdMode_Isodata the calculated threshold value is such that if one computes the average gray level of pixels below the threshold and the average gray level of pixels above the threshold, the threshold lies exactly halfway between them.

4.2 Manual Threshold Modes

With manual Threshold modes, the user must supply a threshold value to the thresholding function. Following thresholding methods are available in EasyImage.

Single Threshold Relative

When the threshold argument of the EasyImage::Threshold function is set to EThresholdMode_Relative, the thresholding function computes a threshold value that is such that the fraction of the image pixels that lies below the threshold is equal to the value of the relativeThreshold argument.

Single Threshold Absolute

When the threshold argument of EasyImage::Threshold function is set to a value in the range of pixel values of the source image; the thresholding function uses that value as the threshold value.

Double Threshold

The EasyImage::DoubleThreshold function requires a LowTheshold and a HighTheshold arguments.

4.3 Histogram-Based Threshold

When a histogram of the source image is available, the histogram-based threshold method can be used to speed up the thresholding operation. Therefore, it is required to compute a threshold value prior to invoke the thresholding function.

Following methods are available to compute the threshold value from an existing histogram data using any of the three automatic threshold modes:

- **EasyImage::HistogramThreshold**
- **EasyImage::HistogramThresholdBW16**

These functions return also the average gray levels computed separately on the pixels below and above the threshold.

4.4 Calculating a Threshold Value

When a histogram of the source image is not available, the EasyImage::AutoThreshold function can be used to calculate a threshold value using a specified threshold mode for BW8 and BW16 images. Following threshold modes are available:

- **EThresholdMode_Relative**
- **EThresholdMode_MinResidue**
- **EThresholdMode_MaxEntropy**
- **EThresholdMode_Isodata**

The EasyImage::AutoThreshold function supports flexible masks.

4.5 Key to Success of a Thresholding Operation

- The object and background areas should be of uniform color and be illuminated uniformly. Uniformization of the image may be required prior to the thresholding operation.
- The ranges of gray levels covered by the object and the background must be sufficiently separated. In other words, the lightest background pixel should be darker than the darkest object pixel.

Even when these conditions are met, one should consider if the threshold value should be kept constant or should be adapted to the ambient light intensity. In the first case, **absolute threshold** is sufficient. Otherwise, **relative threshold** or **automatic threshold** is preferred.

5. Arithmetic and Logic

The usual arithmetic operations can be applied to images, rather than to numbers.

- Arithmetic operations:
  - Add, Subtract
  - Multiply, divide
  - Copy
  - Invert, module, shift
- Logical and bitwise operations: AND, OR, XOR, NOT
- Minimum, maximum

The pixels of two source images are combined one by one to generate the pixels of the resulting image. Pixel-wise arithmetical and logical combinations can also be done between an image and a constant.

One application is the comparison between images: subtracting the pixels of two images taken in similar conditions dramatically enhances the differences and can be used as a simple conformity checking.

Another application is to compensate a non-uniform lighting by dividing the target image by the image of the background alone.

Other operations involve logical combinations of pixels to handle masks: you can remove unwanted areas of an image by preparing an appropriate mask image, and clearing all the pixels that belong to the mask.

All arithmetic operations are handled by a single function accepting several overloads: EasyImage::Oper.

These methods and function take one or two source arguments and one destination argument. The source arguments can be images or integer constants. When the source operands are a color image and a gray-level image, each color component is combined with the gray-level component. The result is a color image.

The following enumeration list the supported arithmetic or logic pixel-wise operators: EArithmeticLogicOperation.

- General
  - Compare (abs. value of the difference)
  - Saturated sum
  - Saturated difference
  - Saturated product
  - Saturated quotient
  - Modulo
  - Overflow-free sum
  - Overflow-free difference
  - Overflow-free product
  - Overflow-free quotient
  - Bitwise AND
  - Bitwise OR
  - Bitwise XOR
  - Minimum
  - Maximum

- Equal
- Not equal
- Greater or equal
- Lesser or equal
- Greater
- Lesser

- Copy
  - Sheer Copy
- Invert
  - Invert (negative)
- Shift
  - Left Shift
  - Right Shift
- Logical
  - Logical AND
  - Logical OR
  - Logical XOR
- Overlay
  - Add an overlay
- Set

Operators Copy if mask = 0 and Copy if mask <> 0 are very handy to perform masking: the first image argument serves as a mask that allows or disallows changing the pixel values in the destination image

- Copy if mask = 0
- Copy if mask <> 0

The following combinations are allowed:

	GENERAL	COPY	INVERT	SHIFT	LOGICAL	OVERLAY	SET
Const BW8 -> Image BW8		x					
Const C24 -> Image C24		x					
Image BW8 -> Image BW8		x	x				
Image BW8 -> Image C24		x	x			x	
Image C24 -> Image C24		x	x				

Const BW8, Image BW8 -> Image BW8	x						
Image BW8, Const BW8 -> Image BW8	x			x			x
Image BW8, Image BW8 -> Image BW8	x				x		x
Image BW8, Image BW8 -> Image C24	x					x	
Const C24, Image C24 -> Image C24	x						
Image C24, Const C24 -> Image C24	x			x			
Image C24, Image C24 -> Image C24	x				x		
Image C24, Image BW8 -> Image C24	x				x	x	
Image BW8, Image C24 -> Image C24	x				x		x

Notes. For logical operators, a pixel with value 0 is assumed **FALSE**, otherwise **TRUE**. The result of a logical operation is 0 when **FALSE** and 255 otherwise.

6. Non-Linear Filtering

6.1 Morphological Operators

In certain cases, e.i. with impulse noise, linear filtering behaves poorly. In such cases, it may be advantageous to recourse to non-linear methods such as so-called morphological filters. These combine the pixels forming a given pattern (the "structuring element"), usually a three by three square, using an "all or nothing" transform. This allows among other to thicken or thin the image details, or glue together objects that were previously disjoint.

In this respect, the "top-hat" filter is interesting: it has the property to retain all the tiny image details while removing everything else. The "median" filter efficiently removes impulsive noise.

The morphological operators combine the pixel values in a neighborhood of given shape (square, rectangular or circular) and replace the central pixel of the neighborhood by the result. The combining function is non-linear, and in most cases is a rank filter: consider the N values in the given neighborhood, sort them increasingly and select the K-th largest. Three special cases are most often used: K can be 1 (minimum of the set), N (maximum) or N/2 (median). They correspond to the so-called erosion, dilation and median filter operations.

Erosion, Dilation, Opening, Closing, Top-Hat and Morphological Gradient filters operations documented in this section all use rectangular or circular kernels of odd size. It is important to well understand how the kernel size is specified.

Kernel Size Specification

Kernel size specification

Some examples

HalfWidth/HalfHeight	**Actual** width/height
0	1
1	3
2	5
3	7

...	...

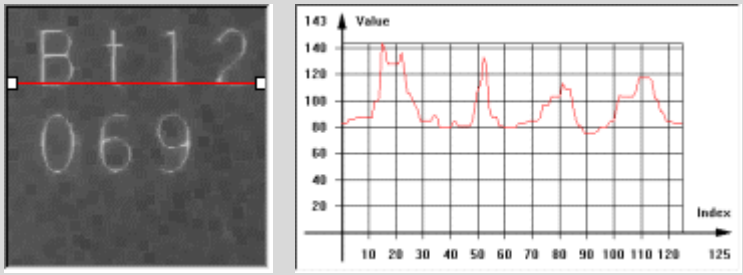
Most of the morphological operations (except top-hat and median filters) can be applied destructively, i.e. the destination image being the same as the source, or not. The destructive operations are faster.

Erosion, Dilation

Erosion thins the white objects by removing one layer of pixels along the object edges. When the kernel size gets large, the tiny white objects completely disappear and the black ones get fatter.

To perform a dilatation, EasyImage provides the following functions:

- **EasyImage::ErodeBox**, **EasyImage::ErodeDisk**
- On bi-level images:
  - **EasyImage::BiLevelErodeBox**
  - **EasyImage::BiLevelErodeDisk**

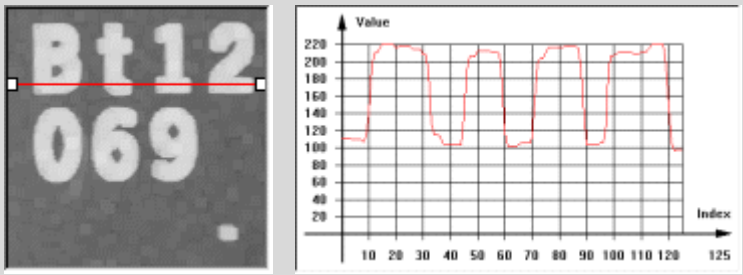


Erosion

Dilation is the dual of erosion. Dilation thickens the white objects by adding one layer of pixels along the object edges. When the kernel size gets large, the white objects get fatter and the tiny black ones disappear.

To operate a dilation EasyImage provides the following functions:

- **EasyImage::DilateBox**, **EasyImage::DilateDisk**
- On bi-level images:
  - **EasyImage::BiLevelDilateBox**
  - **EasyImage::BiLevelDilateDisk**



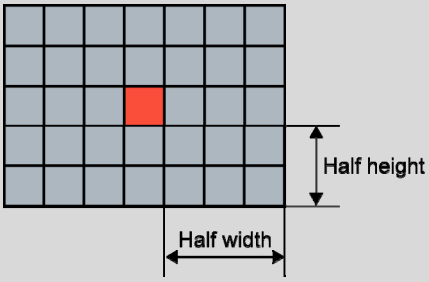
Dilation

Opening, Closing

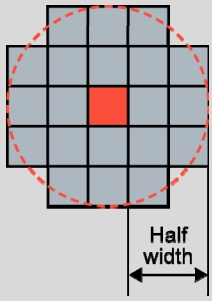
Opening is an erosion followed by a dilation. The global effect is to preserve the overall shape of objects, while dropping the white details smaller than the kernel size. This provides an efficient way to remove white dust or other tiny objects.

To perform opening operations, EasyImage provides the following functions:

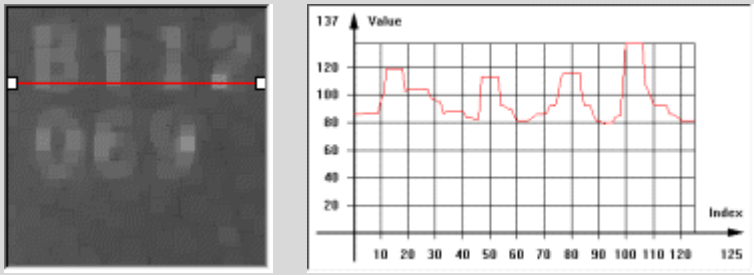
- **EasyImage::OpenBox**, **EasyImage::OpenDisk**
- On bi-level images:
  - **EasyImage::BiLevelOpenBox**
  - **EasyImage::BiLevelOpenDisk**



Rectangular kernel of half width = 3 and half height = 2



Circular kernel of half width = 2

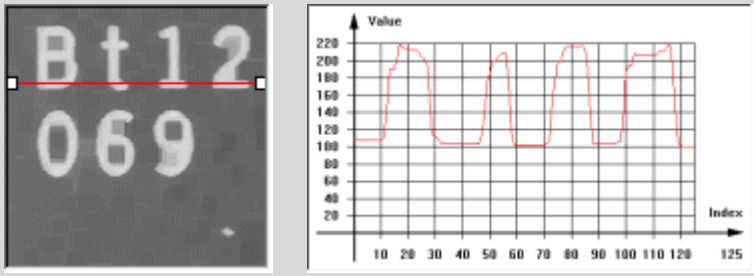


Opening

Closing is the dual of opening. It is a dilation followed by an erosion. The global effect is to preserve the overall shape of objects, while dropping the black details smaller than the kernel size. This provides an efficient way to remove black dust or other tiny holes.

To perform closing operations, EasyImage provides the following functions:

- **EasyImage::CloseBox**, **EasyImage::CloseDisk**
- On bi-level images:
  - **EasyImage::BiLevelCloseBox**
  - **EasyImage::BiLevelCloseDisk**



Closing

Thinning, Thickening

EasyImage::BiLevelThick or EasyImage::Thick applies a thickening operation on a bi-level image or on another image, using a 3x3 kernel.

The thickening kernel coefficients must be 0 (matching black pixel, value 0), 1 (matching non black pixel, value > 0) or -1 (don't care). When a match is found between the kernel coefficients and the neighborhood of a pixel, the pixel value is set to 255.

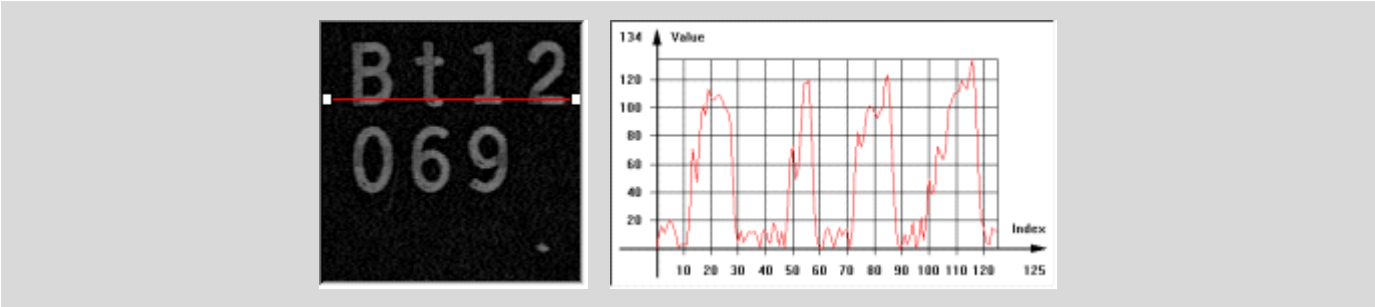
EasyImage::BiLevelThin or EasyImage::Thin applies a thinning operation on a bi-level image or on another image, using a 3x3 kernel.

The thinning kernel coefficients must be 0 (matching black pixel, value 0), 1 (matching non black pixel, value > 0) or -1 (don't care). When a match is found between the kernel coefficients and the neighborhood of a pixel, the pixel value is set to 0.

Top-Hat Filters

The **top-hat filters** (black or white) simply take the difference between an image and its opening (or closure). Thus, they keep the features that an opening or closing operation would erase. The resulting effect is a perfectly flat background where only black or white features smaller than the kernel size appear. The top-hat filter is excellent for improving non-uniform illumination.

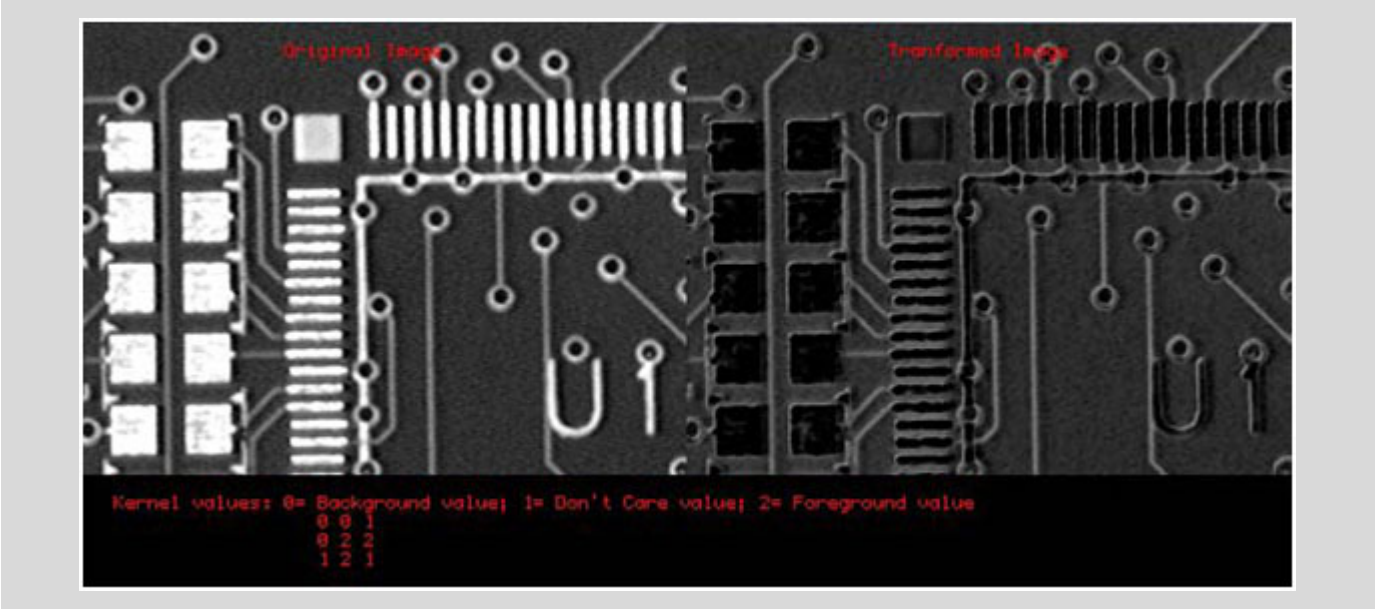
- **White top-hat filter** to enhances the thin white features:
  - **EasyImage::WhiteTopHatBox**
  - **EasyImage::WhiteTopHatDisk**
  - On bi-level images
    - **EasyImage::BiLevelBlackTopHatBox**
    - **EasyImage::BiLevelBlackTopHatDisk**
- **Black top-hat filter** to enhances the thin black features:
  - **EasyImage::BlackTopHatBox**
  - **EasyImage::BlackTopHatDisk**
  - On bi-level images
    - **EasyImage::BiLevelBlackTopHatBox**
    - **EasyImage::BiLevelBlackTopHatDisk**



White top-hat

Hit-and-Miss Transform

Hit-and-miss transform operates on BW8, BW16 or C24 images or ROIs to detect a particular pattern of foreground and background pixels in an image.



Hit-and-miss transform

The EasyImage::HitAndMiss function has three arguments:

- A pointer to the source image of type **EROIBW8**, **EROIBW16**, or **EROIC24**
- A pointer to the destination image of type corresponding to the type of the source image
- A kernel of type **EHitAndMissKernel**

The sizes of the source and destination images must be identical.

Two constructors are available for the kernel object:

- **EHitAndMissKernel(int startX, int startY, int endX, int endY)**
- **EHitAndMissKernel(unsigned int halfSizeX, unsigned int halfSizeY)**

The first constructor requires the following arguments:

- **startX**, **startY**: the coordinates of the top leftmost element of the kernel. These values must be less than or equal to zero.
- **endX**, **endY**: the coordinates of the bottom rightmost element of the kernel. These values must be greater than or equal to zero.

The constructed kernel has the following characteristics:

- **kernel width = (endX – startX + 1)**
- **kernel height = (endY – startY + 1)**

There are no explicit restrictions on the size of the kernel.

The second constructor requires the following arguments:

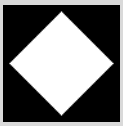
- **halfSizeX**: the half of the kernel width – 1. This value must be greater than zero.
- **halfSizeY**: the half of the kernel height – 1. This value must be greater than zero.

The constructed kernel has the following characteristics:

- **kernel width = ((2 x halfSizeX) + 1)**
- **kernel height = ((2 x halfSizeY) + 1)**
- **kernel StartX = - halfSizeX**
- **kernel StartY = - halfSizeY**

Detecting Corners on a Binary Image

The hit-and-miss transform is a morphological operator that can be used to detect location of corners.



Binary source image

Detecting the Left Corner

1. Define the kernel

The pixel at the left corner is characterized by the fact that it has black pixels as neighbors at its immediate left, top and bottom; whereas it has white pixels as neighbors at its right. This suggests the use of the following hit-and-miss kernel for the detection of the left corner:

```
 -+
-++
 -+
```

2. Apply the filter on the source image

Note that the resulting image should be properly sized.

Resulting image

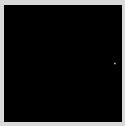
As desired, the only highlighted pixel is located over the left corner of the rhombus.

Locating the three remaining corners

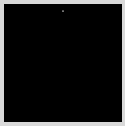
The process is identical for locating the three remaining corners:

1. Declare three kernels that are the rotation of the filter above

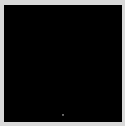
2. Detect the right corner



3. Detect the top corner



4. Detect the bottom corner



Morphological Distance

EasyImage::Distance computes the morphological distance function on a binary image (0 for black, non 0 for white). So, each pixel of the destination image will contain, at the end of the processing, the morphological distance of the corresponding pixel in the source image. The distance function at a given pixel tells how many erosion passes are required to set it to black.

Morphological Gradient

The morphological gradient is the difference between the dilation and the erosion of the image, using the same structuring element.

The kernel size is a pair of odd numbers; they must be halved before they are passed. For instance, a 3x5 kernel is passed as 1x2.

The effect of the morphological gradient is to remove everything in the image but the edges. It is used to perform edge detection.

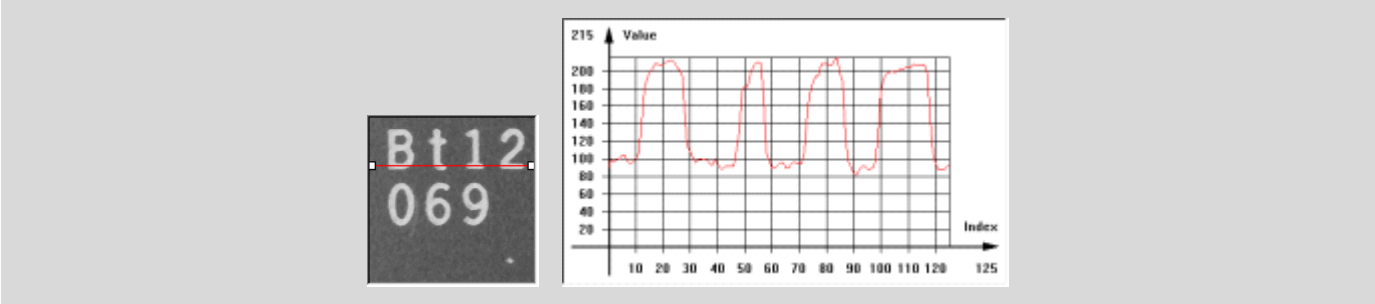
To perform a morphological gradient, EasyImage provides the following functions:

- **EasyImage::MorphoGradientBox**, **EasyImage::MorphoGradientDisk**.
- On bi-level images:
  - **EasyImage::BiLevelMorphoGradientBox**
  - **EasyImage::BiLevelMorphoGradientDisk**

Dilation – Erosion = Gradient

6.2 Median Filter

The median filter (EasyImage::Median or for bi-level images EasyImage::BiLevelMedian) replaces every pixel by the median (central value) of its neighbors in a 3x3 square kernel. This way, outlier pixels are discarded. The net effect is a very effective removal of impulse noise, while edges and image sharpness are well preserved.



Median filter

7. Geometric Transforms

Various causes may render the shape of an image unsatisfactory: it can be too large, too small, or even distorted: if the lenses of the camera exhibit optical aberrations, or if the field of view is not parallel to the focal plane, deformations occur. Sometimes even, the feature of interest has undergone anamorphosis: think of a text wrapped around an annular label, as can be found on CDs.

In such cases, a geometric transformation or correction can be applied. Since the image is made up of a rectangular grid of pixels, the transformed image grid usually is no more rectangular. For this reason, some resampling and possibly interpolation must be done.

Geometric transforms displace the pixels of an image.

Note: The resulting image of a geometric transform is always rectangular. Anyway, the corresponding area in the source image may be a rectangle not aligned on the image axis, a parallelogram or even a more complex shape. Also, some pixels in the destination image may have corresponding pixels outside of the source image. In such cases, the pixels will be left black.

When the source coordinates corresponding to a given destination pixel are not integer, some kind of interpolation technique is required. The nearest neighborhood method is the cheapest in terms of computing time. It simply states that the closest source pixel should be used. The bilinear interpolation method is more accurate. It uses a weighted average of the four neighboring source pixels. Anyway, it is slower.

7.1 Image Registration - Alignment

EasyImage::Register registers an image by realigning one, two or three pivot points to reference positions.

Registration is the process of realigning two misaligned images so that point-to-point comparisons are possible. The simplest way to achieve this is to accurately locate features in both images (landmarks or pivots), using pattern matching, point measurement or whatever other technique, and realign one of the images so that the landmarks are superimposed.

Out-of-image-bounds pixels are black.

When a **single pivot point** is used, the registration transform is a simple translation. If interpolation bits are used, sub-pixel translation is achieved.

When **two pivot points** are used, the registration is a combination of translation, rotation and optionally scaling. If scaling is not allowed, the second pivot point will not be matched exactly in general. Anyway, for most applications scaling should not be used unless it corresponds to a change of lens magnification or viewing distance.

When **three pivot points** are used, the registration is a combination of translation, rotation, shearing correction and optionally scaling. The so-called shear effect can arise when acquiring images with a misaligned line-scan camera.

To achieve good accuracy, the pivot points should be chosen as far apart as possible.

7.2 Horizontal and Vertical Mirroring

EasyImage::HorizontalMirror

EasyImage::VerticalMirror

7.3 Translation, Scaling and Rotation with Optional Interpolation

Translation, scaling and rotation are useful when the position of an object of interest is not constant (because of loose fixture, camera movement, ...), or its apparent size changes because of variable lens magnification. In such cases, it can be necessary to measure the change in position somehow and generate a corrected image.

Scale and Rotate

EasyImage::ScaleRotate: this general linear transform performs:

- Image translation by an arbitrary distance
- Image scaling by an arbitrary scaling factor
- Image rotation by an arbitrary angle.

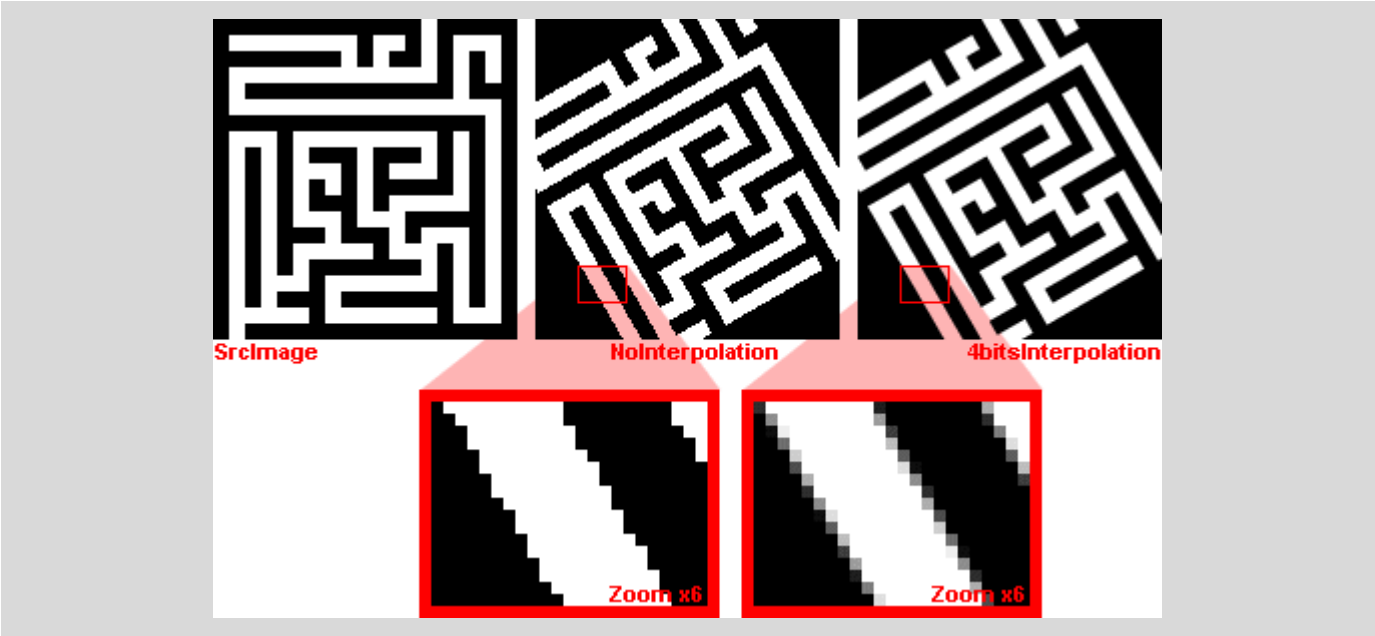
The translation is specified by means of the position coordinates of a pivot-point in the source image and a corresponding pivot point in the destination image.

The scaling is specified by means of scaling factor values for the X- and Y-axis separately.

The rotation is specified by means of a rotation angle value.

For resampling, the nearest neighbor rule or bilinear interpolation with 4 or 8 bits of accuracy is used. The size of the destination image is arbitrary.

Out-of-image-bounds pixels are black.



Scale and rotate example

Shrink

EasyImage::Shrink: resizes an image to a smaller size. Pre-filtering is applied to avoid aliasing.

This function is actually a subset of the EasyImage::ScaleRotate function.

7.4 LUT-Based (Un)Warping

In an image, the feature of interest can have undergone anamorphosis such as a text wrapped around an annular label, as can be found on CDs. EasyImage provides functions allowing to unwarp a circular ring-wedge shape into a straight rectangle. A ring-wedge is delimited by two concentric circles and two straight lines passing through the center.

EasyImage::Warp: transforms an image so that each pixels are moved to the locations specified in the "warp" images used as look-up tables.

EasyImage::SetCircleWarp prepares suitable warp images for use with function EasyImage::Warp.

8. Vector Operations

It is sometimes convenient to extract 1-dimensional data from an image. Think of extracting a profile along a given line segment to detect peaks, drawing iso-level curves to describe the shape of an enclosed object, count the frequencies of every gray value to detect the presence or absence of a given object... These operations generate linear sets of data that are handled as so-called vectors. Subsequent operations on vectors are fast because of the reduced amount of data.

8.1 Projections

The projection is the sum of all gray-level values in a given direction.

- **Projection on a column**: **EasyImage::ProjectOnAColumn** projects an image horizontally onto a column. This function supports **flexible mask**.
- **Projection on a row**: **EasyImage::ProjectOnARow** projects an image vertically onto a row. This function supports **flexible mask**.

Pixel gray/color levels are added when projecting into an EBW32Vector. When projecting into an EBW8Vector/EBW16Vector/EC24Vector, pixel values are averaged, instead.

8.2 Profile

Profile Sampling

A profile is a series of pixel values sampled along a given path in an image. EasyImage::ImageToLineSegment copies the pixel values along a given line segment (arbitrarily oriented) to a vector. The line segment must be wholly contained within the image. The vector length is adjusted automatically. This function supports flexible mask.

A path is described by a series of pixel coordinates stored in a vector. EasyImage::ImageToPath copies the corresponding pixel values to the vector. The function supports flexible mask.

A contour is a closed or not (connected) path, forming the boundary of an object. EasyImage::Contour follows the contour of an object, and stores its constituent pixels inside a profile vector.

Profile Analysis

When a profile has been obtained, it can be processed to find peaks or transitions:

- A **peak** is a maximum or minimum of the signal and may correspond to the crossing of a white or black line or thin feature.
- A transition corresponds to an object edge (black to white or white to black). It can be detected by taking the first **derivative** of the signal and looking for peaks in it.

Profile Derivative

EasyImage::ProfileDerivative computes the first derivative of a profile extracted from a gray-level image. Taking the derivative transforms transitions (edges) into peaks.

Note. Since the **EBW8** data type only handles unsigned values, the derivative is shifted up by 128. Values under [above] 128 correspond to negative [positive] derivative (decreasing [increasing] slope).

Profile Peaks

A peak is the portion of the signal that is above [below] a given threshold. The peak amplitude is defined to be the difference between the threshold value and the maximum [minimum] signal value. The peak area is defined to be the surface comprised between the signal curve and the horizontal line at the given threshold. The result is stored in a peaks vector.

EasyImage::GetProfilePeaks detects peaks in a gray-level profile. Maxima as well as minima are considered. To eliminate false peaks due to noise, two selection criteria are used.

Profile Insertion Into an Image

EasyImage::LineSegmentToImage copies the pixel values from a vector or a constant to the pixels of a given line segment (arbitrarily oriented). The line segment must be wholly contained within the image.

EasyImage::PathToImage copies the pixel values from a vector or a constant to the pixels of a given path.

9. Statistics

A lot of **global counts** and other statistics (numerical parameters related to a whole image) can be computed. These can give summarized information about the image contents, such as the global illumination or contrast, presence or absence of an object, presence of saturation...

The **image moments** are sums involving the pixel coordinates and possibly weighted by pixel values. They provide information such as object position and extent.

Note. These quantities are computed on whole images (possibly thresholded). To compute them on separate objects, resort to EasyObject instead.

EasyImage handles the measurement of:

- **EasyImage::Area** counts the pixels whose values are above (or on) a threshold. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 source images are supported.
- **EasyImage::AreaDoubleThreshold** counts the pixels whose values are comprised between (or on) two thresholds. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 source images are supported.
- **EasyImage::BinaryMoments** computes the zero-th, first or second order moments on the binarized image, i.e. with a unit weight for those pixels with a value above or equal to the threshold, and zero otherwise. This function supports flexible mask. Exhibiting an input mask argument, the function can count 4 overloads. BW8, BW16 source images are supported.
- **EasyImage::WeightedMoments** computes the zero-th, first, second, third or fourth order weighted moments on the gray-level image. The weight of a pixel is its gray-level value. This function supports flexible mask. Exhibiting an input mask argument, the function can count 8 overloads. BW8, BW16 source images are supported.
- **EasyImage::GravityCenter** computes the coordinates of the gravity center of an image, i.e. the average coordinates of the pixels above (or on) the threshold. This function supports flexible mask. Exhibiting an input mask argument, the function can count 2 overloads. BW8, BW16 source images are supported.
- **EasyImage::PixelCount** Counts the pixels in the three value classes separated by two thresholds. This function supports flexible masks and has overloads for BW8 and BW16 source images.
- **EasyImage::PixelStat** computes the minimum, maximum and average gray-level value altogether. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 source images are supported.
- **EasyImage::PixelMax** computes the maximum gray-level value in an image. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 source images are supported.
- **EasyImage::PixelMin** computes the minimum gray-level value in an image. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 source images are supported.
- **EasyImage::PixelAverage** computes the average pixel value in a gray-level or color image. For a color image, this function computes the means of the three pixel color components, the variances of the components and the covariances between pairs of components. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and C24 source images are supported.
- **EasyImage::PixelVariance** computes the average and variance of the pixel values. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and C24 source images are supported.
- **EasyImage::PixelStdDev** computes the average and standard deviation of the pixel values. For a color image, this function computes the standard deviations and correlation coefficients (covariance over the product of the standard deviations) of the pairs of pixel component values. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and C24 source images are supported.

- **EasyImage::PixelCompare** counts the number of pixels differing between two images. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and C24 source images are supported.

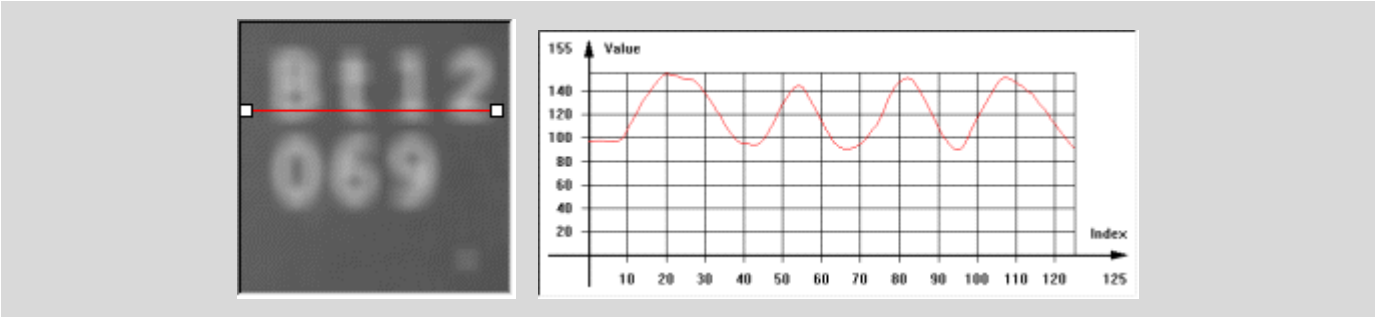
9.1 Sliding Window Statistics

Additionally to the global statistics, the average and standard deviation of the gray-level values can be computed in a sliding window, i.e., computed for each and every position of a rectangular window centered on every pixel. The window size is arbitrary.

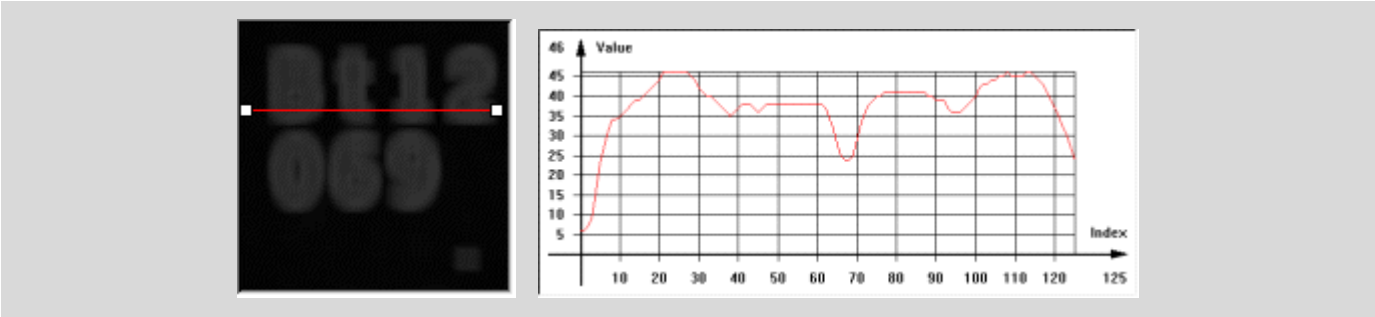
Note. The computing time of these functions does not depend on the window size.

The result of the operation is another image. The local average, EasyImage::LocalAverage, corresponds to a strong low-pass filtering.

The local standard deviation, EasyImage::LocalDeviation, enhances the regions with a high frequency contents, such as noisy or textured areas.



Sliding window average



Sliding window standard deviation

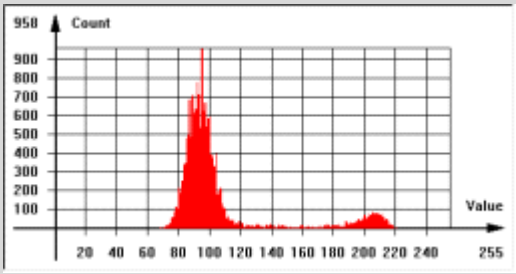
9.2 Histograms

A histogram is a statistical summary of an image: it counts the frequencies (number of occurrences) of every gray-level value in an image.

Histogram Computation

The EasyImage::Histogram function computes the histogram of an image.

Observation of the shape of an image histogram reveals a lot of characteristics. For instance, every maximum in the histogram curve corresponds to a dominant color in the image. In simple cases, the histogram is bi-modal, exhibiting a large peak in the dark values corresponding to the background, and a smaller peak in the light values.



Typical image histogram

EasyImage::CumulateHistogram: calling this function after EasyImage::Histogram allows you to compute the cumulative histogram of an image, i.e. the count of pixels below a given threshold value (instead of the count of pixels with a given gray value, as computed by EasyImage::Histogram).

EasyImage::Histogram supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and BW32 source images are supported.

Histogram Analysis

A lot of image statistics can be efficiently derived from a histogram. This is called **histogram analysis**. Appropriate values for image thresholding can also be derived from a histogram.

The following image parameters can be derived from a histogram thanks to the EasyImage::AnalyseHistogram and EasyImage::AnalyseHistogramBW16 functions:

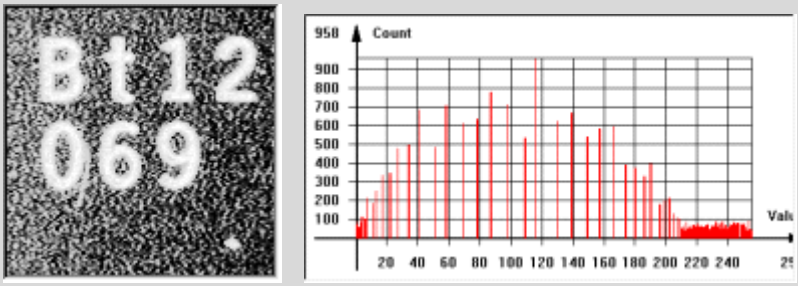
- Total number of pixels.
- Smallest and largest pixel value (gray-level range).

- Average and standard deviation of the pixel values.
- Value and frequency of the most frequent pixel.
- Value and frequency of the least frequent pixel.

Note. Other image statistics can be computed without resorting to histograms.

9.3 Histogram Equalization

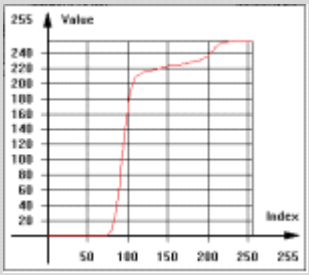
The histogram equalization function, EasyImage::Equalize, re-maps the image gray levels so that the histogram fills in the whole dynamic range as uniformly as possible. This has the effect of maximizing image contrast. It usually reveals a lot of image details in the dark areas.



Equalized image and histogram

LUT Setup

Image equalization works by computing a transformation lookup table from the image histogram. In some cases, it can be more efficient to keep the image histogram for other purposes (for instance, to get images statistics) and/or keep the equalization lookup table (to apply it to other images). In such cases, EasyImage::SetupEqualize allows working explicitly with the histogram and LUT vectors.



Equalization lookup table

9.4 Image Focus

Image Focus Measurement

A sharp focusing can be achieved, if the EasyImage::Focusing quantity is maximum for a given image.

10. Noise Reduction and Estimation

Noise reduction can play an important role in image processing applications. In electronic devices, the occurrence of noise cannot be avoided totally. For this reason, acquired images are always noisy (this is best observed on live images where the pixel values fluctuate around the true intensity). When acquired with 8 bits of accuracy, the noise level typically amounts to about 3 to 5 gray-level values.

When too important, noise has unwanted consequences: the visual quality of images can degrade and certain processing operations (thresholding, high-pass filtering) will enhance noise in a non-acceptable way.

One distinguishes several forms of noise. It can be **additive** (the noise amplitude is not related to the image contents) or **multiplicative** (the noise amplitude is proportional to the local intensity). It can be **uniform** (the noise amplitude follows a smooth distribution centered around zero) or **impulse** (the noise amplitude is infinite). Impulse noise produces a "salt and pepper" effect, while uniform noise blends.

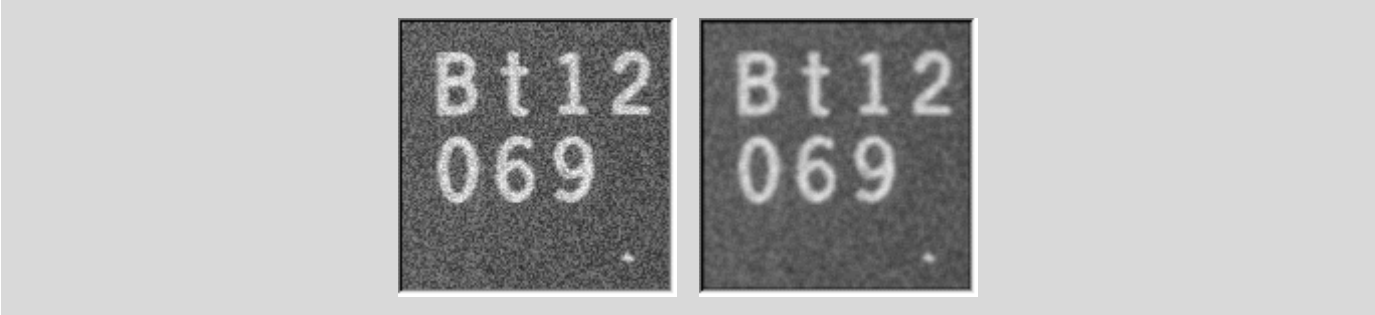
Noise Measurement

EasyImage provides ways to estimate the amount of noise in images. Two or more successive images are required. In the simplest mode, two noisy images are compared. (Other modes are available: if a noise-free image is available, it is compared to a noisy one; a noise-free image can also be built by temporal averaging.)

10.1 Spatial Noise Reduction

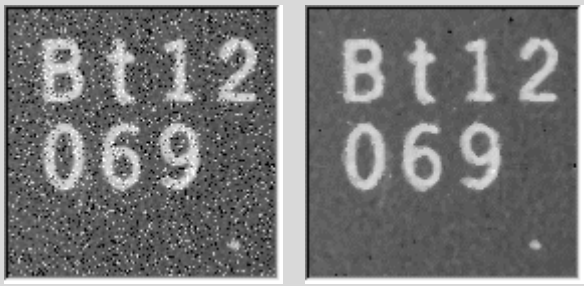
One way to reduce noise is to work on single images and use the correlation between neighboring pixel values. Two approaches are effective:

- **Convolution** replaces the value at each pixel by a combination of its neighbors, leading to some kind of local averaging. Linear filtering is recommended to reduce uniform noise. Beware that it tends to blur edges.
  - **EasyImage::ConvolLowpass1**
  - **EasyImage::ConvolLowpass2**
  - **EasyImage::ConvolLowpass3**
  - **EasyImage::ConvolUniform**



Uniform noise reduction by low-pass filtering

- **Median filtering**, **EasyImage::Median**, replaces the value at each pixel by the value in the pixel neighborhood that is the median (5-th largest value in a 3x3 neighborhood). Median filtering is recommended to reduce impulse noise.



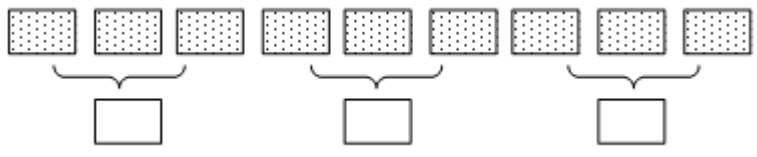
Impulse noise reduction by median filtering

10.2 Temporal Noise Reduction

Spatial noise reduction has limitations due to the fact that it cannot distinguish noise from actual signal changes, so that no technique can lead to perfect noise reduction without spoiling part of the signal. However, if several images of the same (still) scene are available, noise can be separated from signal because it varies from frame to frame while the signal remains unchanged.

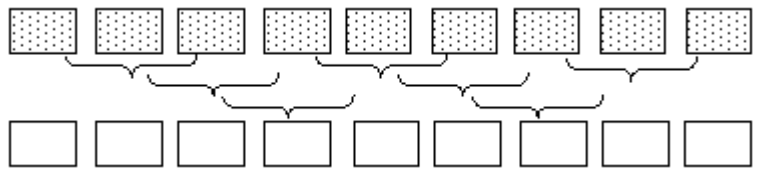
EasyImage provides three ways to minimize noise by means of several images:

- **Temporal average**: just accumulate N images and average them; this is done by means of standard arithmetic operations, as illustrated below.



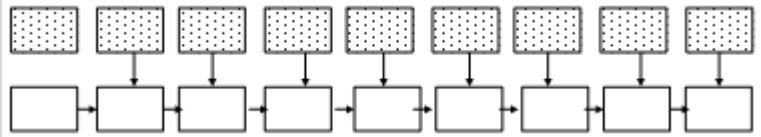
Simple average

- **Temporal moving average**: the simple temporal average process has the disadvantage of producing one de-noised image after N acquisitions only, disallowing fast display refresh. On the opposite, the moving average process uses the last N images and updates the de-noised image each time a new one is acquired. It does this in such a way that the computation time does not depend on N. The whole process is handled by **EMovingAverage**.



Moving average

- **Temporal recursive average**: rather than combining noisy images together, **EasyImage::RecursiveAverage**.combines a noisy image with the previously de-noised image.



Recursive average

Temporal noise reduction is achieved by combining the successive values of individual pixels across time. EasyImage implements two such techniques known as recursive averaging and moving averaging.

Recursive averaging is a well known process for noise reduction by temporal integration. The principle is to continuously update a noise-free image by blending it, using a linear combination, with the raw, noisy, live image stream.

Algorithmically, this amounts to apply the following recurrence:

$$DST_N=a*Src+(1-a)*Dst_{N-1}$$

where a is a mixture coefficient. The value of this coefficient can be adjusted so that a prescribed noise reduction ratio is achieved.

This procedure is effective when applied to still images, but generates a trailing effect on moving objects because of the transient behavior of the filter. The larger the noise reduction ratio, the heavier the trailing effect is.

To work around this, a non-linearity can be introduced in the blending process: small gray-level values variations between successive images are usually caused by noise, while large variations correspond to changes in the signal itself (camera displacement or object movements). Function EasyImage::RecursiveAverage uses this observation and applies stronger noise reduction to small variations and conversely. This way, noise is better reduced in still areas and trailing is avoided in moving areas.

For optimal performance, the non-linearity must be pre-computed once for all using function EasyImage::SetRecursiveAverageLUT.

Note. Before the first call to the **EasyImage::RecursiveAverage** method, the 16-bit work image must be cleared (all pixel values set to zero).

10.3 Noise Estimation

Root-Mean-Square Noise

EasyImage::RmsNoise computes the root-mean-square amplitude of noise, by comparing a given image to a reference image. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and C24 source images are supported.

The reference image can be noiseless (obtained by suppressing the source of noise), or affected by a noise of the same distribution as the given image.

Signal-to-Noise Ratio

EasyImage::SignalNoiseRatio computes the signal to noise ratio, in dB, by comparing a given image to a reference image. This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. BW8, BW16 and C24 source images are supported.

The reference image can be noiseless (obtained by suppressing the source of noise) or be affected by a noise of the same distribution as the given image.

The signal amplitude is defined as the sum of the squared pixel gray-level values while the noise amplitude is defined as the sum of the squared difference between the pixel gray-level values of the given image and the reference.

11. Operation on Interlaced Video Frames

When an image is interlaced, the two frames (even and odd lines) are not recorded at the same time. If there is movement in the scene, a visible artifact can result (the edges of objects exhibit a "comb" effect).

When the movement is uniform and horizontal (objects on a conveyor belt), one cure to this problem is to shift one of the frames horizontally with respect to the other frame using EasyImage::RealignFrame. The amplitude of the shift can be estimated automatically.

EasyImage::GetFrame extracts the frame of given parity from an image while EasyImage::SetFrame replaces the frame of given parity in an image.

The size of the destination image is determined as follows:

```
dstImage_Width = srcImage_Width
dstImage_Height = (srcImage_Height + 1 - odd ) / 2
```

EasyImage::MatchFrames determines the optimal shift amplitude by comparing two successive lines of the image. These lines should be chosen such that they cross some edges or non-uniform areas.

EasyImage::RebuildFrame rebuilds one frame of the image by interpolation between the lines of the other frame.

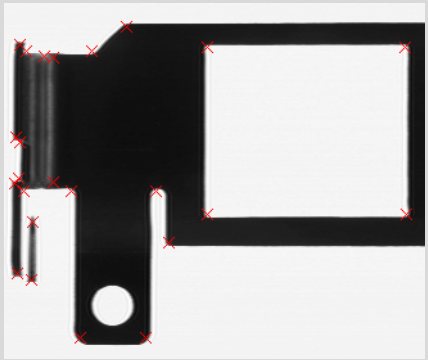
The same image should be used as source and destination. If the destination image differs from the source image, only the shifted rows are copied. To use a different destination image, the source image must be copied first in the destination image object.

EasyImage::SwapFrames: interchanges the even and odd rows of an image. This is helpful when acquisition of an interlaced image has confused even and odd frames.

12. Feature Point Detectors

12.1 Harris Corner Detector

The Harris corner detector is popular due to its strong invariance to rotation, illumination variation and image noise.



Harris corner detector example

It operates on a grayscale BW8 image and delivers a vector of points of interest.

The EasyImage Harris corner detector requires three parameters:

- The integration scale σi, actually the standard deviation of the Gaussian Filter used for scale analysis.
- A "cornerness threshold" expressed as a fraction ranging from 0 to 1 of the maximum1 value of the cornerness of the source image.
- A Boolean that enables the sub-pixel detection or not.

The differentiation scale σd, actually the standard deviation of the Gaussian Filter used for the noise reduction during the computation of the gradient, is derived from the integration scale through the following relation: σd = 0,7 x σi

The following characteristics are available for every point of interest:

- The corner position (pixel coordinates with sub-pixel accuracy if enabled)
- The cornerness measure
- The magnitude of the gradient w.r.t. the differentiation scale σd
- The value of the gradient along the X-axis w.r.t. the differentiation scale σd
- The value of the gradient along the Y-axis w.r.t. the differentiation scale σd

The API of the Harris corner detector is constituted by a single class named EHarrisCornerDetector and the following methods are applicable:

- **Apply**: apply the Harris corner detector on an image/ROI.
- **EHarrisCornerDetector**: constructs a **EHarrisCornerDetector** object initialized to its default values.
- **GetDerivationScale**: returns the current derivation scale.

- **GetScale**: returns the integration scale.
- **GetThreshold**: returns the current threshold.
- **GetThresholdingMode**: returns the current thresholding mode for the cornerness measure.
- **IsGradientNormalizationEnabled**: returns whether the gradient is normalized before the computation of the cornerness measure.
- **IsSubpixelPrecisionEnabled**: returns whether the sub-pixel interpolation is enabled.
- **SetDerivationScale**: sets the derivation scale.
- **SetGradientNormalizationEnabled**: sets whether the gradient is normalized before the computation of the cornerness measure.
- **SetScale**: sets the integration scale.
- **SetSubpixelPrecisionEnabled**: sets whether the sub-pixel interpolation is enabled.
- **SetThreshold**: sets the threshold on the cornerness measure for a pixel to be considered as a corner.
- **SetThresholdingMode**: sets the thresholding mode for the cornerness measure.

Basic usage of Harris Corner Detector

An object of the EHarrisCornerDetector class can be reused across applications of the Harris detector, in order to reduce the setup time.

Create an instance of the detector

Create an instance of the detector and set the appropriate method, for instance, the integration scale, SetScale, with the structures of interest that could have a spatial extent of 2 pixels.

Apply the detector

As a new image arrives, the detector can be applied through the Apply method of the EHarrisCornerDetector object. This method takes two arguments:

- the input image and
- a container object of type **EHarrisInterestPoints**, that will hold the interest points that are found in the input image.

Output of the detector

Once the detector has been applied, the elements of the output vector can be individually accessed.

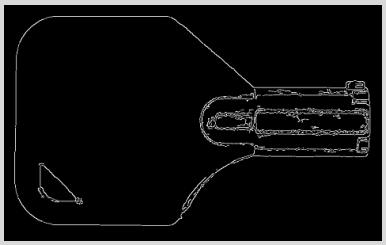
12.2 Canny Edge Detector

The Canny detector is known as the optimal edge detector.

The Canny edge detector offers three excellent characteristics for the image processing applications:

- A good detection: find as many edges in the image as possible
- A good localization: the found edges are as close as possible to the “real” edges in the image
- A minimal response: a single edge response is accepted for each position, i.e. avoiding multiple close or intersecting edge responses

The EasyImage Canny edge detector operates on a grayscale BW8 image and delivers a black-and-white BW8 image where pixels have only 2 possible values: **0** and **255**. Pixels corresponding to edges in the source image are set to value **255**; other pixels are set to value **0**.



Canny edge detector example

The Canny edge detector requires only two parameters:

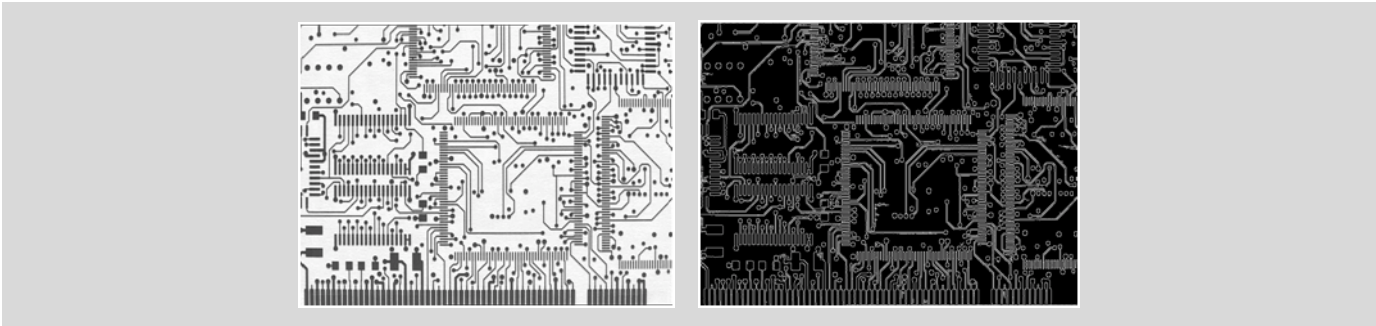
- The **characteristic scale of the features of interest**, actually the standard deviation of the Gaussian filter used to smooth the source image.
- **A gradient threshold with hysteresis** (two values) expressed as a fraction ranging from 0 to 1 of the maximum magnitude of the gradient of the source image.

The API of the Canny edge detector is rather simple; it is constituted by a single class named ECannyEdgeDetector with the following methods:

- **Apply**: apply the Canny edge detector on an image/ROI.
- **ECannyEdgeDetector**: constructs a **ECannyEdgeDetector** object initialized to its default values.
- **GetHighThreshold**: returns the high hysteresis threshold for a pixel to be considered as an edge.
- **GetLowThreshold**: returns the low hysteresis threshold for a pixel to be considered as an edge.
- **GetSmoothingScale**: returns the scale of the features of interest.
- **GetThresholdingMode**: returns the mode of the hysteresis thresholding.
- **ResetSmoothingScale**: prevents the smoothing of the source image by a Gaussian filter.
- **SetHighThreshold**: sets the high hysteresis threshold for a pixel to be considered as an edge.
- **SetLowThreshold**: sets the low hysteresis threshold for a pixel to be considered as an edge.
- **SetSmoothingScale**: sets the scale of the features of interest.
- **SetThresholdingMode**: sets the mode of the hysteresis thresholding.

Note that the EasyImage Canny edge detector provides control to adjust the scale analysis and doesn't allow sub-pixel interpolation. It delivers only binary image after thresholding.

The **result image** must have the same dimensions as the input image.



Source image and the result after a Canny edge detection

13. Overlay

EasyImage::Overlay overlays an image on the top of a color image, at a given position.

If a color image is provided as the source image, all the pixels of this image are copied to the destination image, but the ones that equal the reference color.

If a BW8 image is provided as the source image, all the pixels of the overlay image are copied to the destination image, but the ones that equal the reference color, the latter being replaced by the content of the source image.

This function supports flexible mask. Exhibiting an input mask argument, the function can count 3 overloads. C24, C15 and C16 source images are supported.

Note. When a C24 image is used as overlay source image, the color of the overlay in destination image is the same as the one in the overlay source image, thus allowing multi colored overlays.

14. Scalar Gradient

EasyImage::GradientScalar computes the (scalar) gradient image derived from a given source image.

The scalar value derived from the gradient depends on the preset lookup-table image.

The gradient of a gray-scale image corresponds to a vector, the components of which are the partial derivatives of the gray-level signal in the horizontal and vertical direction. A vector can be characterized by a direction and a length, corresponding to the gradient orientation, here called argument, and the gradient magnitude, here called magnitude.

Function GradientScalar generates a gradient direction or gradient magnitude map (gray-level image) from a given gray-level image. For efficiency, a pre-computed lookup-table is used to define the desired transformation. This lookup-table is stored as a standard EImageBW8/EImageBW16. Use one of EasyImage::ArgumentImage or EasyImage::ModulusImage once before calling GradientScalar.

EasyColor

1. What Is EasyColor?

EasyColor includes a set of optimized color systems transformation functions and color analysis functions. The color systems supported are RGB, XYZ, L*a*b*, L*u*v*, YUV, YIQ, ISH, LSH, VSH, LCH and YSH. EasyColor provides efficient means to convert images between these systems and to transform color images into gray level images and vice versa.

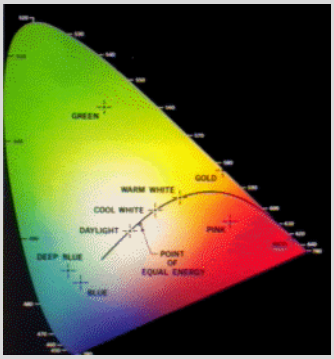
Although the RGB (red, green, blue) representation of color images is well suited for color reproduction (it is used by monitors and cameras), many other representations have been designed for various purposes. More particularly, the “Intensity/Saturation/Hue” color systems are well suited for machine vision applications. EasyColor supports several of them. They separate the achromatic (black and white) component (Intensity) from the chromatic components (Saturation and Hue) which are used to describe colors. This allows a more intuitive interpretation of colors and is very useful to segment colors while eliminating lighting effects. It is thus required, when doing color image processing, to convert the RGB images coming from the camera to another color space, such as LSH, ISH or YSH. EasyColor provides a set of optimized color space conversion functions.

Also included in EasyColor are traditional color image processing functions (such as Bayer pattern conversion and color balance correction), as well as powerful color analysis functions, which allow the user to detect and classify color objects and defects. For example, color image segmentation allows you to decompose a color image in different regions by assigning a class to every pixel. Color image segmentation can be used in conjunction with EasyObject to perform blob analysis on the segmented regions. It is also possible to filter pixels by selecting ranges of values for each component, for example, selecting “olive green” pixels based on their hue only, with a loose discrimination on the intensity and saturation to eliminate surface and lighting effects.

EasyColor deals with color images rather than gray-scale images. The core function categories are:

- Color transformations
  - Lookup tables (LUTs)
  - LUT for specific usages
    - Colorimetric systems conversion
    - LUT for gain/offset (color)
    - LUT for color calibration
    - LUT for color balance: gamma pre-compensation, white balance
- Color image components
  - Merging and extracting image components
  - Pseudo-coloring
- Color classification for segmentation
- Special color formats
  - YUV 422 decompression
  - Bayer patterns to RGB

1.1 What Is Color?



Visible color gamut in the XYZ color space

The human eye is sensitive to the **intensity** of light (dark areas opposed to bright), which is captured by gray-level images. This is known as the *achromatic sensation*.

It is also sensitive, to some extent, to the spectral composition of light, i.e. the superposition of different wavelengths. Lab experiments have shown that any colored sensation can be described as the mixture, in given proportions, of three well chosen primary colors, such as **red**, **green** and **blue**; this is known as the *tri-stimulus theory*.

If the proportions of each color component, or *channel*, are coded on 8 bits, one obtains a **24-bit-per-pixel** representation, often referred to as *true color* image. This format is very often used in image processing because it allows to represent as many colors as the eye can distinguish.

We give names to the different colors we perceive: orange, indigo, purple, olive green... These correspond to different **hues**.

Furthermore, colors can be seen as more or less pure. This property is called the **saturation**: the more saturated a color is, the more vivid it appears; the less saturated it is, the closer it comes to gray. This qualitative description of color is sometimes better suited for processing because it is not sensitive to the choice of the color primaries. Anyway, since it does not correspond to the native digitized color representation, some conversions may be required.

Note. In any case, a color image is 3 times as large as a gray-level image, in terms of storage requirements, and requires at least 3 times more processing.

1.2 Color Systems

Though the RGB representation is well suited for color reproduction (a color monitor screen works by displaying a mixture of red, green and blue light), many other representations have been designed for various purposes.

Three families of systems can be distinguished:

- The **mixture** systems give the proportions of the three primaries to be combined.

- The **luma/chroma** systems separate the achromatic (black and white) sensation and the chromatic sensation (using two independent components). These systems are convenient when a black and white image is required as well (as is the case with television).
- The **intensity/saturation/hue** systems further describe the chromatic sensation in terms of saturation and hue. These allow a more intuitive interpretation of colors and are very useful to eliminate lighting effects.

Another way to class the color systems refers to the technical field involved:

- The **RGB** components are mainly used for display purposes.
- The **XYZ** primaries, that have been designed by the *Commission Internationale de l'Eclairage*, have become a universal standard for device-independent color representation.
- The **YUV** primaries have been introduced by the broadcast industry to allow efficient transmission of color images by compressing the chrominance information.

Since no color system can fulfill all purposes, EasyColor supports a range of them, with additional variants.

Supported color systems

	RGB-based	XYZ-based	YUV-based
Mixture	RGB	XYZ	—
Luma/Chroma	—	L*a*b* L*u*v*	YUV YIQ
Intensity/Saturation/Hue	ISH LSH VSH	LCH	YSH

EasyColor uses the RGB system as the preferred internal representation (under Windows, only RGB images can be displayed directly). The format used is compatible with the 24-bit Windows Bitmap requirements.

Quantized vs. Continuous Values

Most of the time, the color coordinates used in the classical systems are described as **continuous** values, often normalized to the **[0..1]** interval. Computations on such values, termed unquantized, are simpler.

On the other hand, the storage of images in a frame buffer imposes to use a byte representation, corresponding to **discrete** values, in the **[0..255]** interval. Such values are termed quantized.

In principle, all image processing operations apply to the quantized representation; anyway, for convenience, conversion operations between the various color systems can also be specified using unquantized coordinates.

1.3 Color Image Processing

The color images not only contain three times more raw information than the gray-level ones, they also allow richer processing. For instance, all three RGB components reflected by an object have amplitude proportional to the intensity of the light source. By considering the ratio of two color components, one obtains an illumination-independent image. With a clever combination of three pieces of information per pixel, one can extract better features.

There are several ways to process a color image. These can be summarized as follows:

- **Component extraction**: very often, the best way to handle a color image is to extract from the triple color information the most relevant feature, to reduce the amount of data. For instance, different objects in the field of view may be distinguished by their hue alone. So, as a pre-processing step, the color image can be transformed to a gray-level one containing only the hue values. The latter can be processed as usual.
- **De-coupled transformations**: some operations can be done separately on each color component. For instance, adding two images together means that you sum the corresponding red, green and blue component images. The result is stored, component by component, in a resulting color image.
- **Coupled transformations**: in the most complex case, all three color components are combined to produce three derived components. Think of the conversion between the YIQ representation, that might be provided by a color camera, to the RGB representation desired for display or further processing.

Note. It is also worthwhile to mention that a color image is a vector field (three components per pixel), as opposed to a gray-level image, which is a scalar field. Processing vector quantities is harder than processing scalars, and not all operations on scalars can be generalized to vectors.

2. Color Transformations

2.1 Lookup Tables (LUTs)

The EColorLookup class lets the user define its own color transform, and take advantage of the fast lookup processing capabilities. Additionally, the source and destination color systems can be specified, for consistency checking.

Note. The user transform is defined using unquantized values.

Whenever a color lookup table is used, a decision must be made regarding the **number of table entries**, and whether **interpolation** is used.

Each time a color lookup table is filled, all the entries are recomputed. When the IndexBits property of the lookup table equals 6, this may take a very long time. Such large lookup tables should be computed once only. Different combinations of IndexBits and Interpolation provide a trade-off between accuracy and speed for the table pre-computation and table use.

IndexBits	Number of entries	Table size (bytes)
4	$2^{(3*4)}$ = 4,096	14,739
5	$2^{(3*5)}$ = 32,768	107,811
6	$2^{(3*6)}$ = 262,144	823,875

2.2 Computing LUT for Specific Usages

To apply some transform to a color image, you initialize a color lookup once for all, using one of the following functions: EasyImage::GainOffset, ConvertFromRGB, ConvertToRGB, Calibrate, WhiteBalance.

This color lookup is used at will in a transformation operation such as EasyColor::Transform. Several types of transformations are available.

When no interpolation is used, the accuracy of the transformed values roughly corresponds to the number of index bits. When interpolation is used and the transform is smooth enough, the resulting values recover 8 bits of accuracy per component.

Few table entries will correspond to small storage requirements, at the expense of accuracy. No interpolation improves the running time at the expense of accuracy.

When the involved transform is linear (such as YUV to RGB), interpolation will always give exact results, regardless the number of table entries.

Colorimetric Systems Conversion

The native image format is RGB. Whenever another color system is preferred, a conversion must take place. The color lookup table supports any conversion to and from any system and RGB.

When color transforms need to be applied, they can be highly time-consuming. For instance, computing the L*a*b* components (which have the nice property to define a "perceptually uniform" color space) is a complex, non-linear operation. Applying this transform to all pixels of an image is just unacceptable.

To circumvent this difficulty, EasyColor provides a powerful lookup table mechanism: a lookup table is an array of values that tells what output corresponds to a given input. If the lookup table can be pre-computed, applying the transform becomes feasible.

Given that a color pixel can take 16,777,216 (2^{24}) different values, a full color lookup table should include as many entries, and would occupy 50 MB of memory.

The lookup table device can be used to convert an image from one color system to another. It can also be programmed to apply any user-defined transformation.

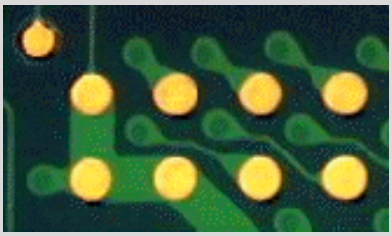
Furthermore, some operations use the lookup table on the fly and avoid the need to store the transformed image. For instance, it is possible to alter the U (of YUV) component of an image while the image remains stored in RGB format.

See also Special Color Formats.

LUT for Gain/Offset (Color)

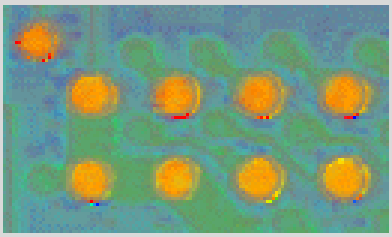
The contrast enhancement transform lets apply separate gains and offsets to each of the three components of an image. The image is assumed to be in the RGB representation, but the gains and offsets can be applied in any color space representation. The actual process is the sequence of transforming from RGB to the targeted color space, applying the gains and offsets and transforming back to RGB.

When applied to a mixture representation, all three gains and offset should vary in a similar way. When applied to luma/chroma representations, the gain and offset for the chromatic components should vary in a similar way. When applied to intensity/saturation/hue representation, it makes little sense to apply gain and offset to the hue component.



Enhanced saturation

Note. The contrast enhancement function can be used to uniformize a given component: setting the gain to 0 for some component will amount to setting all pixels to the value of the offset for this component.



Uniform lightness

LUT for Color Calibration

For accurate color measurement and faithful color rendition, it may be important to recalibrate the colors to correct color distortions introduced by the image acquisition chain. For this to be possible, one has to compare sample colors from the image and compare them to their true values. A calibrated color chart, such as the IT8, is required.

To sample colors accurately, it is best to take the average color in a suitable ROI. You can do so by using PixelAverage. The true color values are specified in a universal (device independent) color system, namely XYZ.

Note. Even though the reference colors are described by their XYZ coordinates, the image to be calibrated must contain RGB information.

The calibration transform can be based on a single reference color, three reference colors or four of them. In the first case, calibration amounts to gain adjustment for the three color components. In the second and third case, a linear or affine transform is used.

LUT for Color Balance

The color adjustment functions allow improving the contents of a color image with respect to two unwanted effects: gamma correction and white imbalance (see below).

For efficiency, such corrections are applied by means of a color lookup table prepared once for all. (It implements a de-coupled color transformation). The way to use it is straightforward: setup the lookup table by means of WhiteBalance, and use it at will on a series of images by means of Transform.

Gamma Pre-Compensation

Many color cameras use to apply a gamma pre-compensation process that deals with the non-linear response of the display device (such as a TV monitor). (This non-linear response corresponds to a potential law of exponent gamma in a Cartesian scale, i.e. a linear law of slope gamma in a bi-logarithmic scale.)

The pre-compensation process applies the inverse transform to the signal, so that the image renders correctly on the display. Three pre-defined gamma values are available, depending on the video standard at hand:

Video standard	Gamma value	EasyColor property
NTSC	1/2.2	CompensateNtscGamma
PAL	1/2.8	CompensatePalGamma
SMPTE	0.45	CompensateSmpteGamma

Gamma pre-compensation can be applied by means of **Transform**, with lookup tables set up by **WhiteBalance**.

Note. Ideally, if gamma pre-compensation is used for display purposes, it should be used after processing, if any. Doing it before processing would change the result because of the non-linearity so introduced.

Gamma Pre-Compensation Cancellation

Many existing color cameras have a built-in gamma pre-compensation feature. Most of the time, it can be turned off.

If this feature is not desired and cannot be turned off by hardware, its effect can be cancelled by applying the direct gamma transform instead. The following pre-defined gamma values are available for this purpose:

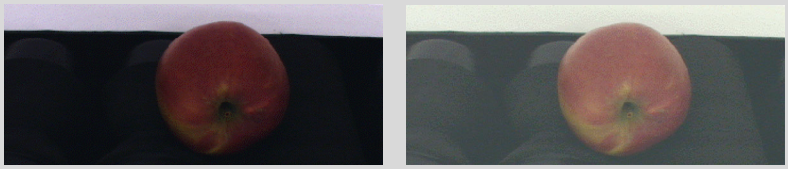
Video standard	Gamma value	EasyColor property
NTSC	2.2	NtscGamma
PAL	2.8	PalGamma
SMPTE	1/0.45	SmpteGamma

Gamma pre-compensation can be applied by means of **Transform** with lookup tables set up by **WhiteBalance**.

Note. Pre-compensation cancellation and pure pre-compensation correspond to exponents that are inverse of each other.

White Balance

A camera may exhibit color imbalance, i.e. the three color channels having mismatched gains, or the illuminant (the light sources) not being perfectly white. When this occurs, the white areas do not look white but instead appear as some strongly unsaturated color. The white balance correction automatically adjusts three independent gains so that the components of a white pixel become equal.



Raw image, and image with white balance and gamma pre-compensation

This means that a white balance calibration step is required, during which a white surface must be shown to the camera and the corresponding color component are measured. PixelAverage can be used for this purpose.

3. Color Image Components

A color image can be seen as a set of three color planes, each corresponding to a color component. The color planes are themselves continuous tone images. A color image contains three black and white images. Inversely, three separate gray-level images can be combined to form a single color image.

3.1 Merging and Extracting Image Components

EasyColor allows changing one color plane at a time, or the three together. It also allows extracting one color plane or the three color planes at the same time. See Compose, Decompose, GetComponent, SetComponent.

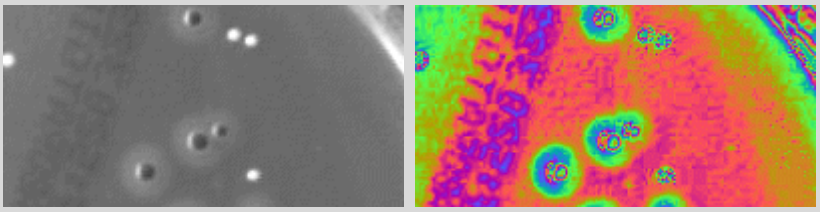
When performing these operations, a color lookup transform can be used on the fly, allowing for instance to build an RGB image out of lightness, saturation and hue planes.

Note. In the Windows bitmap format, the color planes of an image are interleaved (blue, green and red pixels follow each other) rather than planar. The functions described in this section perform the necessary interleaving/de-interleaving operations.

3.2 Pseudo-Coloring

Pseudo-coloring transforms gray-level to colors. The pseudo-colors are normally used to enhance the display of gray-level images. The trick is to define a regular gamut of 256 colors. Each of these colors will be assigned to the pixels with corresponding gray-level value. To define the shades of pseudo-colors, EasyColor allows specifying a trajectory in the color space of an arbitrary system.

Note. Pseudo-coloring can be done at display time only using a color palette. This working mode is supported by the drawing functions of the image. (See Image and Vector Drawing)The pseudo-color mechanism on the other hand actually generates a color image obtained by transforming the input image. The color image can be saved and/or transformed as any other color image.

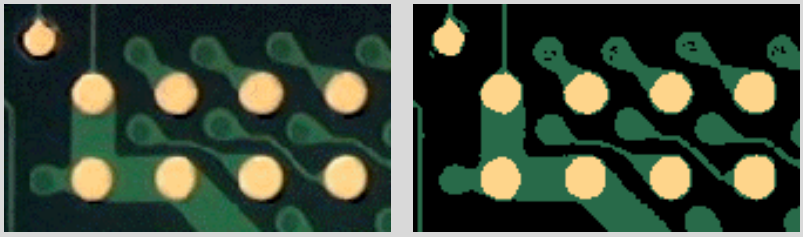


Gray-level and pseudo-colored image

4. Color Classification for Segmentation

EasyColor implements a classification scheme that can be combined with blob analysis (EasyObject) allowing to perform effective color segmentation.

In the presence of a color image, a powerful way to separate the different objects in the scene is to consider a set of distinct colors and to associate each and every pixel with the closest of these colors. This EasyColor process associates each pixel with a layer index that can be used thereafter in EasyObject with the labeled image segmenter.



Raw image and segmented image (3 colors)

5. Special Color Formats

5.1 YUV 422 Decompression

In the YUV system, it has been established that sub-sampling the chroma components does not degrade the visual image quality. The 4:4:4 format uses three bytes of information by pixel. The 4:2:2 format is such that only the U and V chroma of the even pixels are kept and they are stored with the even and odd luma, as follows:

$$Y_{[even]} \quad U_{[even]} \quad Y_{[odd]} \quad V_{[even]}$$

Thus, only two bytes per pixel are required.

Two functions are provided to convert between the 4:2:2 and 4:4:4 representations: Format422To444 and Format444To422.

Note. These functions only deal with data encoding. They do not convert between the RGB and YUV systems. This can be achieved by means of color lookup tables.

5.2 Bayer Patterns to RGB

The Bayer pattern is a special color image encoding format that allows to capture color information from a single sensor. A color filter with a specific layout is placed in front of the sensor so that part of the pixels receive red light only, while others receive green or blue only. An image encoded after the Bayer pattern has the same format as a gray-level image and conveys three times less information. The true horizontal and vertical resolutions are smaller than those of a true color image.

G	B	G	B	G	B
R	G	R	G	R	G
G	B	G	B	G	B
R	G	R	G	R	G
G	B	G	B	G	B
R	G	R	G	R	G

RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB

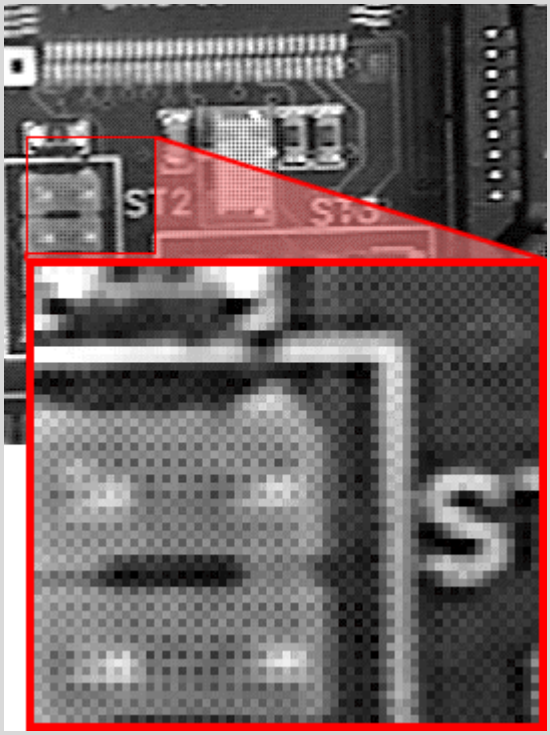
Bayer vs. true color format

Note. The Bayer pattern is assumed to start with GB/RG block in the upper left corner, i.e. with a green pixel at every even-column/even-row locations. If the image is cropped by the camera, this parity rule can be lost. The Bayer functions allow specifying the correct parity of the source image. Note however that parity adjustment is unnecessary when working on a Open eVision ROI.

The Bayer conversion method EasyColor::BayerToC24 transforms an image captured using the Bayer pattern, and stored as a standard gray-level image in a standard true color image. Three working modes are available, depending on the way the missing pixels are reconstructed:

- **In the non-interpolated mode**, the color components are reconstructed by duplicating the nearest pixel to above and/or to the left of the current pixel.
- **In the standard interpolated mode**, an appropriate average of relevant neighboring pixels is performed.
- **In the improved interpolated mode**, an advanced method is used to interpolate the unknown component values. This mode reduces visible artifacts along object edges.

The more complex the interpolation, the slower the conversion will be. However, it is highly recommended to use interpolation.



Bayer pattern encoded image

Converted images with no (top), standard (left) and improved interpolation method (right)

Bayer processing

Even though a Bayer encoded image is not compatible with a true color one (EC24), white balance and gamma correction can be applied to it as such, using the normal EColorLookup class/control (see Color Balance). The benefit of this approach is speed, since a Bayer image is three times smaller.

EasyObject

1. What Is EasyObject?

The EasyObject library handles image segmentation, i.e. the decomposition of images into separate coded elements, also called blobs. The coded elements can be objects or holes. EasyObject supports BW1, BW8, BW16 and C24 source images.

Once the coded elements have been constructed, they can be handled as independent entities. Various geometric parameters or features, such as area, width, or ellipse of inertia, can be computed on them. Then the objects or holes of interest can be selected by means of the layer they belong to, of their position, of a rectangular ROI or of their computed features. The selection criteria can be combined incrementally (select the small objects; among these, select those close to the right edge...).

When the desired objects have been selected, they can be listed and inquired about their geometric characteristics. They can also be sorted using the values of the geometric features.

The rule that determines what constitutes an object is simply the grouping of neighboring pixels of the same gray level range. In image processing parlance, it uses thresholding followed by connected components labeling.

In addition, EasyObject also allows detecting holes in objects. Indeed, the object holes could also be the real objects of interest.

The hole construction process is always subsequent to the real object construction process. It is sometimes easier to localize first an object of interest, then to detect its hole(s) (with EasyObject) and finally to measure some of their characteristics (with EasyGauge or EasyObject).

The hole building process is optional, ensuring that users not needing this feature will not suffer from an execution time increase.

Moreover, users can select the real objects on which the hole detection has to be performed.

When blobs and holes are built, the inclusion relationship between holes and objects is computed. Finally, as holes are well-formed EasyObject coded elements, all the EasyObject functions implemented to analyze the objects work with them too.

EasyObject supports the restriction of the blob analysis to rectangular and nested ROIs as well as to complex or disconnected-shape regions of the image thanks to the flexible masks.

This library has been re-factored to globally improve the execution time, especially for large images and images with numerous objects.

From Open eVision 1.1, EasyObject is accessible through a new object-oriented API centered on the ECodedImage2 class. Earlier versions of EasyObject are not compatible with this new EasyObject API. For maintenance purpose, the legacy API is still available and documented in a dedicated section.

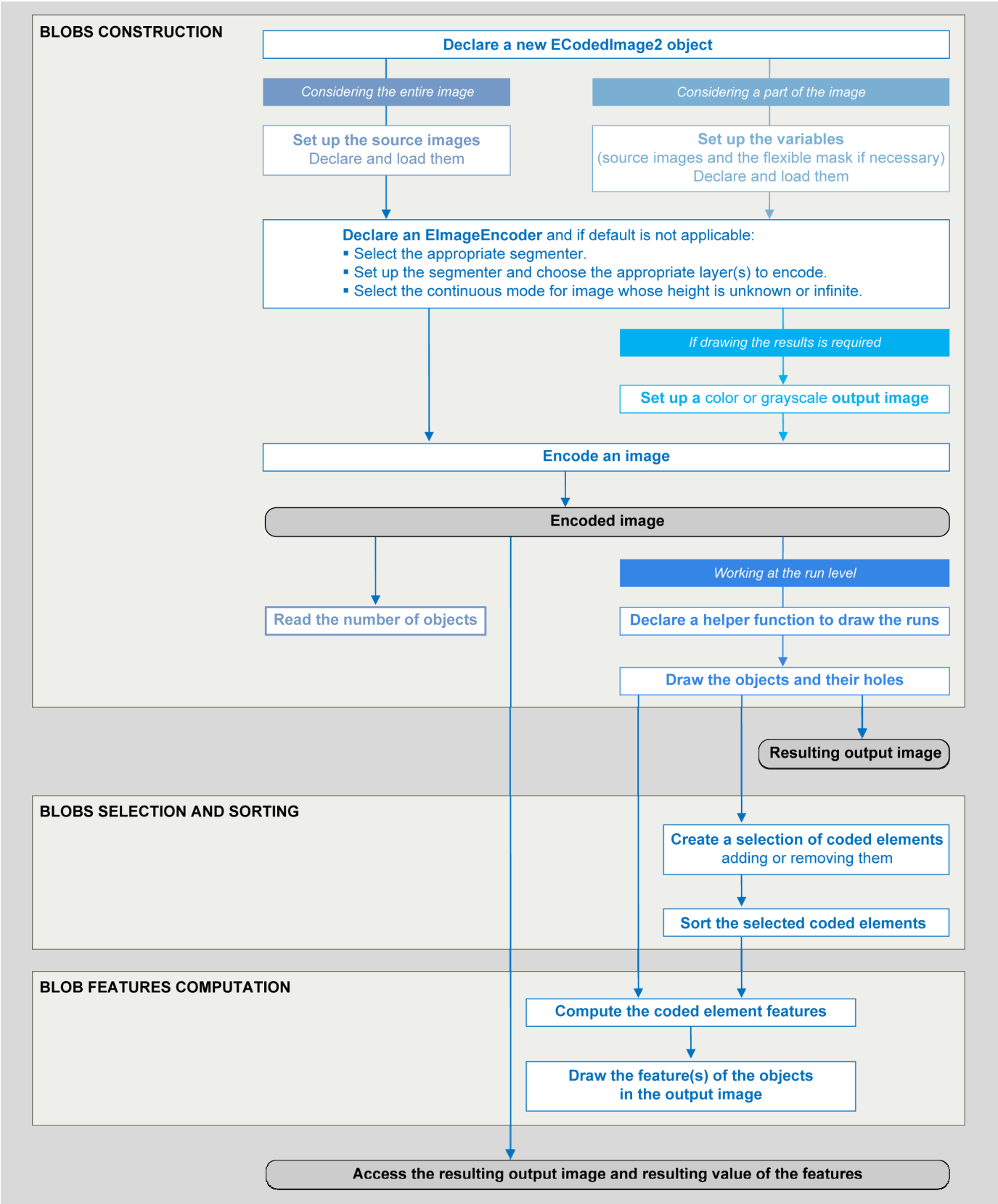
The library methods fall in the following categories:

Image encoding

- **Run construction**
  - Segmentation
    - Grayscale single threshold
    - Grayscale double threshold
    - Color single threshold
    - Reference image
    - Image range
    - Labeled image
  - Pixel aggregation
- **Object construction:** run aggregation into objects

- **Hole construction:** run aggregation into holes
- **Continuous mode**
- **Object feature extraction:** geometric parameter computation
- **Object selection and sorting:** according to any feature value

2. EasyObject Workflow



EasyObject workflow

3. Constructing the Blobs

To construct the blobs, EasyObject encodes the images with a **two-step process**.

- The **segmentation** process classifies the pixels of the source image according to their value and creates layers.
- The **pixel aggregation** assembles connected pixels to build coded elements (blobs) for each layer separately.

The so-called image encoder of EasyObject analyses the blobs of the image and store the result into a coded image. Such a data structure maintains a set of superimposed, mutually exclusive image layers. The pixels of a given image layer are characterized by the same properties, such as being above a threshold. An image encoder takes an image as an input, and stores its results (i.e. the computed blobs) into a coded image.

The EImageEncoder class provides methods to configure the image segmentation and is responsible for:

- the encoding of an image as an instance of the **ECodedImage2** class,
- the extraction of the runs (see "Working at the Run Level" in Image Encoder) in a source image,
- the aggregation of the runs into objects,
- as well as the proper handling of the continuous mode (see Normal vs. Continuous Mode).
- If applicable, select the appropriate image segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.

To build the objects of the image, the image encoder is applied on the source image and the result is stored into the coded image, using EImageEncoder::Encode. The encoding of an image can be restricted to an arbitrary shaped area using the flexible masks as an argument.

Once the image is encoded, it is already possible to consult the number of objects that have been built.

3.1 Image Segmenters

Open eVision provides seven segmentation methods described below.

A segmentation method can be get and set with the following functions: GetSegmentationMethod and SetSegmentationMethod.

Grayscale Single Threshold (by default)

EGrayscaleSingleThresholdSegmenter is applicable to BW8 and BW16 grayscale images and produces coded images with two layers:

- The **black layer** (usually Layer 0) contains unmasked pixels having a gray value strictly below the Threshold value
- The **white layer** (usually Layer 1) contains the remaining unmasked pixels, i.e. unmasked pixels having a gray value greater or equal to the Threshold value.

EasyObject provides 5 thresholding methods that are applicable to all pixel of the image:

- **Absolute** (integer value): It represents the first gray value of the white layer. It must be set with **SetAbsoluteThreshold** method. It can also be get with **GetAbsoluteThreshold** method.
- **Relative** (%): The relative threshold is a user-defined float value in range 0 to 1. It represents the fraction of image pixels that belongs to the Black layer. It must be set with **SetRelativeThreshold** method. It can also be get with **GetRelativeThreshold** method.

- **Minimum Residue** (by default): The threshold is an automatically computed value such that the quadratic difference between the source and thresholded image is minimized.
- **Maximum Entropy**: The threshold is an automatically computed value such that the entropy (i.e. the amount of information) of the resulting thresholded image is maximized.
- **IsoData**: The threshold is an automatically computed value that lies halfway between the average dark gray value (i.e. gray levels below the threshold) and average light gray values (i.e. gray levels above the threshold).

Grayscale Single Threshold with a minimum residue thresholding method is set by default. Only the objects whose pixels have a value that is above this threshold are encoded.

Grayscale Double Threshold

EGrayscaleDoubleThresholdSegmenter is applicable to BW8 and BW16 grayscale images and produces coded images with three layers:

- The **black layer** (usually Layer 0) contains unmasked pixels having a gray value strictly below the Low Threshold value
- The **white layer** (usually Layer 2) contains unmasked pixels having a gray value above or equal the High Threshold value
- The **neutral layer** (usually Layer 1) contains the remaining unmasked pixels.

The Low Threshold and High Threshold are user-defined integer values applicable to all pixels of the image. They must be set with SetLowThreshold and SetHighThreshold methods respectively. The threshold values are also gettable with GetLowThreshold and GetHighThreshold methods respectively.

Color Single Threshold

EColorSingleThresholdSegmenter is applicable to C24 color images; it produces coded images with two layers:

- The **white layer** (usually Layer 1) contains unmasked pixels that belong to the cube of the color space defined by the threshold point and the white point (255,255,255).
- The **black layer** (usually Layer 0) contains the remaining unmasked pixels.

The **Color Threshold** is a set of three **user-defined** integer values designating a color in the color space. It is applicable to all pixels of the image.

It must be set with SetThreshold method. It can be get with GetThreshold method.

Color Range Threshold

EColorRangeThresholdSegmenter is applicable to C24 color images; it produces coded images with two layers:

- The **white layer** (usually Layer 1) contains unmasked pixels that belong to the cube of the color space defined by the Low Threshold point and the High Threshold point.
- The **black layer** (usually Layer 0) contains the remaining unmasked pixels.

The Low Threshold and High Threshold are each a set of three user-defined integer values designating a color in the color space. They are applicable to all pixels of the image. They must be set with SetLowThreshold and SetHighThreshold methods respectively. The color threshold values are also gettable with GetLowThreshold and GetHighThreshold methods respectively.

Reference Image

EReferenceImageSegmenter is applicable to BW8, BW16, and C24 images; it produces coded images with two layers. The threshold is defined for each pixel individually by means of a Reference Image of the same type as the source image.

- For **grayscales** images, the **white layer** (usually Layer 1) contains unmasked pixels having a gray value in a range defined by the gray value of the respective pixel in the Reference Image and the white Color.
- For **color images**, the **white layer** (usually Layer 1) contains unmasked pixels having a color inside the cube of the color space defined by the color of the respective pixel in the reference Image and the white Color.
- The black layer (usually Layer 0) contains the remaining unmasked pixels.

Pointers to the **Reference Image** can be set or get using the functions associated with the type of the source image:

- BW8
  - **SetReferenceImageBW8**
  - **GetReferenceImageBW8**
- BW16
  - **SetReferenceImageBW16**
  - **GetReferenceImageBW16**
- C24
  - **SetReferenceImageC24**
  - **GetReferenceImageC24**

Image Range

In the following cases, a constant threshold can not be sufficient:

- the background is not uniform enough,
- the illumination is not uniform across the image
- when only differences between the image and a reference image (ideal) are to be enhanced,

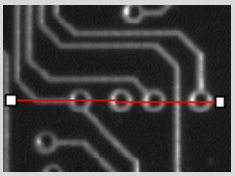
In these cases, a segmentation using **pixel-by-pixel thresholding** is useful. It gives an allowed range of values for each pixel.

The problem is to find ways to prepare the high and low reference images in a suitable way.

Defining a range for each pixel

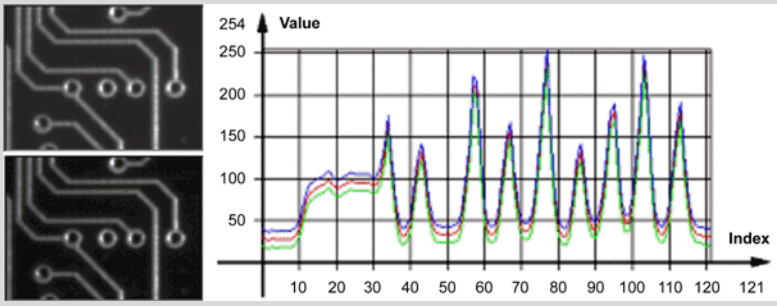
One approach is to start from an image of the scene without defects and to add security margins before comparison.

The validity range for each pixel is specified by giving these two images: the first image (low image) gives the minimum value that is allowed for each pixel, whereas the second image (high image) gives the maximum value. Given the source image, we thus define the low (resp. high) image as being the source image minus (resp. plus) a fixed value:



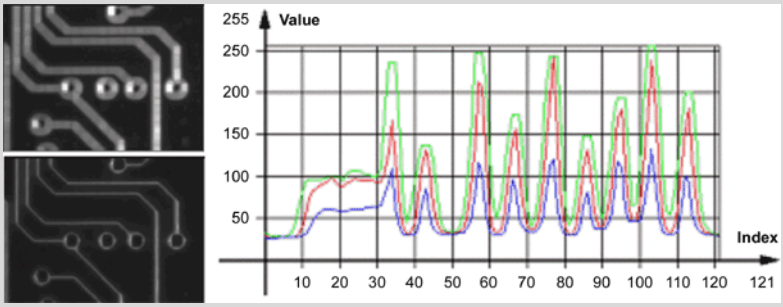
Source image

Because of noise and slight variations in the illumination, some tolerance on the gray-level values must be provided. The high [low] threshold image is obtained by adding [subtracting] a constant all over the image. (This assumes that the noise distribution is uniform and additive).



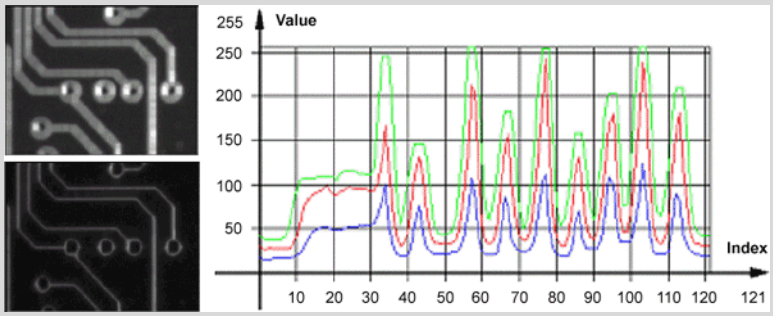
Gray-level tolerance margins

Additionally, mis-registration must be corrected: for various reasons, the image to be inspected may be acquired with a slight shift in some direction. One way to cope is to allow some displacement freedom by enlarging the light and dark areas. This is achieved by means of dilate and erode morphological operations. The geometric tolerance margin so obtained is roughly as large as the morphological filter size.



Geometric tolerance margins

Combining both kinds of tolerance margins gives the best results.



Combined margins

Working with the Image Range Segmenter

EImageRangeSegmenter is applicable to BW8, BW16, and C24 images; it produces coded images with two layers.

The low threshold and the high threshold are defined for each pixel individually by means of two reference images of the same type as the source image: the Low Image and the High Image.

- For **grayscales** images, the **white layer** (usually Layer 1) contains unmasked pixels having a gray value in a range defined by the gray value of the corresponding unmasked pixels in the Low Image and the High Image.
- For **color** images, the **white layer** (usually Layer 1) contains unmasked pixels having a color inside the cube of the color space defined by the colors of the corresponding unmasked pixels in the Low Image and the High Image.
- The **black layer** (usually Layer 0) contains the remaining unmasked pixels.

Pointers to the **Low Image** can be set or get using the functions associated with the type of the source image:

- BW8
  - **SetLowImageBW8**
  - **GetLowImageBW8**
- BW16
  - **SetLowImageBW16**
  - **GetLowImageBW16**
- C24
  - **SetLowImageC24**
  - **GetLowImageC24**

Pointers to the **High Image** can be set or get using the functions associated with the type of the source image:

- BW8
  - **SetHighImageBW8**
  - **GetHighImageBW8**
- BW16
  - **SetHighImageBW16**

- **GetHighImageBW16**

- C24
  - **SetHighImageC24**
  - **GetHighImageC24**

Labeled Image

ELabeledImageSegmenter is applicable to is applicable to BW8, and BW16 grayscale images; it produces coded images with a number of layers equal to the maximum number of gray values: 256 for BW8 images or 65536 for BW16 images. The layer n contains all the unmasked pixels having a gray value equal to n.

By default, all layers are encoded. However, it is possible to restrict the encoding to a single range of layers with SetMinLayer and SetMaxLayer functions. The GetMinLayer and GetMaxLayer functions returns the lowest and the highest values of the index range respectively.

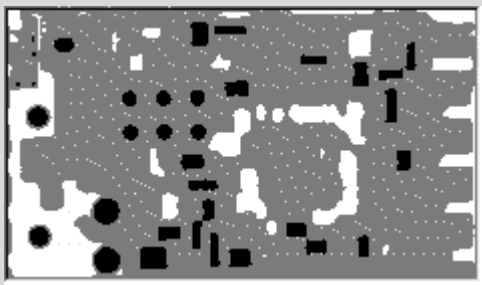
Binary Image

EBinaryImageSegmenter is applicable to BW1 binary images; it produces coded images with two layers: The Black layer (usually, with index 0) contains the unmasked pixels having a binary value equal to zero; and the White layer (usually, with index 1) contains the remaining unmasked pixels, i.e. unmasked pixels having a binary value equal to one.

3.2 Image Encoder

The image segmenter is responsible for constructing the runs of an input image. A run is a sequence of adjacent pixels on the same line of the image that share homogeneous properties (such as being above a given threshold). These runs will thereafter be merged in objects by the image encoder. The objects that belong to a specific layer can be built or not, depending on the needs. The objects may have holes.

The class representing the objects (EObject) derives from an abstract class ECodedElement.



Objects building

Selecting the Layers to Encode

Depending on the segmentation method, a single or several layers are encoded by default (refer to the Image Segmenters topic). Then, the image encoder does not encode the pixels from the other layers.

The function **GetMaxLayerIndex** returns the highest value of the Layer Index. It is available for all segmenters.

Enabling/disabling the layer encoding for each layer individually

The following tables list, for each layer the Set/Get function names, and the default value of the Boolean property.

Two-layer segmenters

Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerEncoded	IsBlackLayerEncoded	FALSE
White layer	SetWhiteLayerEncoded	IsWhiteLayerEncoded	TRUE

Three-layer segmenters

Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerEncoded	IsBlackLayerEncoded	FALSE
White layer	SetWhiteLayerEncoded	IsWhiteLayerEncoded	FALSE
Neutral layer	SetNeutralLayerEncoded	IsNeutralLayerEncoded	TRUE

Manually Assigning a Layer Index to Each Layer Individually

The following table list, for each layer the Set and Get function names, and the default value of the Boolean property.

Two-layer segmenters

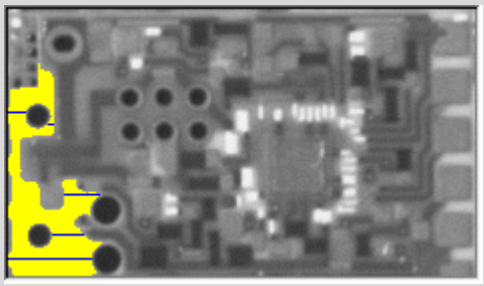
Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerIndex	IsBlackLayerIndex	0
White layer	SetWhiteLayerIndex	IsWhiteLayerIndex	1

Three-layer segmenters

Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerIndex	IsBlackLayerIndex	0
Neutral layer	SetNeutralLayerIndex	IsNeutralLayerIndex	1
White layer	SetWhiteLayerIndex	IsWhiteLayerIndex	2

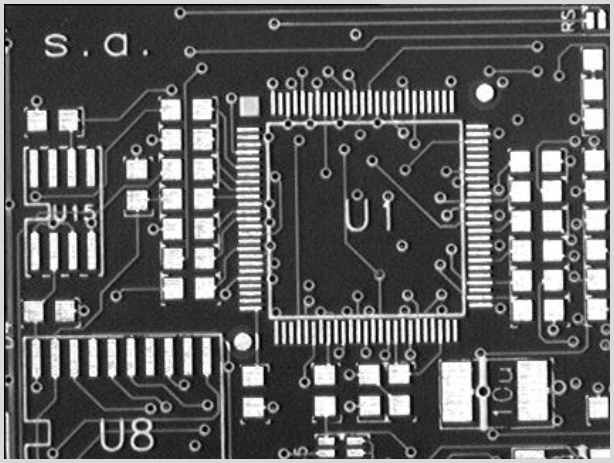
Working at the Run Level

For the sake of computational efficiency, the objects are described as lists of runs. A run is a sequence of adjacent pixels on the same line of the image that share homogeneous properties (such as being above a given threshold). These runs will thereafter be merged in objects by the image encoder.

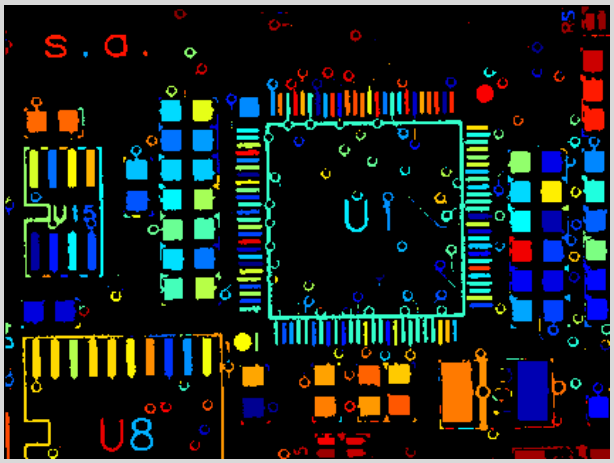


A single object with five enhanced runs

EasyObject allows working at the object level but also down to the runs level. This allows faster processing in critical cases. This is useful to compute custom features on the objects by then list all runs belonging to a given object. An example of working at the run level is described hereafter. In this example, the runs will be colored in the output image.



Source image

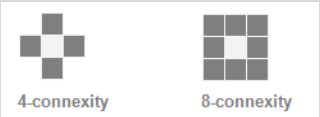


Rendering of the white layer

- **Declare a new ECodedImage2** object
- **Declare an EImageEncoder** and, if applicable, select the appropriate segmenter. Setup the segmenter and choose appropriate layer(s) to encode.
- **Setup an output image**
- **Encode the image**
- **Color the runs in the output image**. Iterate over the objects of a specific layer by constructing a loop and then a run iterator object (**ECodedElement::RunsIterator**). This iterator allows the browsing of the runs of the considered object. Once the iterator is over a run of the considered object, the inner loop will process the pixels spanned by this run in the output image.
- **Select a specific layer**

Connexity

Pixels can touch each other along an edge or by a corner. If only pixels touching by an edge are considered neighbors, one speaks of Four Connexity; if pixels touching by a corner are also considered neighbors, one speaks of Eight Connexity. EasyObject supports both conventions but Eight Connexity is the default value.



3.3 Normal vs. Continuous Mode

Normal Mode

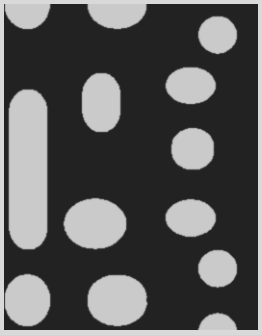
By default, EasyObject works in the **normal mode**. This means that the image encoder does not track the blobs across several successive images. In the normal mode, EasyObject works with one image, without keeping blobs in memory. All the blobs are returned as objects.

Continuous Mode

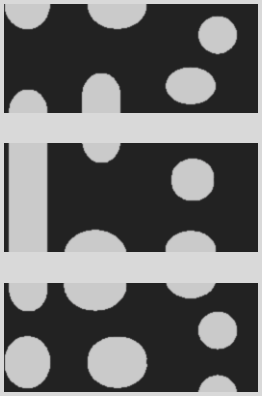
Conversely, the continuous mode allows an image whose height is a priori unknown or infinite (e.g. coming from a line-scan camera) to be processed by EasyObject. In this mode, the image encoder is fed with the sequence of chunks the large image is made of. The continuous mode allows detecting objects that straddle several successive image chunks.

When working in the continuous mode, the image encoder encodes only the objects that contain no run touching the last row of the source image. The objects touching the inferior border of the image are not written in the coded image: These objects are indeed expected to continue in the subsequent image chunks. Such objects are kept in memory, and are consumed when analyzing the subsequent images.

A large image is assumed to be divided in several chunks that are stored in the array EImageBW8 chunk[x].



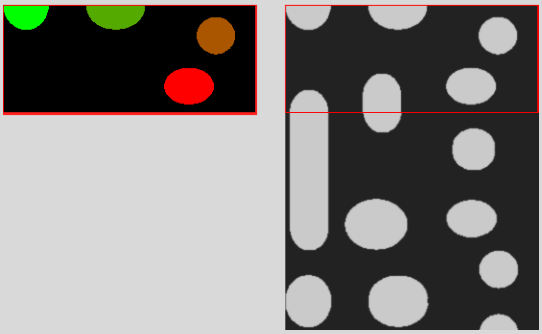
Original image



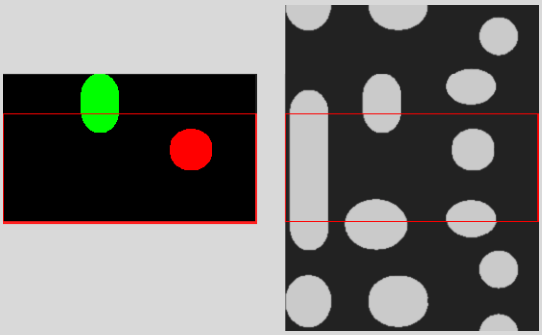
Three chunks of the image

For the example here above, we generate a sequence of color images that will exhibit the objects encoded over the successive chunks.

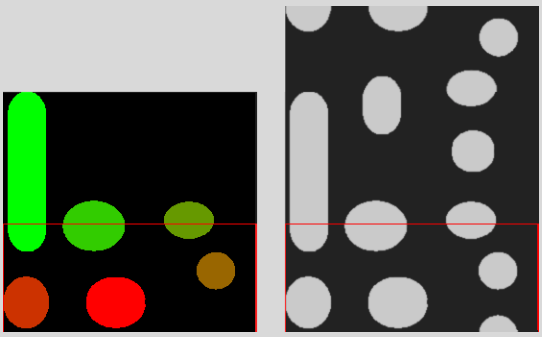
- **Draw the objects encoded in a layer of a coded image.** This code is essentially the same as in "Browsing Runs" code snippet. The only difference is that an offset can be applied along the Y-axis.
- **Define of a function to draw the objects of a layer.** If a coded image contains objects that got started in some previous image: the runs of this object that come from previous image are assigned with a negative Y-coordinate. The zero Y-coordinate corresponds to the first row of the most recently encoded image. The convention is thus to assign the lowest Y-coordinate to the oldest run in the encoded objects. The method **EImageEncoder::GetStartY** can be used to obtain the Y-coordinate of this oldest run.It is necessary to define a function that displays the content of a layer of a coded image.Each object can be displayed with a different color, this color being computed by the function **GetFadedColor** that was described in the "Building Holes" section. This function closely follows the function **DrawRuns** from the same section, but it is adapted to the continuous mode by taking the value of **EImageEncoder::GetStartY** into account.
- **Enable the continuous mode is pretty straightforward.** This is done through the property **EImageEncoder::SetContinuousModeEnabled** of the image encoder. Additional variables can also be declared, for example to store the successive encoded image, or to hold the output images.
- **Analyze the successive chunks.** To encode the successive chunk use the function **Encode(chunk[count], codedImage)** and then **DrawLayer**. This is exactly the same as in the basic use case. **Note:** The variable count spans the integers 0, 1 and 2. When an object from a chunk is not completed yet, and as such, it is kept in the internal memory of the image encoder.



Content of layerImage when count equals 0, after the application of DrawLayer.
Chunk of the large image that is under consideration.
Notice that two objects at the lower-left part of the image chunk are not encoded, as they touch the border of the chunk.



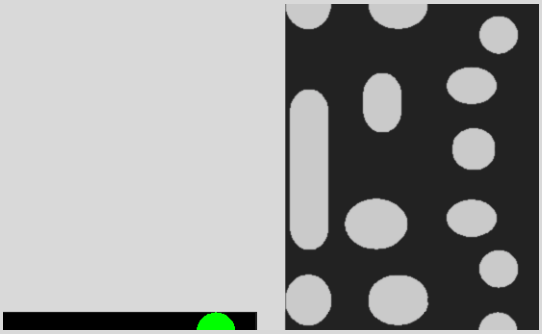
When count reaches the value 1, one of these two objects becomes completed,
which leads to the encoding of the following image.
Two other objects are not encoded yet at this time.
Here is the result of the encoding of the last chunk (count = 2).



Three objects from the previous chunks have been closed, and have thus been encoded.

Flushing the Continuous Mode

After the encoding of the three image chunks, there remains one object to be completed (the one in the bottom-right corner of the large image). However, there is no available chunk anymore to complete the encoding of the object.In some applications, it is necessary to explicitly close this object. The remaining objects can be successfully encoded using the flushing of the image encoder. The internal memory of the image encoder is then empty.



Result of the flush

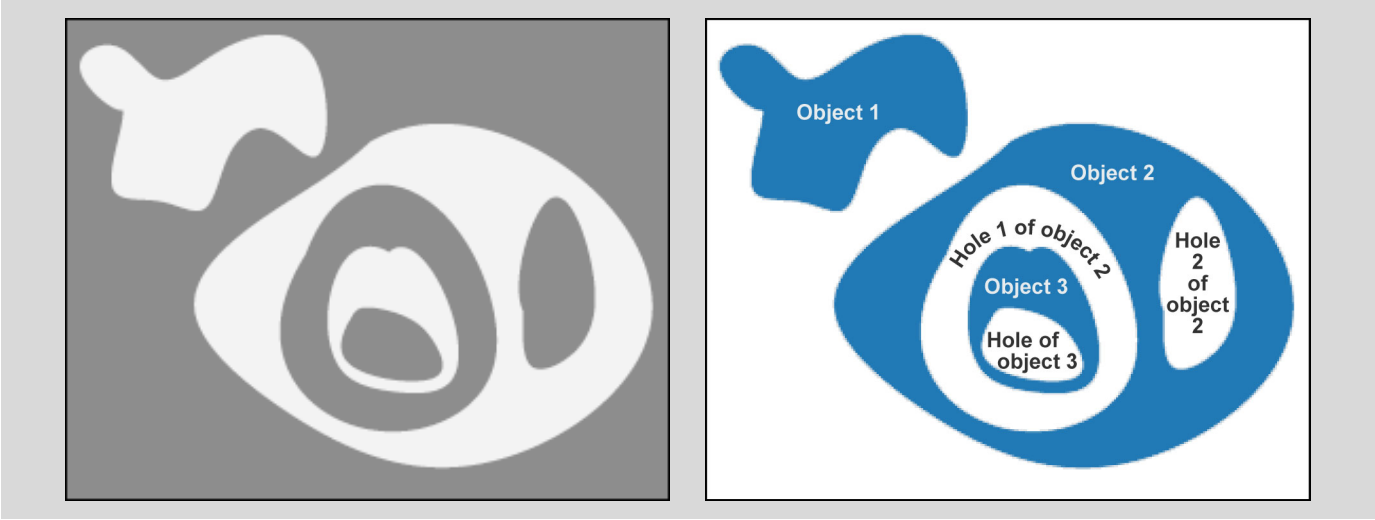
3.4 Holes Construction

EasyObject supports the inspection of holes in defined objects.

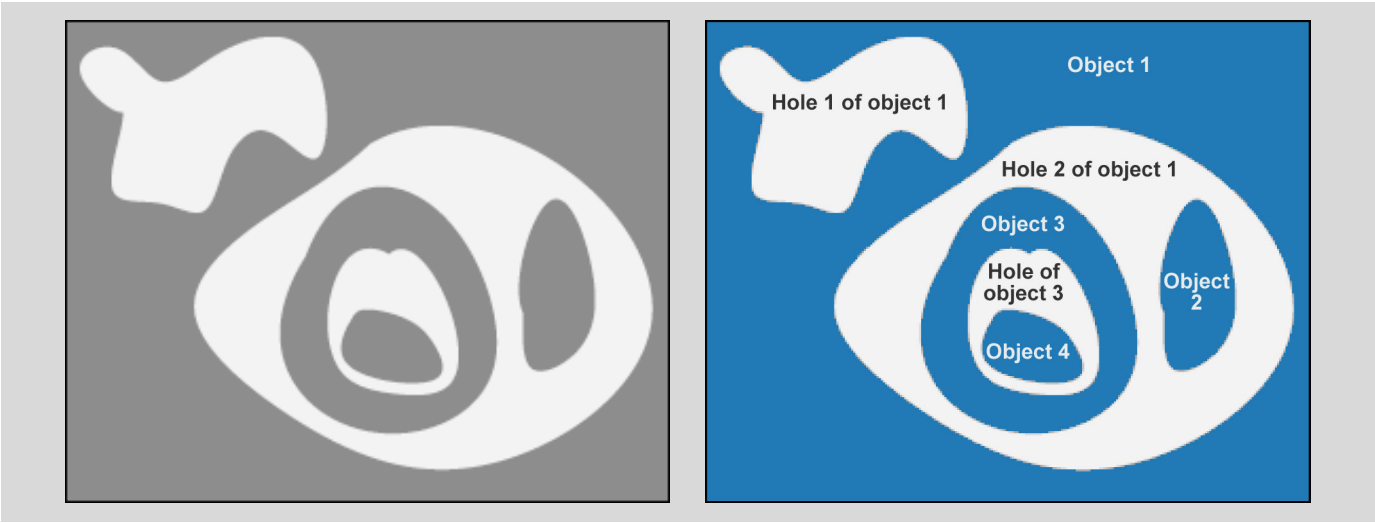
A hole of a given object is defined as a set of connected pixels that are entirely surrounded by this object, but that do not belong to this object, and such that this set is maximal (i.e. it is not possible to add another pixel to the hole). Note that this definition is influenced by the considered connexity mode.

The EasyObject library manages the relationship between objects and holes, defining parent objects for holes. Here are the two major rules managing these relationships:

- A hole has always an object as its parent; the object entirely surrounds the hole!
- A hole has no child. Objects can be found inside a hole; however such objects are excluded from the hole; they are rather considered as separate objects!



Encoding the white layer (3 objects and 3 holes)

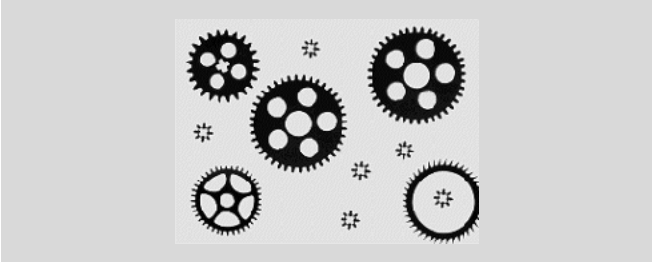


Encoding the black layer (4 objects and 3 holes)

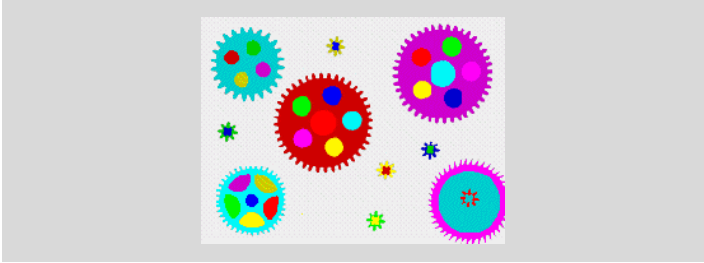
In EasyObject, the construction of the holes is lazy. This means that there is no need to explicitly build the holes. They are constructed on-the-fly when required.

Holes are managed as the objects themselves, benefiting from the same geometrical features. Both, the class representing the objects (EObject) and the class that representing the holes (EHole) derive from a common abstract class (ECodedElement). Consequently, it is possible to share a function for objects and holes.

Coloring the holes



Raw image



Building of objects and all holes

- **Declare a new ECodedImage2** object
- **Declare an EImageEncoder** and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
- **Setup an output image**
- **Encode the image**
- **Declare a helper function to draw the runs.** A helper function (see also the following section Object Construction/Working at the Run Level) is defined and it will draw the runs of a given object or hole in an output image, using, for example, a given color. Such a function can be shared for objects and holes.

- **Draw the objects and their holes in the output image** It is necessary to iterate over the objects of the chosen layer. The runs of each object will be drawn (**DrawRuns**) using a specific color by calling the helper function. Then, iterate over the holes over the current object, and draw the holes runs. Each hole of a given object is drawn with a different color that is computed in a global function (**GetFadedColor**). This function returns a color that is for example a gradation of red to green colors depending upon the hole index.

4. Selecting and Sorting the Blobs

The object segmentation process is "blind" in the sense that it considers any blob as an object. Most of the time, this includes noisy pixels appearing as tiny objects, as well as foreign features. Selection provides an elegant way to get rid of the unwanted blobs and keep only the relevant ones, by filtering them. In EasyObject, the filtering of the objects is handled by the EObjectSelection class.

For the sake of efficiency, object features are computed only on the currently selected objects. This allows to compute the features only when strictly required.

Selection

A selection can be modified in the following ways:

- Directly adding or removing a single object, a hole or a whole **layer** to/from a selection.
- Adding or removing objects or holes based on the value of some specified **feature** (see the feature list in Computing the Coded Element Features).
- Adding or removing objects or holes based on their specific **position**, or whether they lie within a specified ROI rectangle bounds.

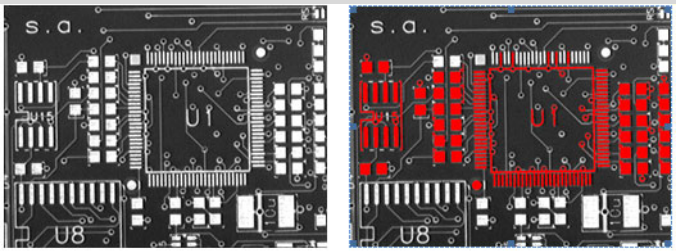
These actions can be performed using the EObjectSelection::Add... or EObjectSelection::Remove... methods. Please note that these actions can be cascaded and combined at will in a single selection.

It is possible to clear a previous selection using EObjectSelection::Clear.

Sorting

Once a selection is built, its elements can be sorted according to the value of any of their features.

In this example, we select the objects that lay in the middle band of an image, and that have a surface that is above 100 pixels.



Source image, and selection of objects

- **Declare a new ECodedImage2** object
- **Declare an EImageEncoder** and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
- **Setup an output image**
- **Encode the image**

- **Create a selection of objects.** To create a selection of objects, an instance of the **EObjectSelection** class must be created. Initially, the selection is empty. Objects must be added to this selection, for instance through **EObjectSelection::AddObjects**
- **Remove objects based on the value of one feature at a time.** The objects in a selection can be unselected by calling one of the **EObjectSelection::Remove** methods.
- **Remove the objects based on their position** can be performed using **RemoveUsingFloatFeature**. For details, see also of the "Working at the Run Level"
- **Sort the selected objects**: **EObjectSelection::Sort**
- **Access the selected objects**

5. Computing the Coded Element Features

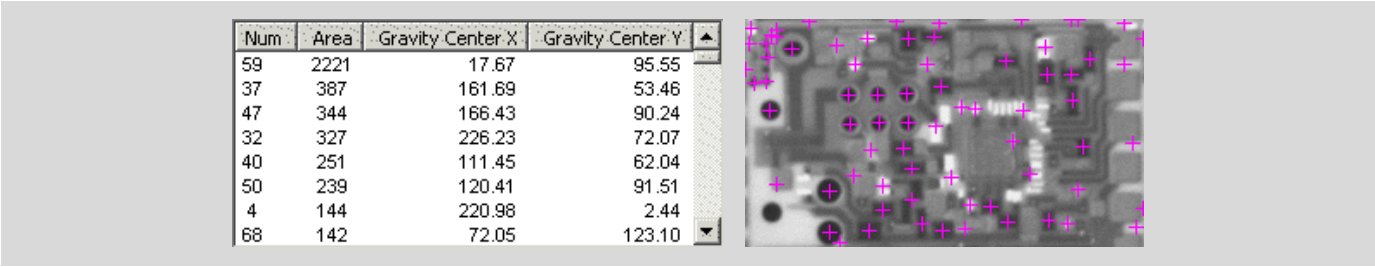
Once an image has been encoded, the value of the coded elements features is readily available. Each feature is accessed through a property. The values can be rounded down.

The available features can be found in the ECodedElement class.

An ECodedElement corresponds to either an object, either a hole in an object. This abstract class provides a large set of methods applicable to a particular coded element.

The set includes methods:

- to get the features of a coded element,
- to draw coded elements,
- and to render flexible masks: **ECodedElement::RenderMask**.



Num	Area	Gravity Center X	Gravity Center Y
59	2221	17.67	95.55
37	387	161.69	53.46
47	344	166.43	90.24
32	327	226.23	72.07
40	251	111.45	62.04
50	239	120.41	91.51
4	144	220.98	2.44
68	142	72.05	123.10

Features computation and display

The objects and the holes and their features are efficiently accessible randomly (i.e. in an index-based fashion).

How to Compute Coded Element Features?

- **Declare a new ECodedImage2** object
- **Declare an EImageEncoder** and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
- **Create an output image**: copy, pixel by pixel, the (grayscale) source image into a (color) output image if the drawing of the resulting features has to be colored.
- **Encode the source image**
- **Draw the features for each object in a layer**

The drawing of the object feature is done for each object in a chosen layer. Then the result is read and it can be round down. A specific drawing can be created to mark the feature (for example, draw a target for a gravity center).

5.1 Computable Features

Methods prefixed with "Get" indicate a lazy evaluation. This means that the result is computed once on the first invocation of the method. The result is then cached.

Methods prefixed with "Compute" indicate that the function is reevaluated at every invocation. The result is never cached. This is the case for methods having an additional parameter or when the result is too big.

Position

Limit (top, bottom, left, right)	**ECodedElement::GetTopLimit** **ECodedElement::GetBottomLimit** **ECodedElement::GetLeftLimit** **ECodedElement::GetRightLimit**
Gravity center (X and Y)	**ECodedElement::GetGravityCenter** **ECodedElement::GetGravityCenterX** **ECodedElement::GetGravityCenterY**
Weight gravity center (X and Y)	**ECodedElement::ComputeWeightedGravityCenter**

Gravity center and weight gravity center

The **gravity center returns** the abscissa of the gravity center of the coded element. While the **weight gravity center** computes the gravity center of a given image over a coded element.

Extents

Area (pixel count)	**ECodedElement::Area**
Feret box (center X and Y, height, width with distinct orientation angles at 22, 45, 68 degrees)	**ECodedElement::ComputeFeretBox** ECodedElement::GetFeretBox22Box ECodedElement::GetFeretBox22Center ECodedElement::GetFeretBox22CenterX ECodedElement::GetFeretBox22CenterY ECodedElement::GetFeretBox22Height ECodedElement::GetFeretBox22Width ECodedElement::GetFeretBox45Box ECodedElement::GetFeretBox45Center ECodedElement::GetFeretBox45CenterX ECodedElement::GetFeretBox45CenterY ECodedElement::GetFeretBox45Height ECodedElement::GetFeretBox45Width ECodedElement::GetFeretBox68Box

	ECodedElement::GetFeretBox68Center ECodedElement::GetFeretBox68CenterX ECodedElement::GetFeretBox68CenterY ECodedElement::GetFeretBox68Height ECodedElement::GetFeretBox68Width
Bounding box (center X and Y, height, width)	**ECodedElement::GetBoundingBox** **ECodedElement::GetBoundingBoxCenter** ECodedElement::GetBoundingBoxCenterX ECodedElement::GetBoundingBoxCenterY ECodedElement::GetBoundingBoxHeight ECodedElement::GetBoundingBoxWidth
Minimum enclosing rectangle (angle, center X and Y, heath, width)	**ECodedElement::MinimumEnclosingRectangle** **ECodedElement::MinimumEnclosingRectangleAngle** ECodedElement::MinimumEnclosingRectangleCenter ECodedElement::MinimumEnclosingRectangleCenterX ECodedElement::MinimumEnclosingRectangleCenterY ECodedElement::MinimumEnclosingRectangleHeight ECodedElement::MinimumEnclosingRectangleWidth

The **Feret box** is defined as the rectangle with the minimum surface rotated at a specified angle that contains all the pixels center points of an object.

The **Bounding box** is the Feret Box at 0°.

The **Minimum enclosing rectangle** is the FeretBox with the minimum surface accross all the possible angles.

The **width** of a **FeretBox rectangle** is actually the length of the rectangle side that exhibits the smallest angle with the X-axis. This is NOT necessarily the smallest side!

The **height** of a FeretBox rectangle is actually the length of the other side of the rectangle.

Miscellaneous

Starting point of the object contour (X and Y)	**ECodedElement::GetContour** **ECodedElement::GetContourX** ECodedElement::GetContourY
Largest run	**ECodedElement::GetLargestRun**
Run count	**ECodedElement::GetRunCount**
Object number (index)	**ECodedElement::GetLayerIndex** **ECodedElement::GetElementIndex**
Pixel gray-level value (average, deviation, variance)	**ECodedElement::ComputePixelGrayAverage** **ECodedElement::ComputePixelGrayDeviation** ECodedElement::ComputePixelGrayVariance
Pixel gray-level value (min and max)	**ECodedElement::ComputePixelMax**

	ECodedElement::ComputePixelMin

Ellipse of inertia

Eccentricity of the ellipse of inertia	**ECodedElement::Eccentricity**
Moment	**ECodedElement::GetCentralMoment** **ECodedElement::GetMoment** ECodedElement::GetNormalizedCentralMoment
Ellipse (angle, height, width)	**ECodedElement::GetEllipseAngle** **ECodedElement::GetEllipseHeight** ECodedElement::GetEllipseWidth
Second order geometric moments (Sigma: X, XX, XY, Y, YY)	**ECodedElement::GetSigmaX** **ECodedElement::GetSigmaXX** ECodedElement::GetSigmaXY ECodedElement::GetSigmaY ECodedElement::GetSigmaYY

Note. The object perimeter can be measured indirectly by tracing the object contour by means of contouring methods, and counting the pixels.

From the standard geometric features, other ones can be derived. For instance, object elongation is computed as the ratio of large to short ellipse axis or limit height over limit width. Object circularity is defined as the ratio of squared perimeter over four times pi times the object area.

Note. Formulas (**N** = area):

$$\sigma_X = I_X = \frac{1}{N}\sum\left(Y_i - \overline{Y}\right)^2$$

$$\sigma_Y = I_Y = \frac{1}{N}\sum\left(X_i - \overline{X}\right)^2$$

$$\sigma_{XX} = I_A = \frac{I_X + I_Y}{2} + \sqrt{\left(\frac{I_X + I_Y}{2}\right)^2 + I_X I_Y + I_{XY}^2}$$

$$\sigma_{XY} = I_{XY} = \frac{1}{N}\sum\left(X_i - \overline{X}\right)\left(Y_i - \overline{Y}\right)$$

$$\sigma_{YY} = I_B = \frac{I_X + I_Y}{2} - \sqrt{\left(\frac{I_X + I_Y}{2}\right)^2 + I_X I_Y + I_{XY}^2}$$

$$\mathrm{WIDTH} = 4\sqrt{I_A}$$

$$\mathrm{HEIGHT} = 4\sqrt{I_B}$$

$$\mathrm{ANGLE} = \mathrm{arccotan}\left(\frac{I_X - I_A}{I_{XY}}\right)$$

Ellipse of inertia

Convex Hull

The convex hull of a shape is the convex polygon of minimum area that completely surrounds an object. The convex hull can be used to characterize the object footprint, as well as to observe concavities. Given that the number of vertices of the convex hull is variable, they are stored in a EPathVector container.

The corresponding function is ECodedElement::ComputeConvexHull.

Convex hull

Graphic Representation

The objects can be drawn onto the source image by means of ECodedImage2::Draw. The following features also have a graphical representation that can be drawn by the means of ECodedImage2::DrawFeature.

Objects	
Diagonals	
Bounding box	
Convex hull	
Ellipse	
Feret box	
Feret box with an angle of 22°	
Feret box with an angle of 45°	
Feret box with an angle of 68°	
Gravity center	
Minimum enclosing rectangle	
Weighted gravity center	

Coordinate System and Conventions

Coordinate system

EasyObject uses a pixel coordinate system where the origin is conventionally at the top left corner of the top left pixel of an image. Consequently, the fractional part of the coordinates of the center of a pixel is ".5". This convention is best suited for the representation of sub-pixel coordinates.

Angles

Accordingly to the mathematical conventions, the angles are now counted inversely: A positive angle brings the X axis on the Y axis.

Evaluating the features: There is one property per feature, withdrawing the need for accessing the feature through an **enumeration**.

6. Flexible Masks in EasyObject

If a searched object has the same characteristics (e. i. same gray level) than other regions of the image that are not of interest, it may be profitable to not consider these other regions for the blob analysis. The image then has to be divided between "do care" and "don't care areas". It can be done using the Open eVision rectangular ROIs or using flexible masks that support complex and disconnected shapes.

EasyObject supports the restriction of the blob analysis to complex or disconnected shaped regions of the image thanks to the flexible masks. They are available for the Encode functions as an argument.

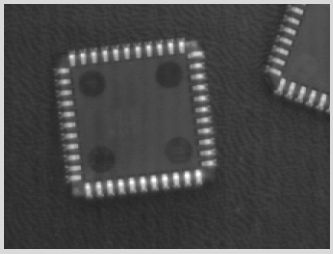
A flexible mask is a BW8 image having the same height and the same width as the source image.

- A pixel value of 0 in the flexible mask masks the corresponding pixel of the source image. The masked pixels don't appear in any Layer of the encoded image.
- Any other pixel value in the flexible mask causes the pixel to be a candidate for the encoding.

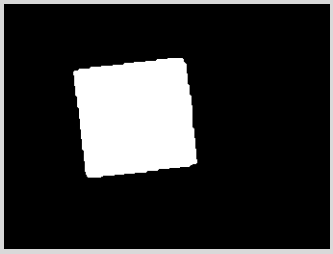
6.1 Using Flexible Masks

The EImageEncoder::Encode function is available with an optional flexible mask argument all the supported types of source images, namely BW1, BW8, BW16, and C24.

Restrict the areas that will be encoded by EasyObject



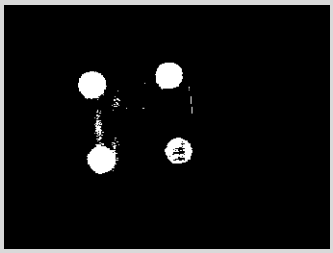
Searching for the four low-contrast circles (chip marks) in the image



Flexible mask used to isolate the central chip and not consider the second chip and the background

- Declare a new **ECodedImage2** object.

- Setup the variables: first declared the source image and the flexible mask and then load them.
- Declare an **EImageEncoder** and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
- **Encode an the source image.** The process of encoding of the chosen layer by restricting the process to the flexible mask is then pretty straightforward.

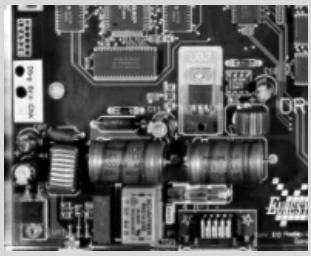


We see that the circles are correctly segmented in the black layer with the grayscale single threshold segmenter

- **Select all the objects of the coded image.**
- **Select the objects of interest** (e.i. by filtering out the objects that are too small).
- **Display the feature of the blob** by iterating over the objects in the selection to display the chosen feature.

6.2 Generating a Flexible Mask from an Encoded Image

A flexible mask can be generated by any application outputting BW8 images or using the Open eVision image processing functions.



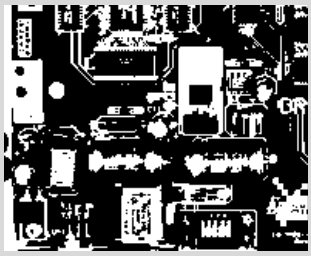
Source image

EasyObject provides three functions to create flexible masks:

Create flexible masks from a specified layer of an encoded image using the ECodedImage2::RenderMask function prototype.

- To encode and extract a flexible mask first, as usual, construct a coded image from the source image.

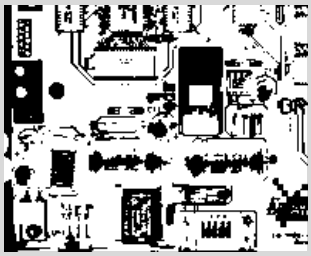
- Choose a segmentation method (for the image above the default method **GrayscaleSingleThreshold** is suitable)
- It is possible to generate a flexible mask from one or several chosen layers of the coded image. Select the layer(s) that should be encoded (e.i. white and the black layers using minimum residue thresholding).
- The **ECodedImage2::RenderMask** method does not automatically resize the mask image. The mask image should have the proper size before the call and it can be done using **mask.SetSize(sourceImage.GetWidth(), sourceImage.GetHeight())**.
- It is now possible to exploit the flexible mask as an argument.



BW8 resulting image that can be used as a flexible mask

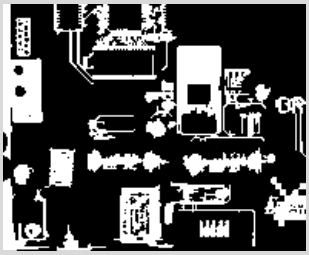
Create flexible masks from a specific coded element (blob or hole) using the ECodedElement::RenderMask method.

- Select the coded elements of interest
- Create a loop extracting a mask from selected coded elements of the coded image.
- Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

Create flexible masks from a selection of coded elements, use the EObjectSelection::RenderMask function prototype. This allows for example to discard small objects resulting from noise.



BW8 resulting image that can be used as a flexible mask

EasyMatch

1. What Is EasyMatch?

EasyMatch is a gray-level and color pattern matching library. It enables you to train the system on a reference pattern and locate its occurrences in other images. This tool is useful when the position of a given part in the field of view is unknown, or if the presence of parts must be controlled.

The library works by using normalized correlation method, i.e. measuring discrepancies between the pattern and the target image. The following features are provided:

- **Multiple pattern occurrences:** several occurrences of a pattern, up to a user-defined number, are returned. Only the reliable ones are retained.
- **Standard, offset-normalized, gain-normalized and fully normalized correlation:** the correlation is computed on continuous tone values (as opposed to binary). It is well known that when the lighting conditions vary, as it is often the case, straight comparison of the pattern and image behaves badly. To cope with this, automatically adjusting the contrast and/or intensity of the pattern before comparison is very effective. This process is known as normalization. EasyMatch provides four distinct normalization modes, depending on whether a gain and/or offset compensation are used.
- **Normal, inverse or mixed contrast:** because of particular lighting effects, an object can appear with inverted contrast (white on black instead of black on white or conversely). Depending on the applications, it can be useful to keep inverted instances or to disregard them. Three matching modes are available: consider positive occurrences only, negative occurrences only or both.
- **Translation, rotation and isotropic/anisotropic scaling:** to find the best matches between the pattern and target image, the target is allowed to translate horizontally and vertically. Additionally, it can be allowed to rotate and/or to change its scale in the X and Y directions simultaneously or independently. The rotation angle and scale factors vary in a user-specified interval. All degrees of freedom can be combined at will.
- **Variable accuracy, up to sub-pixel level:** the accuracy with which the pattern is measured can be chosen (the less accurate, the faster). By default, the position parameters for each degree of freedom are computed with a precision corresponding to the size of a pixel. Lower precision can be enforced. On the other hand, one tenth-of-a-pixel accuracy can be achieved.
- **Don't care pixels:** when the pattern cannot be inscribed in a rectangular ROI, the surrounding of the pattern can be ignored by setting the pixels values below a threshold level. These pixels will not take part in the matching process. The same feature can be used if parts of the template change from sample to sample.
- **Gray-level and color images:** EasyMatch processes BW8 images as well as C24.
- **Non-square pixels:** when images are acquired with non-square pixels, rotated objects appear skewed. Taking the pixel aspect ratio into account can compensate for this effect.

1.1 EasyMatch Working Principles

EasyMatch works by superimposing the pattern over the image and comparing them by computing a **normalized correlation** score. This means that the whole pattern area is processed for each position tried.

If all positions, possibly including the rotated and scaled ones, had to be tried, this would lead to a tremendous amount of work and an unacceptable running time. To alleviate this, a coarse-to-fine approach is used. This means that several search stages, called reductions are performed. The reductions are numbered from 0 [finest] to NumReductions-1 [coarsest])

At the coarsest reduction, an approximate location is found, quickly. Then the location is improved, using the next reductions, and working in a close neighborhood. This arrangement drastically reduces the number of positions to be tried.

At the final stage (FinalReduction), additional processing can be done to achieve sub-pixel accuracy.

Note. The accuracy of the location found at a given reduction is given by the K-th power of 2, where K is the reduction number.

The total number of reductions is dependent upon the MinReducedArea parameter that must be set at learning time. It indicates how small the pattern can be made for rough location. The default value of this parameter usually is a good choice.

When several occurrences are to be found (MaxPositions parameter), the same process of coarse-to-fine search is repeated for the positions giving the best scores.

Because of this, a good match can sometimes be confused for a better one: at the coarsest resolution all matches look alike and their scores are close to each other. In the extreme, a good match may be mistaken for another object. To avoid this, it is possible to compel EasyMatch to consider more instances than needed at the coarse stage. This is the MaxInitialPositions parameter. (This number is progressively reduced to reach MaxPositions in the final stage).

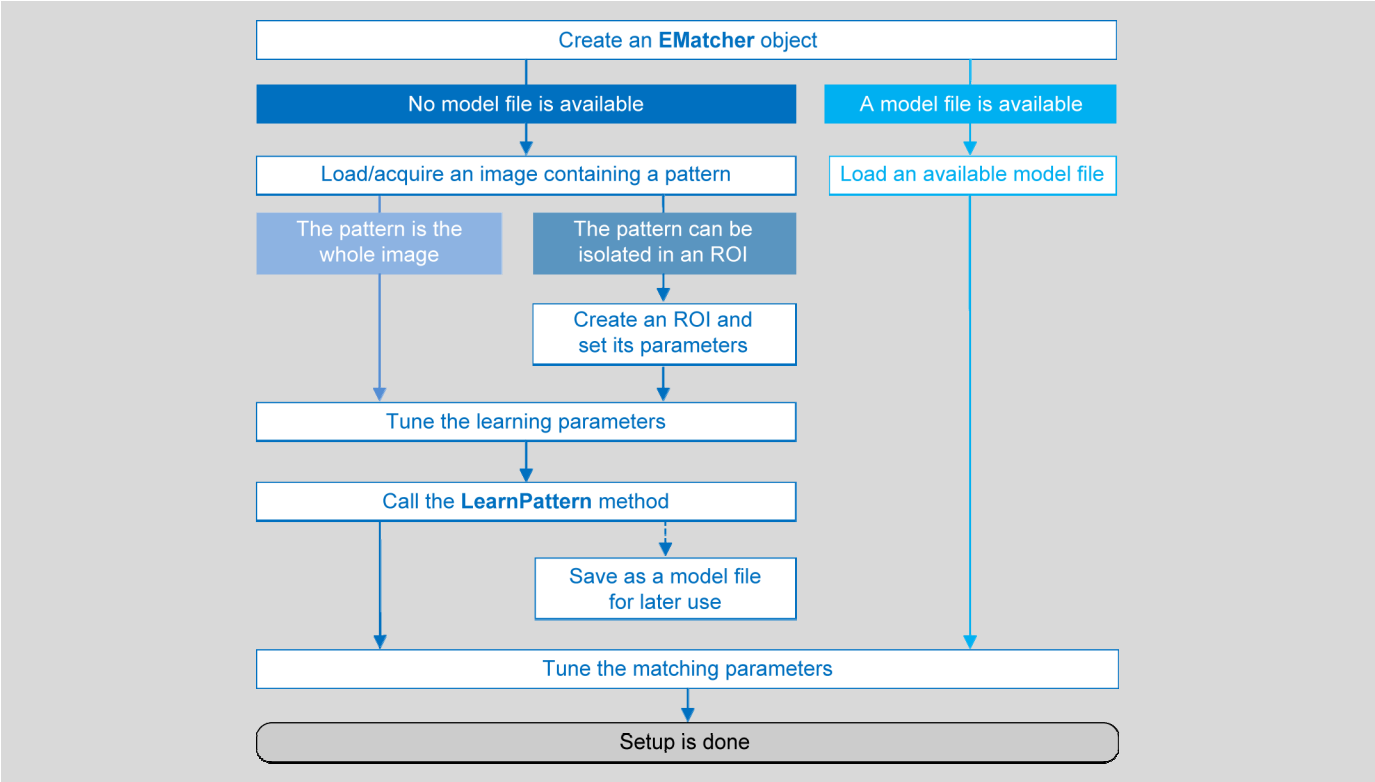
When the requested number of occurrences has been found, they are sorted by decreasing scores. If need be, the worst of them, corresponding to false matches, can be discarded by setting the MinScore threshold.

When computing the correlation score, the pixels marked as "don't care" are not taken into account. A pixel is marked as "don't care" if its value in the pattern falls below the DontCareThreshold value.

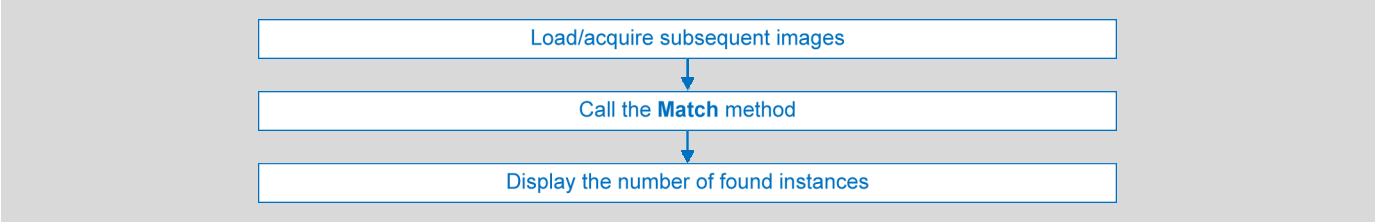
Pattern matching is done in two steps:

1. First, the pattern is shown by defining an ROI that contains the object to be matched. This is the learning phase. The learning is done once for all and the resulting pattern can be saved for later use. Most of the time, the training region can be a rectangle. Anyway, in some cases extraneous features can be included in the training window. To get rid of them, part of the pattern can be defined as a "don't care area". Such pixels will be ignored in the matching process.
2. Then, for each new image, the search can take place. One or more occurrences of the pattern can be located, allowing it to translate and possibly rotate or scale. This is the matching process.

2. EasyMatch Workflow



Learning process



Matching process

3. Choosing the Pattern

In the first place, one has to discover the **degrees of freedom by which the pattern can move**:

- Can it translate?
- Can it rotate?
- Can it scale?

In typical applications, objects can widely translate in the field of view and rotate only slightly. Scaling is less frequent. Regardless the degrees of freedom, the more repeatable the position is, the better.

The next question to be asked is: **is there a single instance of the pattern to be located in the same image or not?** Can we separate these instances in distinct ROIs? If no, multiple occurrences search will be required.

When choosing the object or part of object to be used as a template, the following guidelines should be observed:

- **The template should be representative of the object to be located.** It should always be present and keep the same appearance whatever the lighting conditions. It should also remain at a fixed location with respect to the part. It should be rigid and not change in shape.
- **The template should exhibit good contrast at small scale as well as large scale.** It should remain distinctly visible from a distance, on a reduced image, but also exhibit sharp edges and fine details. This helps both rough and coarse location.
- **The template shape should not be invariant under the degrees of freedom to be measured.**
  - For translation to be easily measured, the pattern should possess contrasted edges in at least two directions.
  - For rotation to be easily measured, the pattern should not be symmetrical by rotation.
  - For instance, a pattern of black and white horizontal stripes cannot detect horizontal translation; a cog wheel cannot help measure large rotations. On the other hand, if rotation does not have to be measured but the part is indeed able to rotate, it is welcome to use a circular pattern.
- **The template should lay on a background free of extraneous objects.** If there is a risk that objects come in close vicinity of the pattern in such a way that the contents of the ROI is not always the same, the corresponding area should be neutralized by means of "don't care" pixels or a mask.
- Always leave a **contrasted margin around the objects** so that both the foreground and background intensities are seen.

4. Learning and Matching Processes

When many degrees of freedom are involved, pattern matching is a complex and lengthy process. In order to reduce the processing time, it is important to understand the working parameters and fine tune them for a given application.

There are two phases in a pattern matching process: the learning process and the matching process.

Learning Process

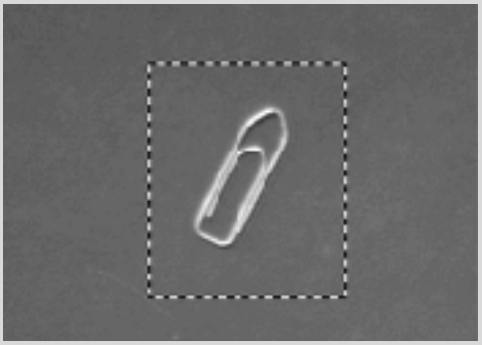
First, one selects an image (or ROI) containing the pattern to be searched for and one lets the matching context pre-process it: this is the learning phase. It is performed using a single function call:

- **LearnPattern**: receives the image/ROI as its argument and pre-processes it.

The learning is done once for all and the resulting pattern can be saved as a model for later use.

The following parameters are relevant at learning time:

- **DontCareThreshold**: If don't care areas are required, the corresponding pixels must hold a value below the **DontCareThreshold**. In some lucky cases, when all the object background must be ignored, merely adjusting the **DontCareThreshold** to the right thresholding value can do. Otherwise, when the don't care area is in no way related to the threshold pattern image, the **DontCareThreshold** should be set to **1** and all pixels belonging to the don't care area should be set to black (value **0**).
- **MinReducedArea**: To achieve acceptable time performance, EasyMatch works by sub-sampling the pattern in the early phases of the processing. The **MinReducedArea** parameter tells how many pixels of the pattern image are kept, at a minimum. The smaller the value, the faster the matching process will be. Anyway, setting this parameter for a too small value can result in unreliable matches. The default value (**64**) is usually a good compromise.
- **FilteringMode**: As the filtering mode parameter, it allows to select the pre-processing type (averaging or low-pass filtering) applied to the image before the decimation. Generally, the filtering is performed using a uniform kernel. However, if the image exhibits sharp gray-level transitions, it is better to choose a low-pass kernel.



Learning a pattern

Matching Process

After the learning process, the pattern can be searched for and located for any subsequent image. This is the matching phase. It is performed using a single function call:

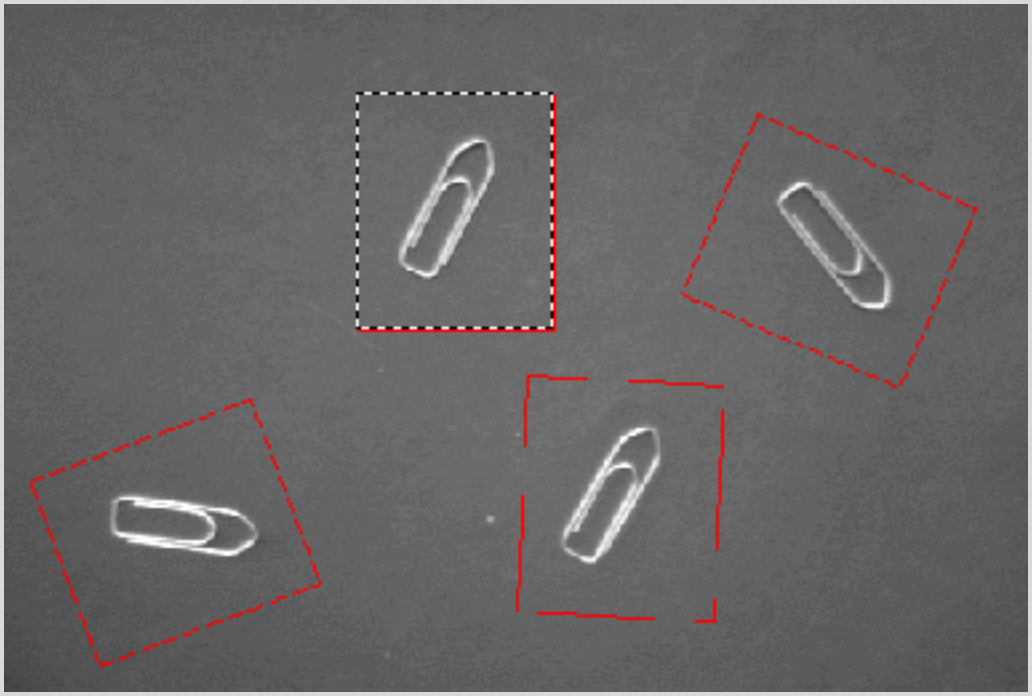
- **Match**: receives the target image/ROI as its argument and locates the desired occurrences of the pattern.

To get the result of the matching, use the following functions:

- **NumPositions** returns the number of good matches found. A good match is defined as having a score higher than prescribed value (the **MinScore** threshold value).
- **GetPosition** returns the coordinates of the N-th good match found. The positions are sorted by decreasing score.

If you want to match several patterns against the same image, create an EMatcher object for each pattern.

To reduce the processing time, matching parameters can be tuned. Refer to Tuning the Matching Parameters.



Matching a pattern

5. Tuning the Matching Parameters

The following parameters are relevant at matching time:

- Correlation mode (way to compare the pattern and the image): **CorrelationMode**.
- Contrast mode (way to deal with contrast inversions): **ContrastMode**.
- Maximum positions (expected number of matches): **MaxPositions**, **MaxInitialPositions**.
- Minimum score (score of the matches considered as bad): **MinScore**, **InitialMinScore**.
- Rotation range: **MinAngle**, **MaxAngle**.
- Scaling range: **MinScale**, **MaxScale**.
- Anisotropic scaling range: **MinScaleX**, **MaxScaleX**, **MinScaleY**, **MaxScaleY**.
- Sub-pixel accuracy: **Interpolate**.
- Low accuracy: **FinalReduction**.
- Non-square pixels: **GetPixelDimensions**, **SetPixelDimensions**.

The following conditions increase the running time:

- allowing rotations;
- allowing scaling;
- allowing anisotropic scaling;
- allowing several matches;
- setting the **MinReducedArea** parameter to a large value.

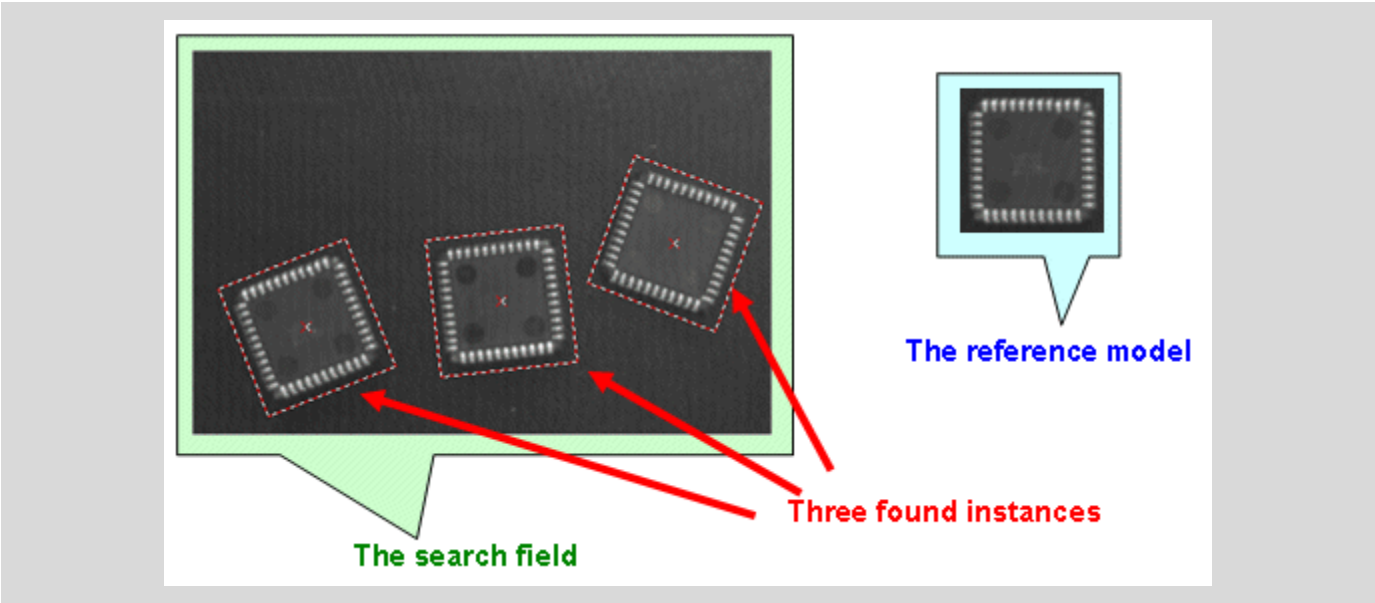
EasyFind

1. What Is EasyFind?

EasyFind is a geometric pattern matcher. As EasyMatch, it allows you to find one or more occurrences of a given pattern in a larger image. However, while EasyMatch is an area-based pattern matcher whereas. EasyFind is feature point-based.

Feature point-based carefully selects salient features in the pattern, so that it only matches the area that convey information, instead of comparing the pattern to the sample image pixel-wise, . This allows faster processing because less elements have to be handled. It also improves robustness because the matching process becomes insensitive to variations in featureless areas.

EasyFind locates rapidly **patterns** in an image, down to the sub-pixel precision. It finds **instances** similar to a **model** in a **search field**, and reports information (**score** and **location**) about these instances.



- **Search field**: the search field is an image or a part of an image expected to contain a pattern to find.
- **Model**: the model is a representation of a pattern serving as a reference for the finding function.
- **Instance**: this is a part of a search field reported as very similar to the model. "Instance" means "case" or "example".

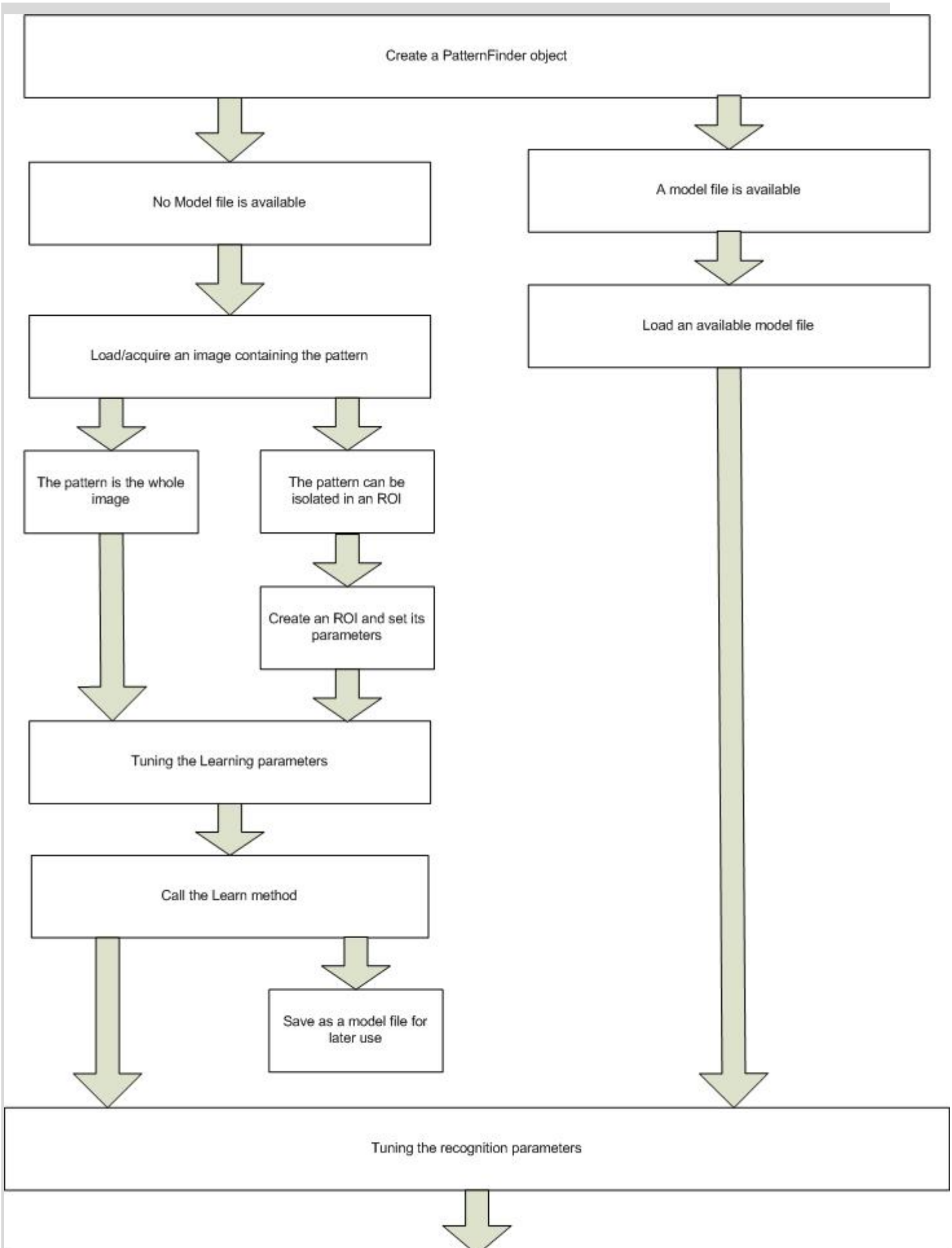
Relying more on image feature and their relative placement rather than individual pixel values, EasyFind will be particularly tolerant of noise, blur, occlusion, missing parts and changes in illumination.

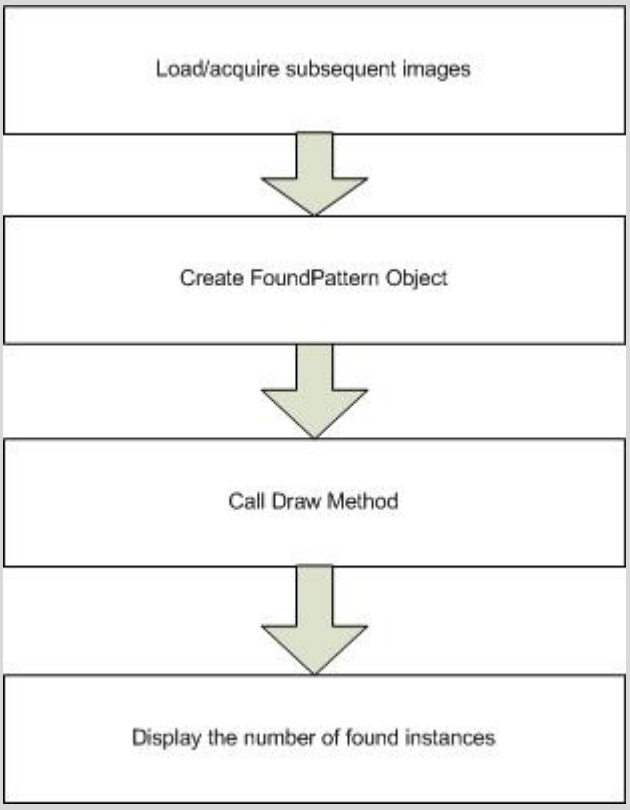
EasyFind's usage is straightforward:

- **Learning** process, where EasyFind extracts information for subsequent finding functions.
- **Finding** function in search fields using the finding function.
- **Retrieving information** about found instances.

2. EasyFind Workflow

Note that the learning process workflow shows the generic steps required to worked in EasyFind.


Create a PatternFinder object
No Model file is available
A model file is available
Load/acquire an image containing the pattern
Load an available model file
The pattern is the whole image
The pattern can be isolated in an ROI
Create an ROI and set its parameters
Tuning the Learning parameters
Call the Learn method
Save as a model file for later use
Tuning the recognition parameters


Load/acquire subsequent images
Create FoundPattern Object
Call Draw Method
Display the number of found instances

3. Learning a Model

EasyFind needs a learning phase before processing the finding function.

During this phase, EasyFind extracts feature points from the model. The type of feature points depends on the selected pattern type.

3.1 How to Choose a Good Model?

There are some advices to select the optimal model.

Consistent Edges and Thin Structure

In this case, models must be well contrasted with sharp edges.

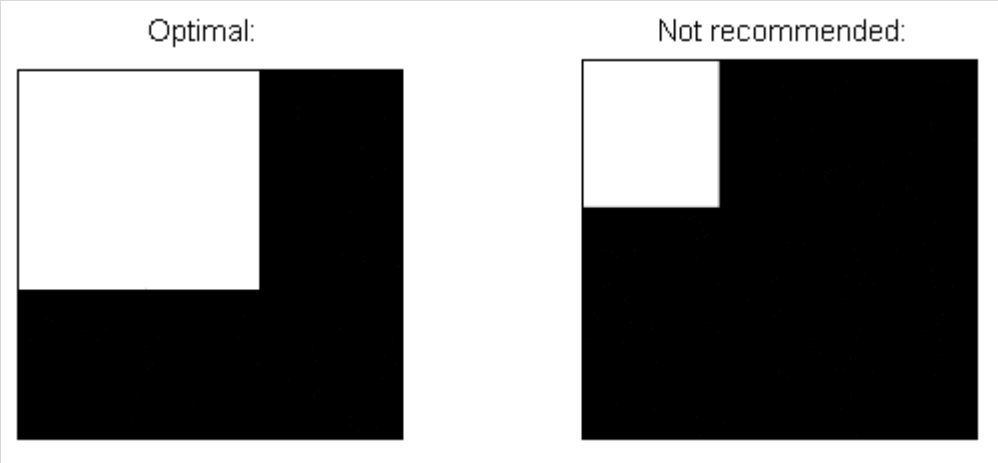
For time efficiency, models should be chosen the most different from the rest of the expected search fields.

Contrasting Regions

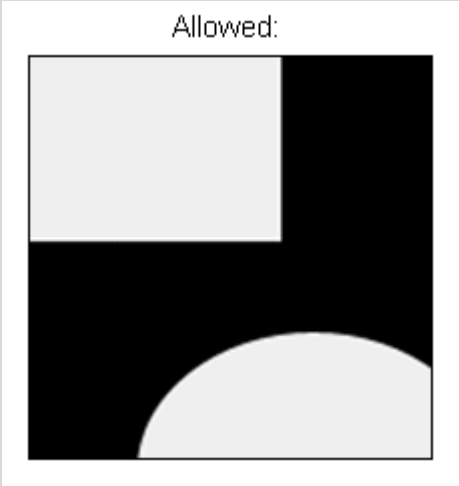
As in the other pattern types, for time efficiency, models should be chosen as the most different from the rest of the expected search field.

Since contrasting regions uses region feature points, precautions must be taken on regions to guarantee a good execution of EasyFind, in terms of time and robustness.

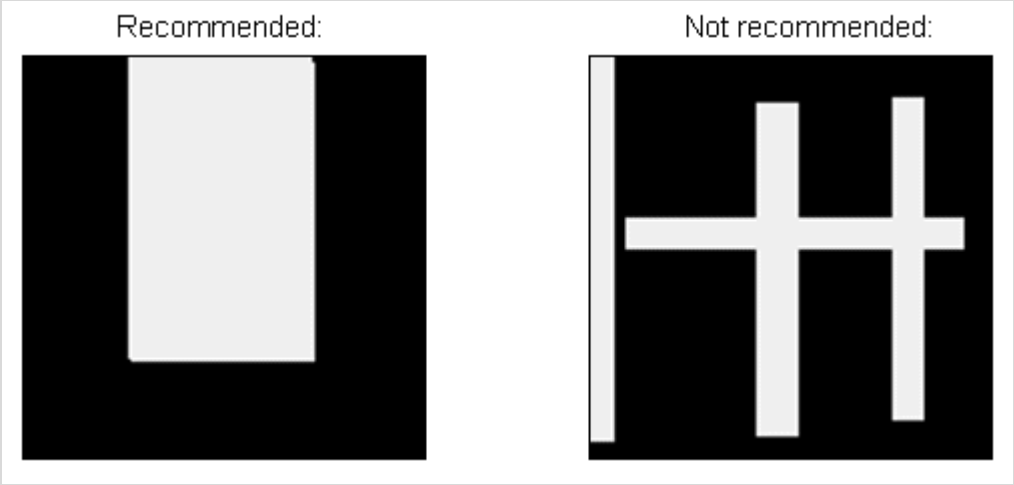
Dark and bright regions areas must be the most equilibrated. Ideally, it should be 50 % of bright area and 50 % of dark area.



There is no problem to have more than one dark and/or bright region.



Thin regions should be avoided as illustrated below.



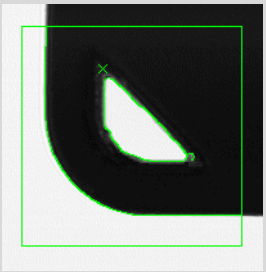
3.2 Learning Parameters

Additional parameters can be helpful for EasyFind during the learning process. Some parameters are different according to the pattern type.

General: Pivot

The pivot is the reference point in the model.

By default, this is the center of the model. The pivot will be the location returned by EasyFind when it finds an instance. This allows, for example, moving the reference point from the center to a specific place in the model like a corner or a hole.



Consistent Edges

Consistent edges learning is fully automatic. There are no parameters.

Thin Structure

In the thin structure pattern type, the contrast (dark on bright or bright on dark) between thin elements and their neighboring regions should be the same for each thin element. By default, this contrast is detected automatically by EasyFind during the learning phase.

It is not always possible to have all thin elements with the same contrast. In that case, it can be helpful to indicate the contrast of the thin elements that EasyFind favors. This is the purpose of the **thin structure modes**:

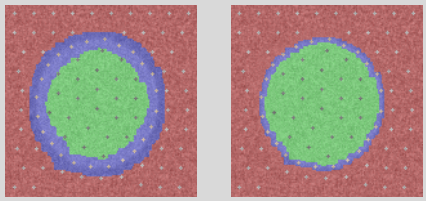
- Automatic
- Thin elements darker than the neighborhood
- Thin elements brighter than the neighborhood

Contrasting Regions: Transition Thickness

Contrasting regions use transition feature points. Transition feature points lie in an area called the transition band (highlighted in blue in the examples below). This band represents the neighborhood of measurements of the feature points.

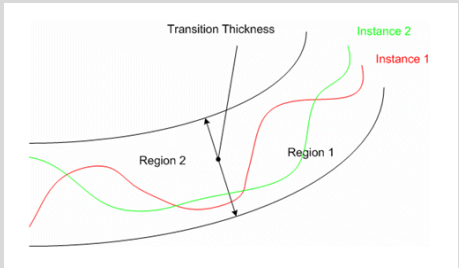
When transitions between regions is unusually smooth or unusually irregular, it may happen that the transition thickness can not cover all the transitions. In such a situation, it may be helpful to modify the size of this neighborhood.

To do so, the thickness of the transition band must be changed with the transition thickness parameter.



Two examples of thickness

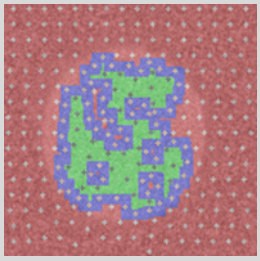
The transition thickness should be chosen in such a way that the variation of the borders between regions of different instances remains included in the transition band. Here is an illustration.



In practice, it is recommended to choose the transition thickness equal to the biggest variation among instances.

Contrasting Regions: Light Balancing

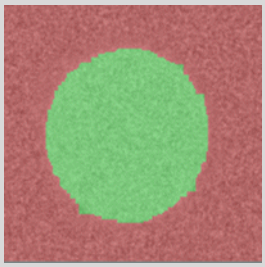
To use region feature points, EasyFind must extract regions in the model. Sometimes, when conditions are poor, EasyFind fails to extract regions correctly, as shown on the following example:



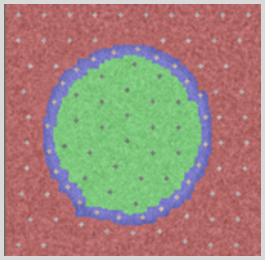
When this occurs, EasyFind requires interaction with an operator to adapt the light balance. The light balance allows EasyFind to brighten or darken regions.

During light balancing, model drawing provides a preview of the model.

In our example, the balance had to be raised. The obtained preview is:



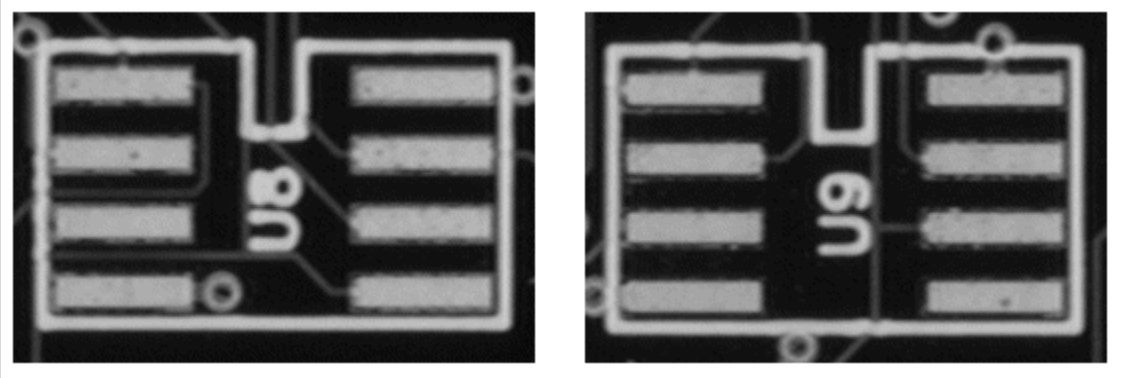
Once the light is balanced, EasyFind has to learn the model again. Once done, drawing the model shows the following:



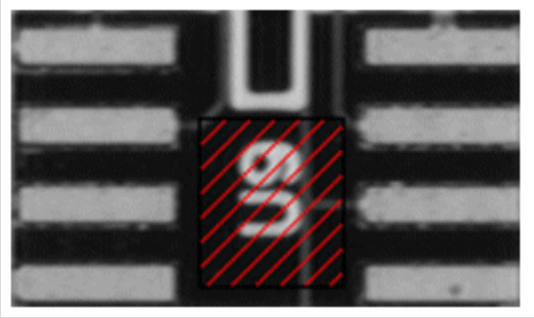
As an alternative to light balancing, the operator may indicate a gray-level threshold. It helps EasyFind extracting region feature points. Note that the forced threshold overrides the light balancing. Once the threshold is forced, EasyFind has to learn the model again.

3.3 Don't Care Area

During the learning process, EasyFind extracts feature points from meaningful characteristics of the model. It may happen that some characteristics are undesirable because they can radically change from instance to instance. Here is an example:



The text appearing at the center of the model is different between the instances and the model. So, it is possible to indicate parts of the model where EasyFind must not extract feature points.



Practically, don't care areas are indicated with a mask (image) of the same size of the model. Zero values indicate don't care areas and 255 values indicate areas taken into account.

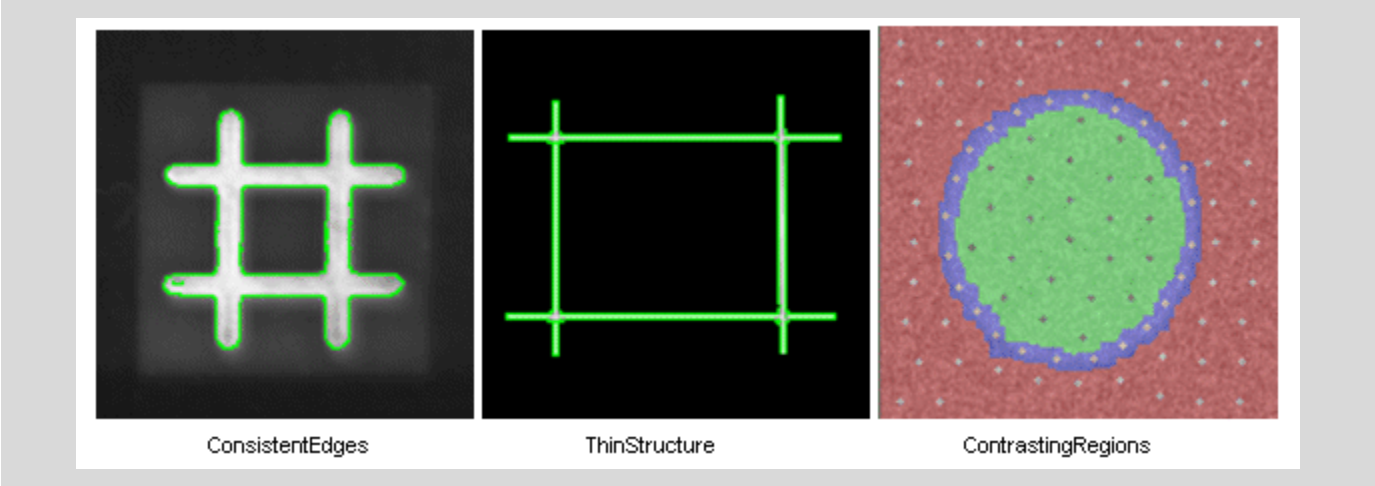
Note. Don't care areas are not taken into account when the pattern type is contrasting regions.

3.4 Learning Results

After the learning phase, it can be helpful to visualize the success of the operation.

EasyFind can draw the extracted feature points on the model. See the DrawModel method of the PatternFinder object.

On the examples, edge feature points appear as green points for ConsistentEdges and ThinStructure and appear as crosses for ContrastingRegions.



4. Feature Points

EasyFind uses meaningful characteristics of the model to find instances in the search field. Characteristics are evaluated by gauges called *feature points*. Feature points are measurements probes. They report specific measurements at their location in the search field.

A feature point is defined by its location in the model as a pair of coordinates (X, Y) and by its type. There are three types of feature points:

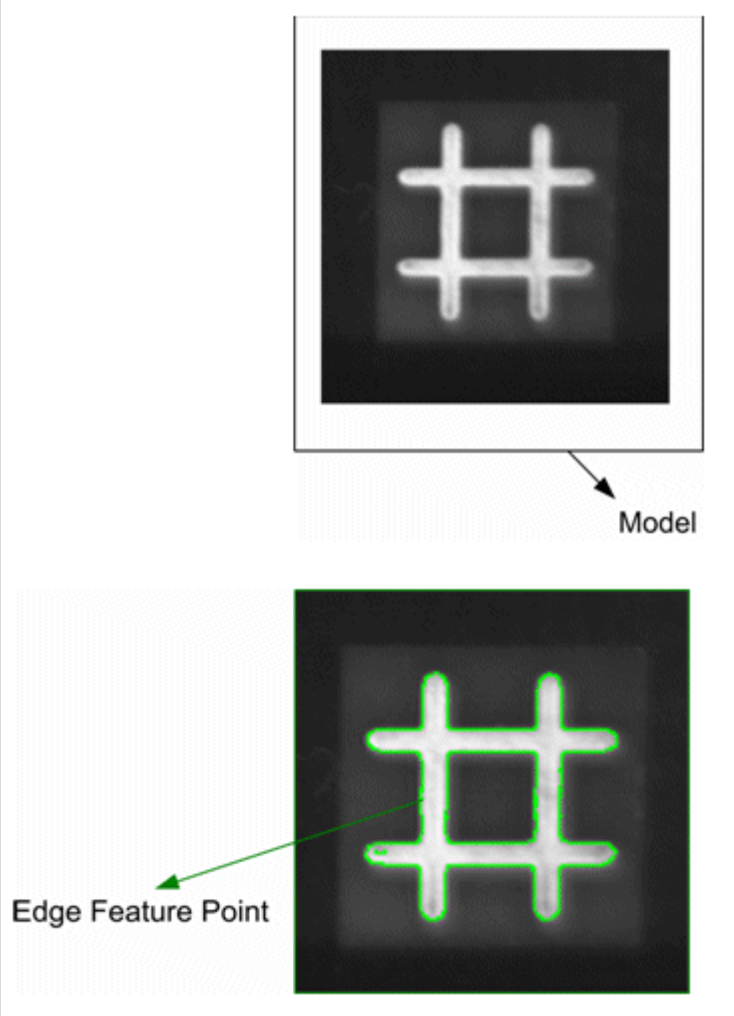
- **Edge feature points**: abrupt change of gray level between two regions.
- **Transition feature points**: smooth change of gray level between two regions.
- **Region feature points**: part of an image with roughly uniform gray level.

Feature points are computed during the learning process.

4.1 Edge Feature Points

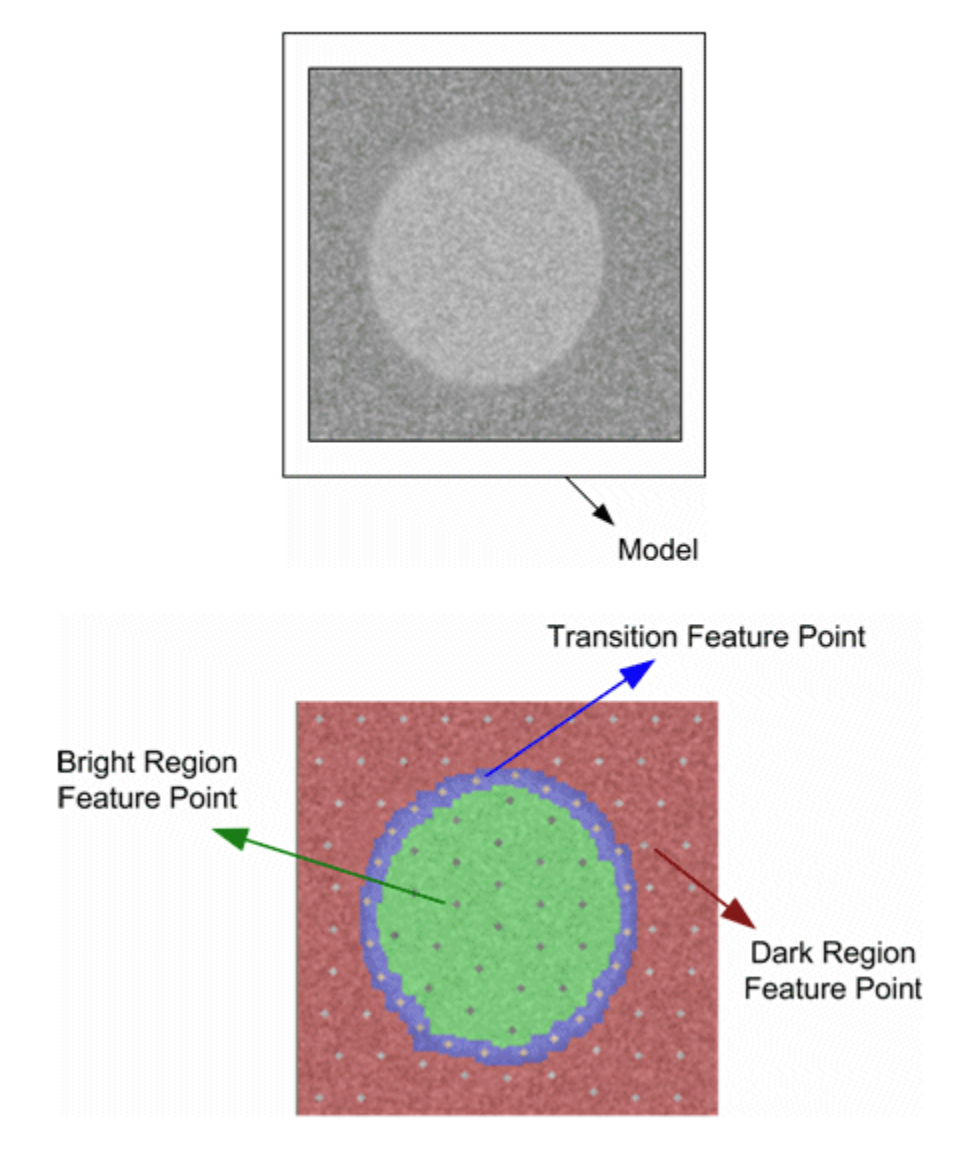
Edge feature points are located on edges in the model. They measure accurately the presence of an edge at their location in the search fields.

Here is an example of a model and its edge feature points.



4.2 Transition Feature Points

Transition feature points are located on transitions in the model. They measure the presence of a transition in the neighborhood of their location in the search fields. The size of the neighborhood can be modified. Transition feature points are represented by crosses in the blue area of the following example.



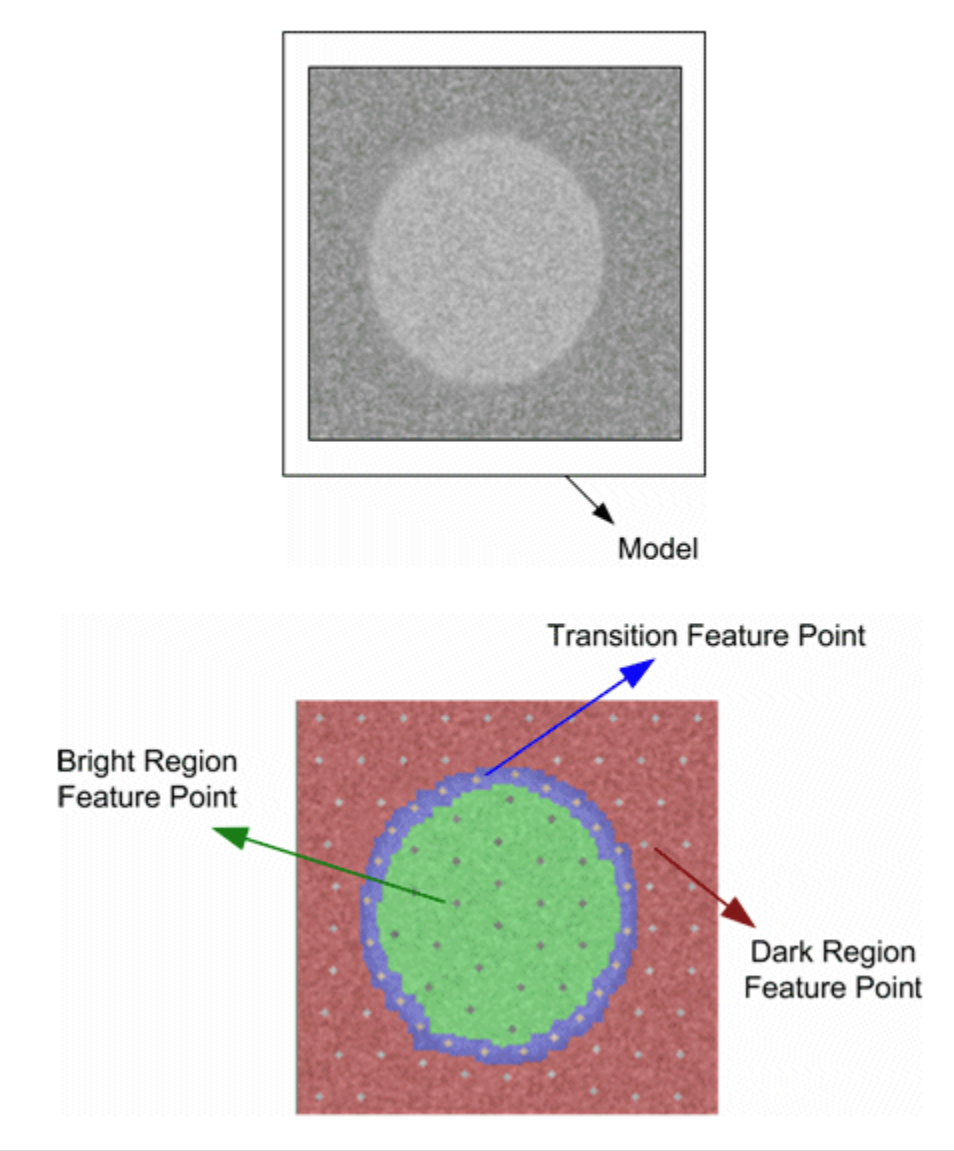
4.3 Region Feature Points

Region feature points are scattered in regions in the model. They measure the presence of a region in the neighborhood of their location in the search fields.

There are two families of region feature points:

- the dark regions feature points (represented by crosses in the red area of the example below),
- the bright regions feature points (represented by crosses in the green area).

Each region feature point of a family must characterize the same region.



5. Pattern Types

In the infinite variety of possible patterns, EasyFind operates for specified categories of them. EasyFind characterizes addressed types of patterns with meaningful characteristics.

Meaningful characteristics are particularities of the pattern used by the finding function. EasyFind distinguishes three pattern types:

- **Consistent edges**
- **Thin structure**
- **Contrasting regions**

5.1 Consistent Edges

Consistent edges applies when following conditions are met:

- The model exhibits sharp contrast transitions. It holds regions delineated by well defined edges.
- The edges are approximately at the same place for each instance in all search fields. There is a fair amount of consistency regarding the edges among the image analyzed by EasyFind.

EasyFind uses edge feature points for this pattern type.

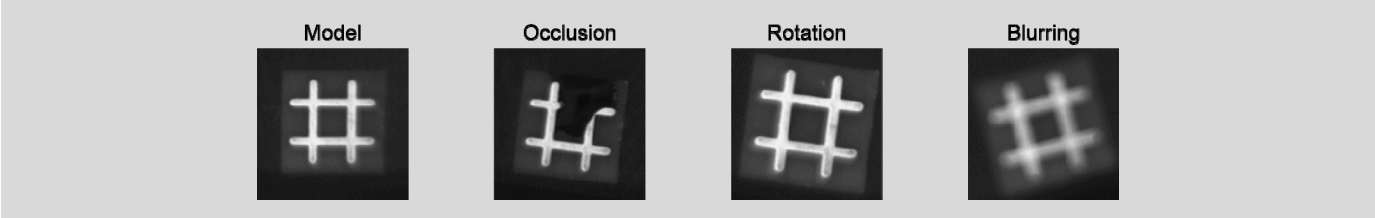
Allowances

For consistent edges, instances can be scaled or rotated.

Robustness

EasyFind for consistent edges is robust against:

- blurring,
- noise,
- occlusion,
- illumination variation.



Model and instances for consistent edges

Scoring method for the Consistent Edges operating mode

The EasyFind library provides a method to establish the score of an instance in the "Consistent Edges" operating mode.

The method relies on a comparison of the similarity of the feature points taken independently, rather than on a global evaluation that considers all the feature points at the same time. Such a point-by-point (i.e. local) scoring method is more resilient to large occlusions and/or large variations of contrast, and can also reduce the computation time of the finding phase.

The scoring method is selectable through the EPatternFinder::ContrastMode property. Three new values are available: PointByPointNormal, PointbyPointInverse, and PointByPointAny. Selecting one of these new values enables the following point-by-point score evaluation method:

- **PointByPointNormal**: matches a feature point from the model with a feature point from the search field if they share the same contrast polarity.
- **PointByPointInverse**: matches a feature point from the model with a feature point from the search field if they exhibit opposite contrast polarity.
- **PointByPointAny**: matches a feature point from the model with a feature point from the search field regardless their respective contrast polarity.

5.2 Thin Structure

Thin structure is suitable for models containing thin elements.

A thin element is a region so thin that its uniform part is almost undistinguishable. Borders between thin elements and regions must be edges and consistent in the instances. The contrast (dark on bright or bright on dark) between thin elements and their neighboring regions should also be the same for each thin element.

Thin structure uses only edge feature points.

Allowances

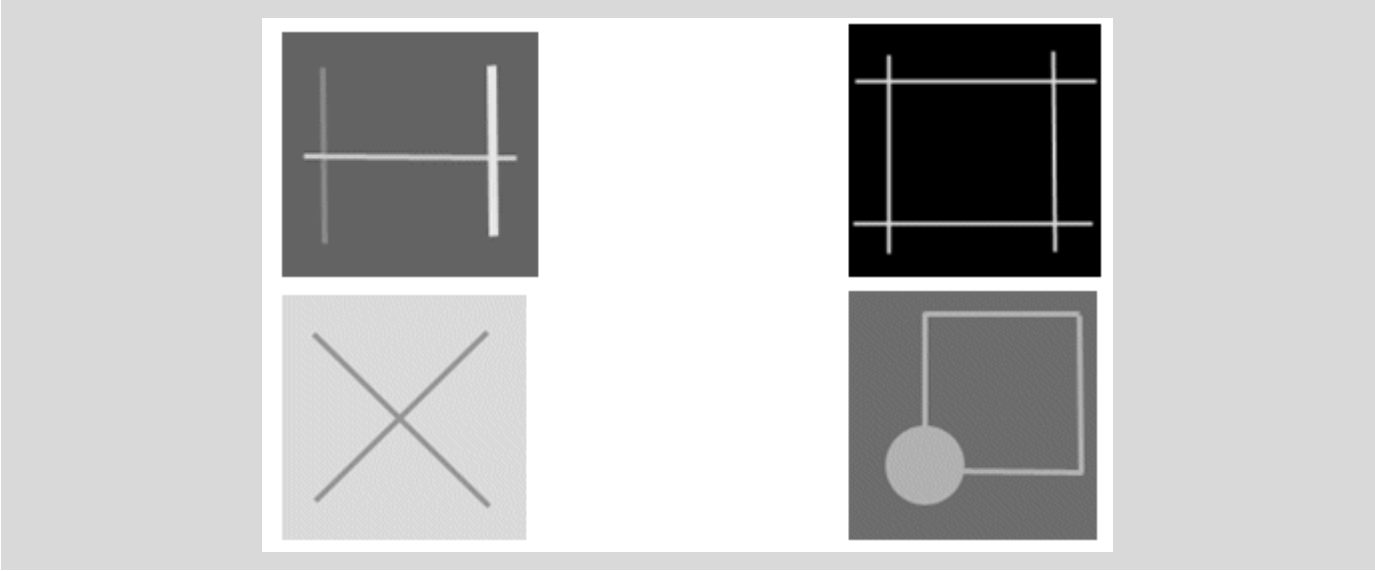
For thin structure pattern type, instances can be scaled or rotated.

Robustness

EasyFind for thin structure pattern type is robust against:

- blurring,
- noise,
- occlusion,

- illumination variation.



Representative models for thin structure pattern type

5.3 Contrasting Regions

Contrasting regions suits to models with following characteristics:

- The instances exhibit transitions or edges that are not consistent in shape regarding to the model.
- The model is composed of several large regions delimited by transitions or edges.

Contrasting regions uses both region feature points and transition feature points.

Allowances

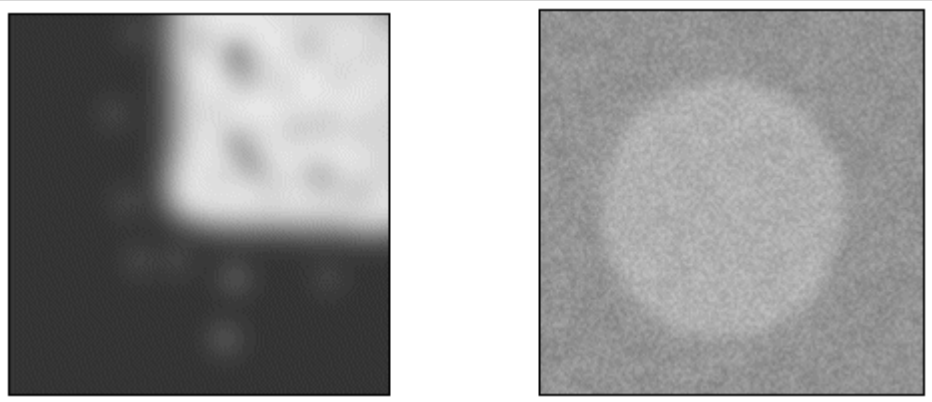
Contrasting regions does not support scaled or rotated instances. This is because models for contrasting regions are most of the time scale-ambiguous or rotation-ambiguous.

Robustness

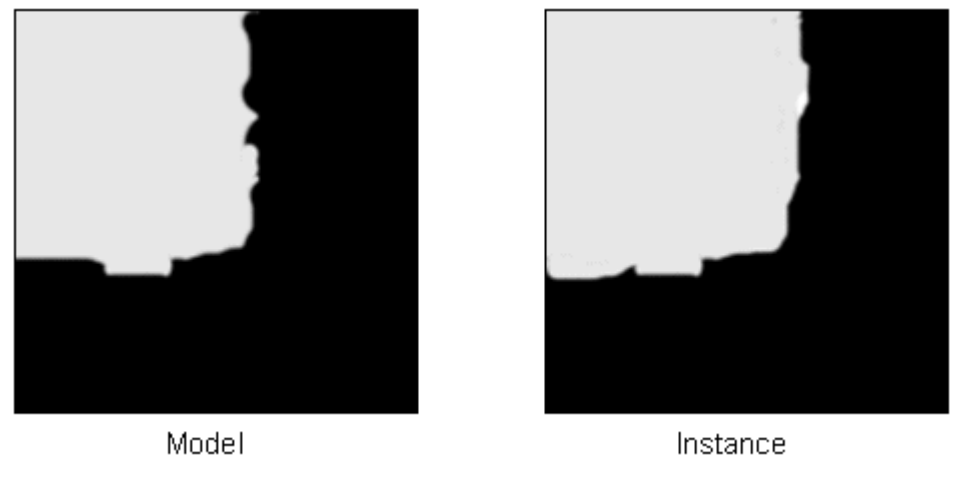
EasyFind for contrasting regions is robust to:

- blurring,
- noise,

- illumination variation.



Two models for contrasting regions exhibiting transitions

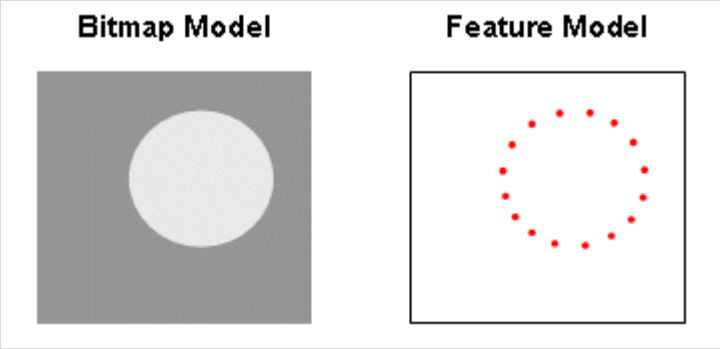


**A model exhibiting edges and an instance which is not consistent in shape.
This is a contrasting regions model.**

6. Feature Model

The model learnt by EasyFind is a bitmap representation of the pattern.

During the learning process, EasyFind computes for itself a feature model which is a set of all extracted feature points.



With this feature model and no other information, EasyFind is able to execute the finding function. Consequently, EasyFind is significantly robust against occlusion, noise, blur and illumination variations in featureless areas.

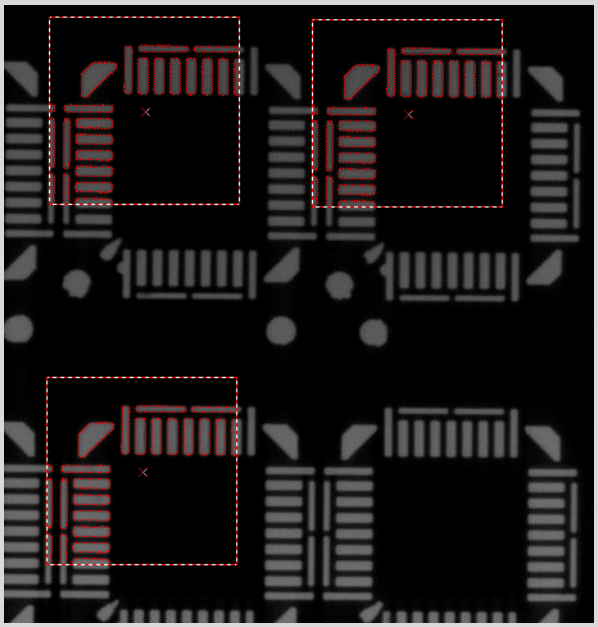
7. Finding in a Search Field

Generally speaking, EasyFind is completely ready after the learning process and can directly perform the finding function. However, some parameters can be set for the finding function and the found instances.

The Maximum Number of Expected Instances

This is the maximum number of returned instances by the finding function.

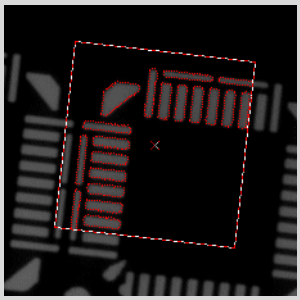
Here is an example where three instances were requested to EasyFind at the most.



The Angle and Scale Range of Found Instances

Ranges are described with a bias and a tolerance. For instance, for an angle bias of 20° and an angle tolerance of 5°, EasyFind will return instances with an angle between 15° and 25° with respect to the learnt model (20° ± 5°).

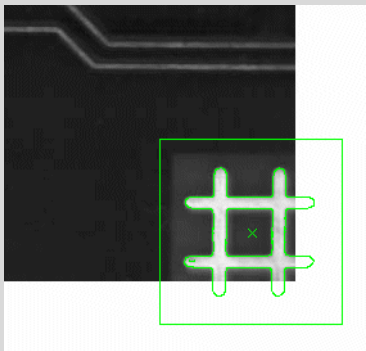
Remember that this is only available for consistent edges and thin structure pattern types.



The Extension of the Search Field

Defined in pixels, the extension of the search field allows instances to be partly out of the search field (only for thin structure and consistent edges pattern types).

Here is an example where the extension of the search field was different of 0 and where EasyFind locates an instance partially out of the search field.



8. Saving/Loading an EasyFind Model

Once all learning and finding parameters are correctly set, EasyFind allows saving this configuration in a file.

This relieves from the need to set all parameters each time a new EasyFind object is created. The configuration file must simply be loaded.

9. Reported Information

When EasyFind locates instances in a search field, it delivers information about these instances.

Location

The location is the place where the instance has been found. It is expressed as a *pair of coordinates* (X, Y) referred to the search field in image coordinates. Coordinates are generally the center of the instance but it can be another point in the model when requested.

Angle

EasyFind finds instances that have a different orientation than the model. The angle reports the *orientation difference*.

Scale

Instances can differ from the model in size. The scale reports the *size ratio*. For instance, **1.25**-valued scale means that the instance is 25 % bigger than the model.

Score

The score reports how closely the found instance looks like the model regarding its meaningful characteristics. A **1**-valued score means that meaningful characteristics of the instance are *identical* in the model. A smaller-than-one value indicates some deviations between meaningful characteristics in the instance and in the model.

EasyGauge

1. What Is EasyGauge?

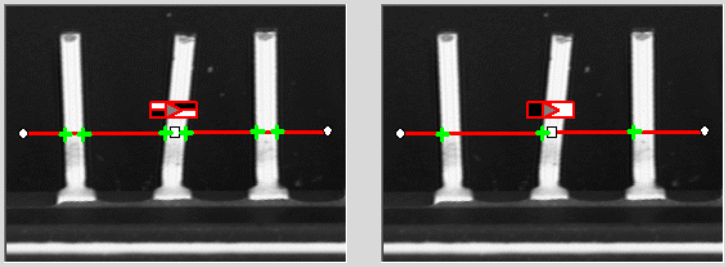
EasyGauge is a second-generation measurement and dimension control library for use in gauging applications. By relying on proven sub-pixel edge detection and least squares fitting algorithms, it allows determining the position, orientation, curvature, size... of manufactured parts with an excellent accuracy. Robustness is ensured by powerful edge-point selection mechanisms that are intuitive and easy to tune, allowing measurement in cluttered images. In addition to these state-of-the-art features, **EasyGauge** also supports adjustment of parallel sides, thus providing means of measuring thickness of flat or bent objects, as well as precise location of corners.

EasyGauge has advanced built-in calibration capabilities able to transparently convert pixel measurements to physical units and relieves the user of the need to convert coordinates. Non-square pixels and rotated coordinate axis are supported. EasyGauge also provides means to determine and correct perspective and optical distortion, with no performance loss.

Last but not least, **EasyGauge** supports grouping of the measurement gauges and lets these groups track the measured items in the image. These can freely translate and/or rotate while the probes are re-positioned accordingly. Derived measurements such as distances between feature points can then be computed.

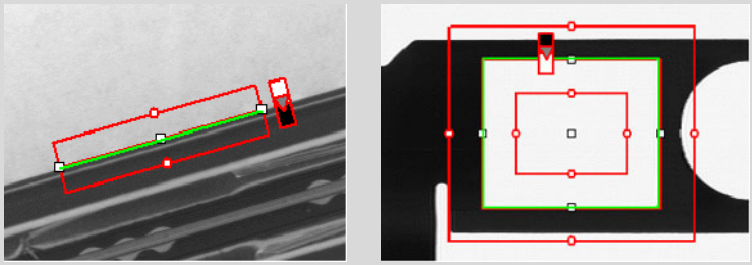
The **EasyGauge** library supports the following operations:

- **Point location**: finds the position of all transition points along a line segment probe that crosses one or several objects edges, and allows selecting the most relevant ones. Crosswise and lengthwise filtering can be activated for noise reduction.

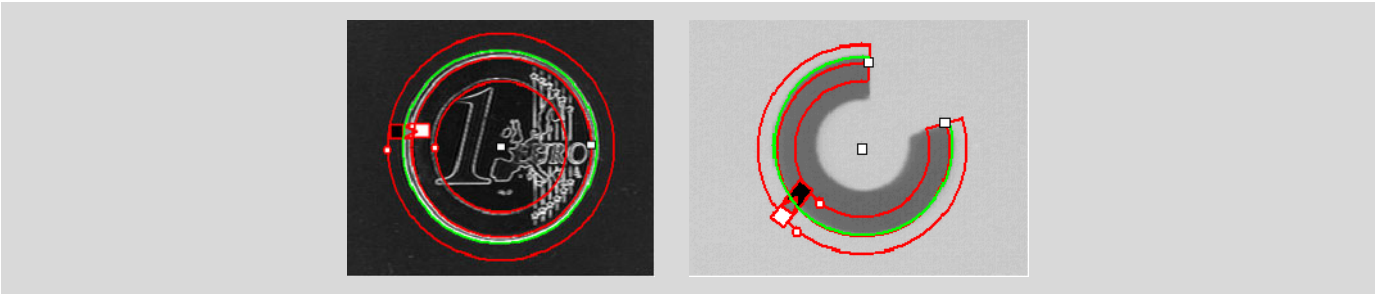


Point location. Contrast-based selection

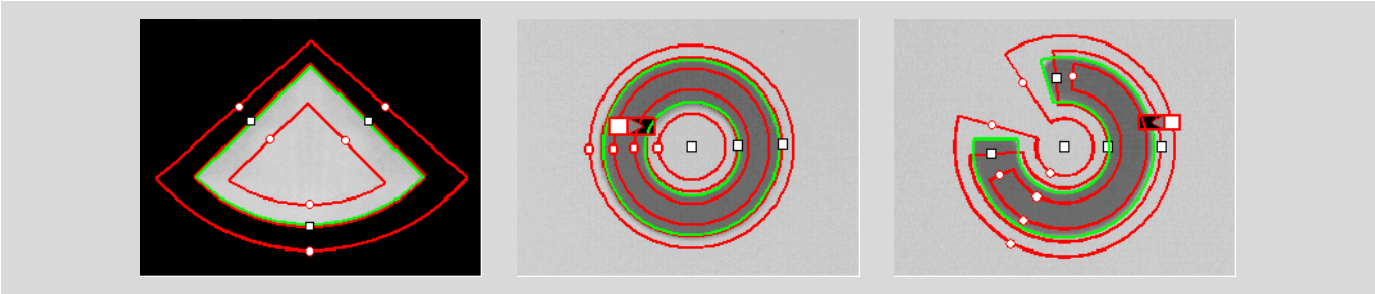
- **Edge fitting**: adjusts a simple predefined geometric model over the edges of an object. The synthetic models that can be fit are the line segment, the circle (or arc thereof), the rectangle and the wedge (disk, ring, sector or curvilinear quadrilateral).



Line and rectangle fitting

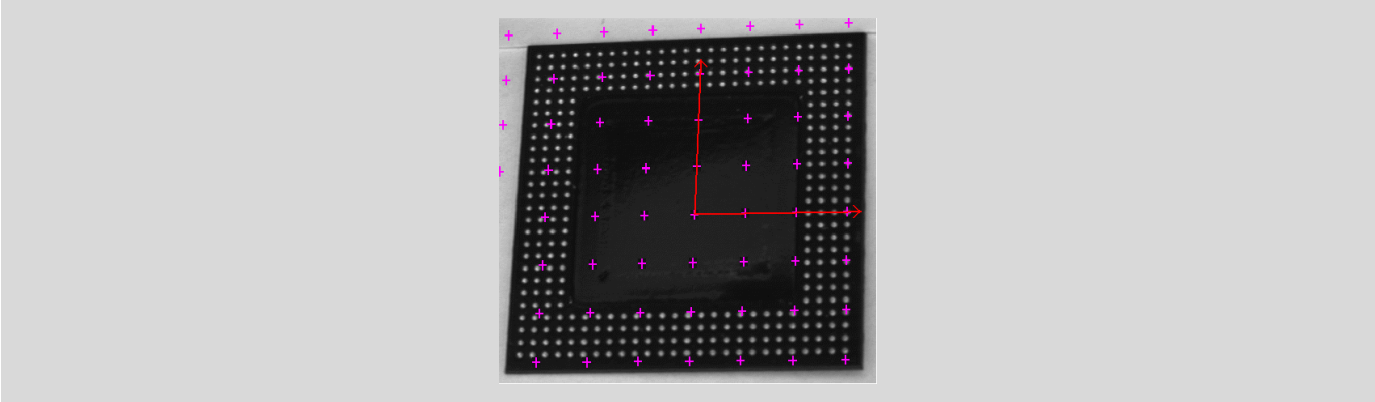


Circle and arc fitting



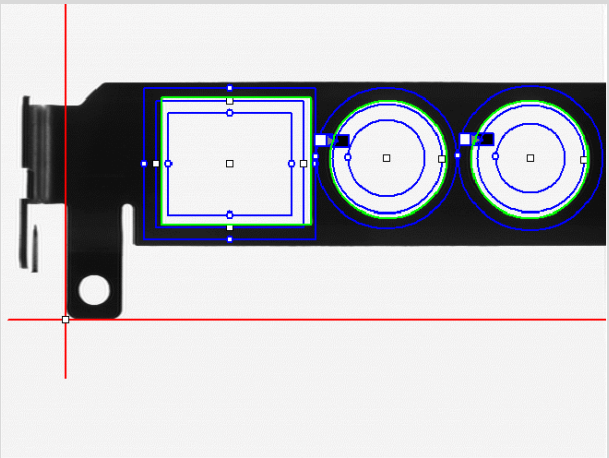
Wedge fitting (sector, ring and ring sector)

- **Field-of-view calibration**: establishes the relationship between point coordinates in the real world and pixels in the image. The calibration transform is handled by a specific device named EWorldShape (C++) or EWorldShape (ActiveX), which provides means to compute the appropriate calibration coefficients (Refer to section EWorldShape for more information). The transform is then implicitly applied to all measurement gauges tied to it.



Field-of-view calibration

- **Gauge grouping**: the measurement gauges can be tied together so that their relative placement remains fixed, and they can be moved (translated and rotated) as a whole. This way, the placement of the gauges can easily follow the movement of the inspected items. Grouping gauges is achieved by attaching them to a common coordinate frame or to another gauge, allowing to automatically track reference edges.



Gauge grouping for object tracking

- **Model edition:** EasyGauge provides means to graphically interact with the gauges to place and size them, combine them as a hierarchy of grouped items, and store/retrieve them and all working parameters to/from model files.

Together, these basic operations virtually allow obtaining any geometric measurement on a part. The calibration transform allows describing the inspected items independently of the viewing conditions such as observation distance and direction, lens focal length, optical aberrations, ... The gauge grouping mechanism allows to completely describe the gauging task in a single model file ready for use.

EasyGauge was designed in such a way that the gauging model can be built separately by means of a graphical editor, and then "played" in the final application. Programming an inspection application is simple and straightforward.

Simple Gauging

Even tough **EasyGauge** provides advanced and sophisticated mechanisms for accurate and combined measurements, the basic use is straightforward: according to the measurement to perform, create the corresponding gauge object, and set the parameters whose default values are not appropriate. Then invoke the desired measurement function and read the resulting position parameters.

Calibrated Gauging

Uncalibrated gauging is easy to implement but it has several major draw backs: first, measurements are performed in a "useless" unit (the pixel), while the parts to be inspected normally have actual sizes expressed in millimeters, miles, microns, yards and the like; second, the corresponding measurement models are not portable in the sense that all gauge positions and sizes have to be reworked if the viewing conditions change; third, if the image formation process induces errors such as optical distortion or perspective, the resulting measurement will be biased.

Please refer to Calibration to learn how to master field-of-view calibration.

Once a calibrator object is available and its calibration parameters have been adjusted, it suffices to attach a gauge to it and all measurements will immediately be performed in the calibrated units, with distortion, if any, implicitly compensated!

Complex Gauging

In simple cases, a single measurement gauge can suffice. For instance, checking the diameter of a hole requires a single circle gauge. Anyway, in other situations, more than one measurement site has to be considered (several holes, overall size and orthogonality of sides to be checked too...). Placing all of them on the inspected scene, adjusting all their working parameters can become a tedious task.

EasyGauge addresses this issue by allowing to save a complete model, including the calibration modes and coefficients, and the various gauges attached to it, in a single file.

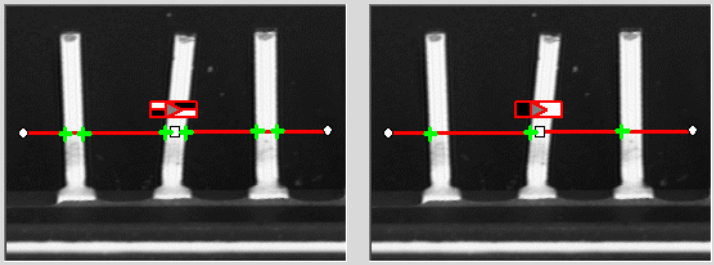
In addition, several gauges can be made integral with each other by attaching them to another item, with the following consequences:

- **Attaching the gauges to an EFrameShape object** fastens them together so that moving the frame (translation and/or rotation) will cause all gauges to move accordingly. In this case, it is the task of the application program to adjust the frame position to track the inspected part.
- **Attaching the gauges to another gauge** will make them move according to the measured position of the supporting gauge. By example, if gauges are attached to a common rectangle gauge, and if the rectangle gauge is detecting the outline of a part to inspect, all gauges will automatically track the part when the rectangle outline is fitted.

2. Point Location

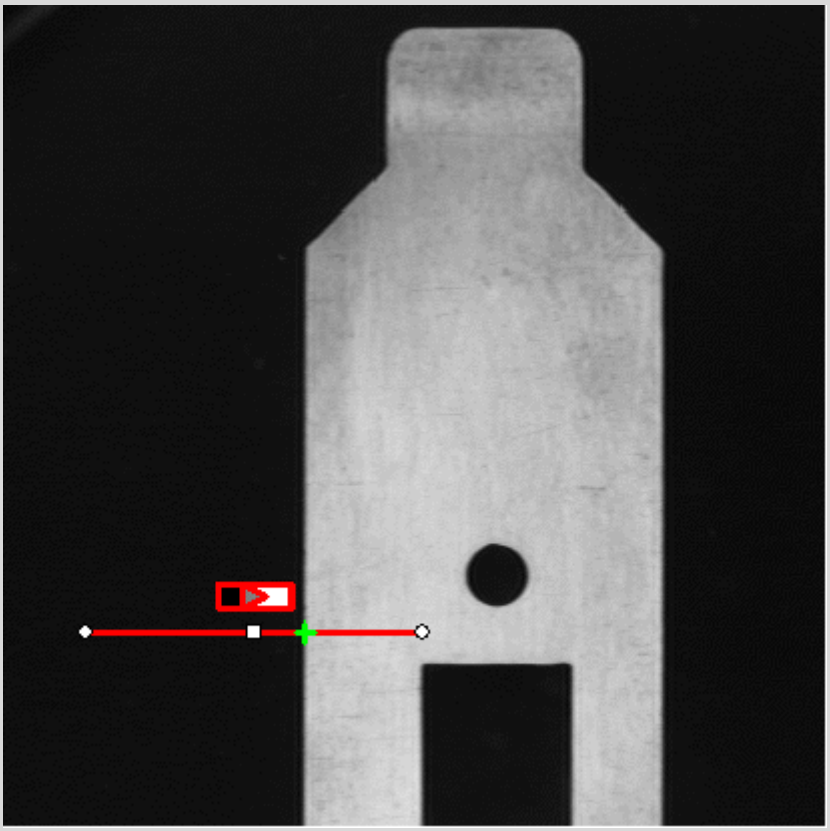
2.1 What Is Point Location?

Point location: finds the position of all transition points along a line segment probe that crosses one or several objects edges, and allows selecting the most relevant ones. Crosswise and lengthwise filtering can be activated for noise reduction.

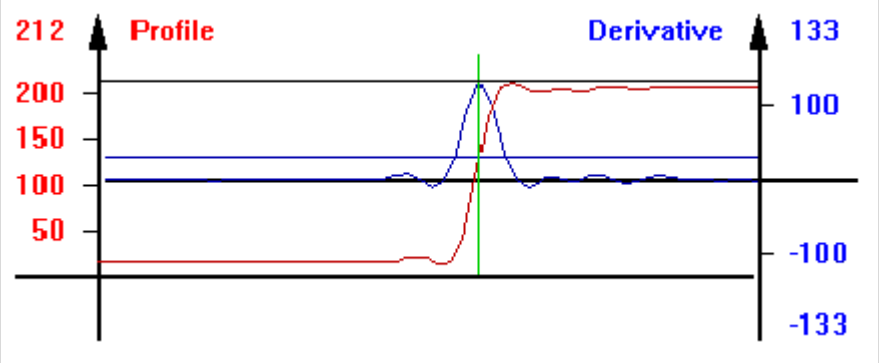


Point location. Contrast-based selection

When one traverses a linear profile extracted from an image, along a line segment, an edge appears as a transition from a dark zone to a light zone (or vice versa). When plotting the pixel values along the gauge, this transition appears as a S-shaped curve. The first derivative of this curve exhibits a peak around the transition point. The better the contrast, the sharper the transition and the higher the peak are.



Point location principle



S-shaped curve and its derivative

EasyGauge extracts the pixel values along a profile (red curve) and then uses peak analysis to determine the transition location. A peak is the area comprised between the derivative curve and a horizontal user-defined threshold level. All the pixel values in the peak area are used to compute the transition location. This is how EasyGauge achieves sub-pixel accuracy.

Notes.

- Sub-pixel accuracy may only be reached if the transition is surrounded by almost uniform regions. These regions have to be 2 pixels wide at least.
- In the case of a transition from BWB, the profile curve is increasing and the peak takes positive values. Otherwise, the curve decreases and the peak extends negatively.
- In the case of BWB or WBW transition type, the peak analysis is based on the gray level profile along the EPointGauge (or sample path) and not its first derivative. Therefore, you are not able to detect peaks using the default threshold value (20) for most of the time.

All useful point measurement parameters are kept in an EPointGauge object. By default, all parameters take values that are suitable to detect reasonably contrasted edges.

The nominal point position is adjusted by means of the Center property. The tolerance value and gauge orientations are set by means of Tolerance.

The peak selection strategies are defined by means of methods TransitionType, TransitionChoice, possibly in association with TransitionIndex.

The noise immunity is adjusted by Threshold, and suitable peak strength is tuned by means of MinAmplitude or MinArea.

The local filter widths can be adjusted with Thickness and Smoothing.

The sampling area is rectangular by default. Previously, it was only possible to achieve the transverse filtering in a non rectangular sampling area. The transverse filtering mode can be selected by calling the RectangularSamplingArea method.

The actual measurement is controlled by method Measure, passing it a pointer to the source image.

In case of a single transition mode, Valid returns TRUE when an appropriate point was found. To obtain the measurement results, set the ActualShape property to TRUE. The ActualShape mode determines whether Center returns the located point (TRUE value) or the nominal point position (FALSE value, default).

In case of a multiple transition mode, NumMeasuredPoints provides the actual number of points found.

Alternatively, GetMeasuredPoint provides the results as an EPoint object. Information pertaining to the located point is then retrieved through this EPoint object. In case of a multiple transition mode, an integer index between 0 and GetNumMeasuredPoints-1, inclusive) must be passed.

The overloaded **Distance** member opens the way to more advanced measurements. It provides a means to retrieve the distance between a point pair or between a point and a line segment, a circle arc or a rectangle.

The GetMeasuredPeak function returns an EPeak structure containing the Area, the Amplitude of the peak, as well as the delimiting coordinates along the probe segment (Start, Length and Center values).

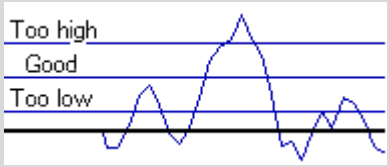
The code fragment below sets up a gauge in order to locate a point on a white item over a black background **ETransitionType_Wb**, keeping the peak the closest to the nominal point **ETransitionChoice_Closest**. This edge is inclined by -45° and the point gauge oriented perpendicular to it (+45°).

- No calibration is required, so that the gauge need not be attached to a calibrator (**EWorldShape**). A tolerance of 10 units (i.e. pixels) is allowed, corresponding to the uncertainty of the position of the edge.

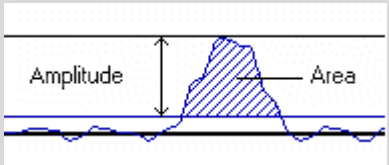
2.2 Peak Selection

Choosing an appropriate threshold level is very important. When the threshold value is too high, significant peaks can be missed and too few pixel values are used to achieve a good precision. On the other hand, setting too low a value will make false peaks appear because of noise. To resolve this dilemma, EasyGauge provides a peak selection mechanism that can reject low contrast or false edges: the strength of a transition can be measured by the peak *amplitude* (difference in ordinate between the threshold and the top of the peak) and the peak *area* (surface comprised between the threshold level and the curve).

Any time an edge is measured, the peak amplitude and area are determined. If either value falls below the desired minimum amplitude or minimum area, the peak is simply disregarded and no point is assumed at that location.

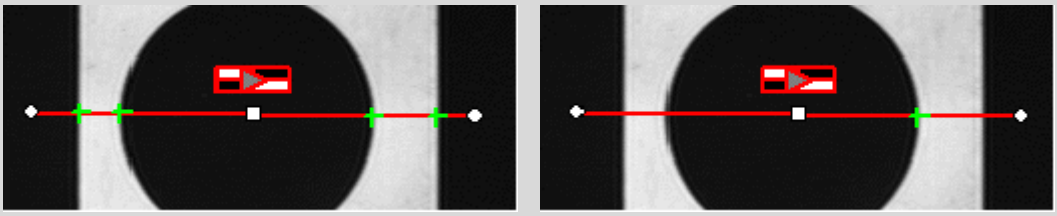


Threshold level selection



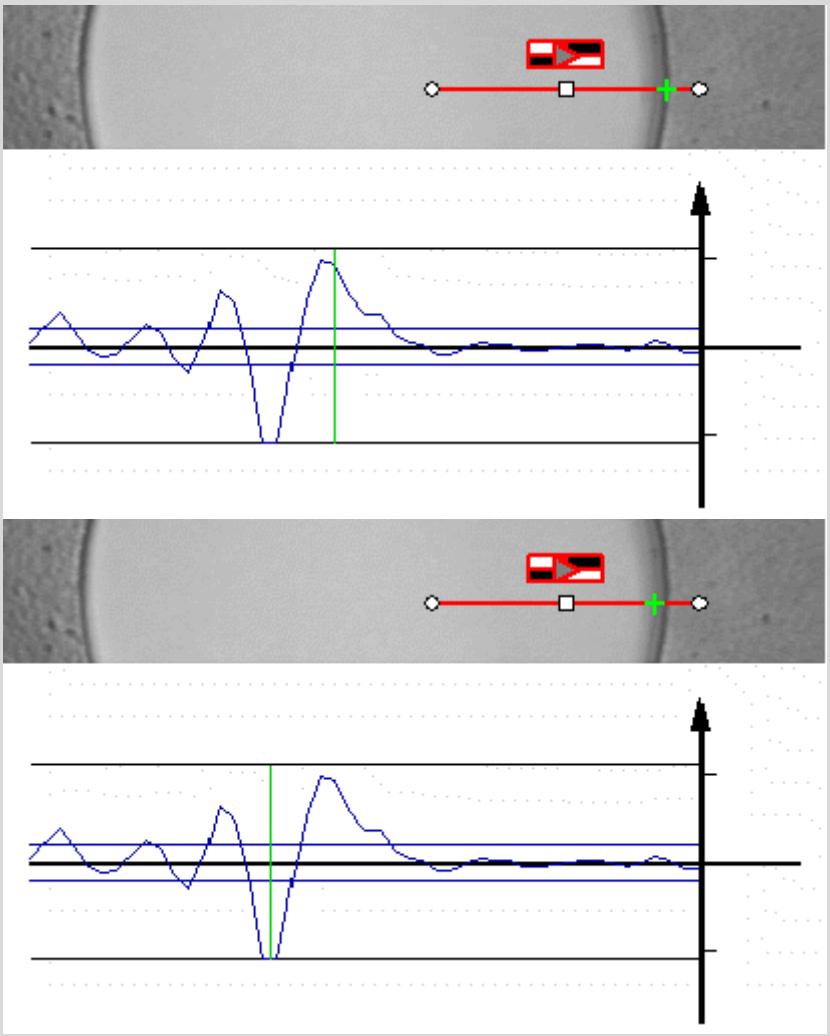
Peak amplitude and area

Quite often, several edge points are present. In some applications, all of them must be considered. EasyGauge allows measuring all points in a single go and retrieve all results afterwards. This is the *multiple transition mode*.

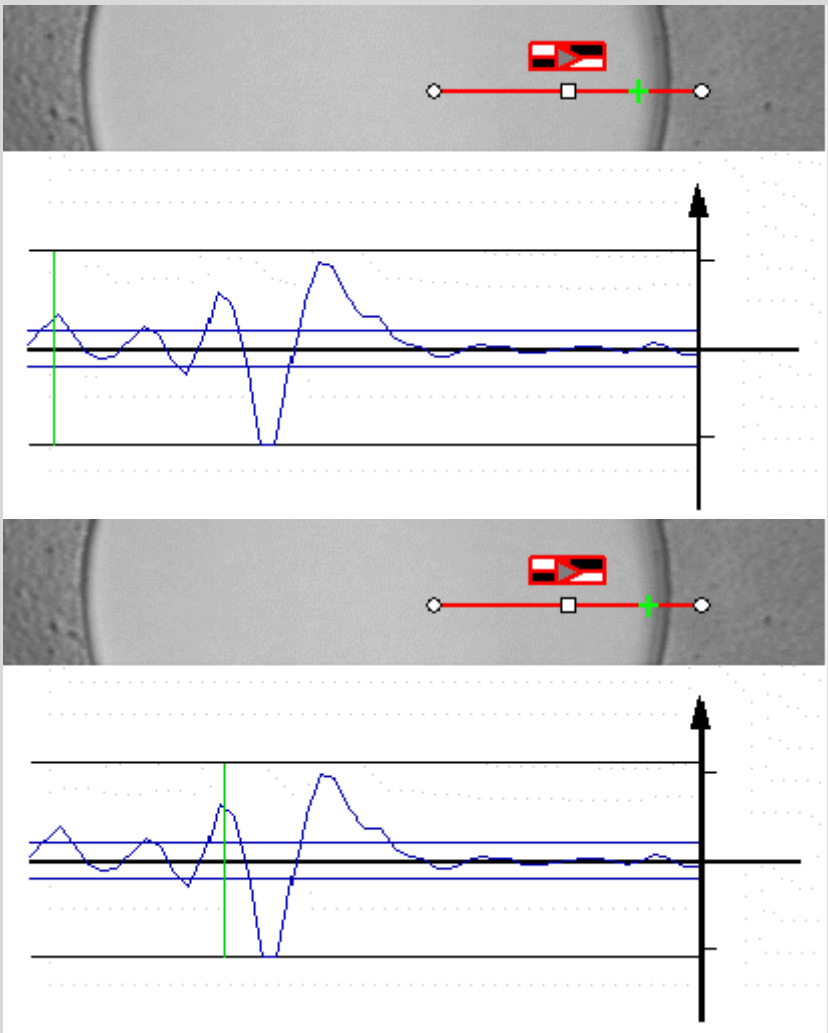


Multiple versus single transition

Alternatively, EasyGauge allows selecting the most relevant transition. Four selection rules are available for peak selection: it can be the highest one, the one with the largest area, the one closest to the gauge center or the N-th one encountered starting from one tip of the gauge.

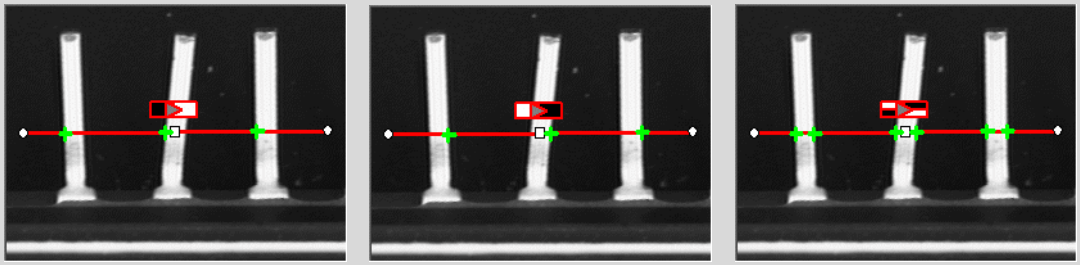


Best area and best amplitude choices



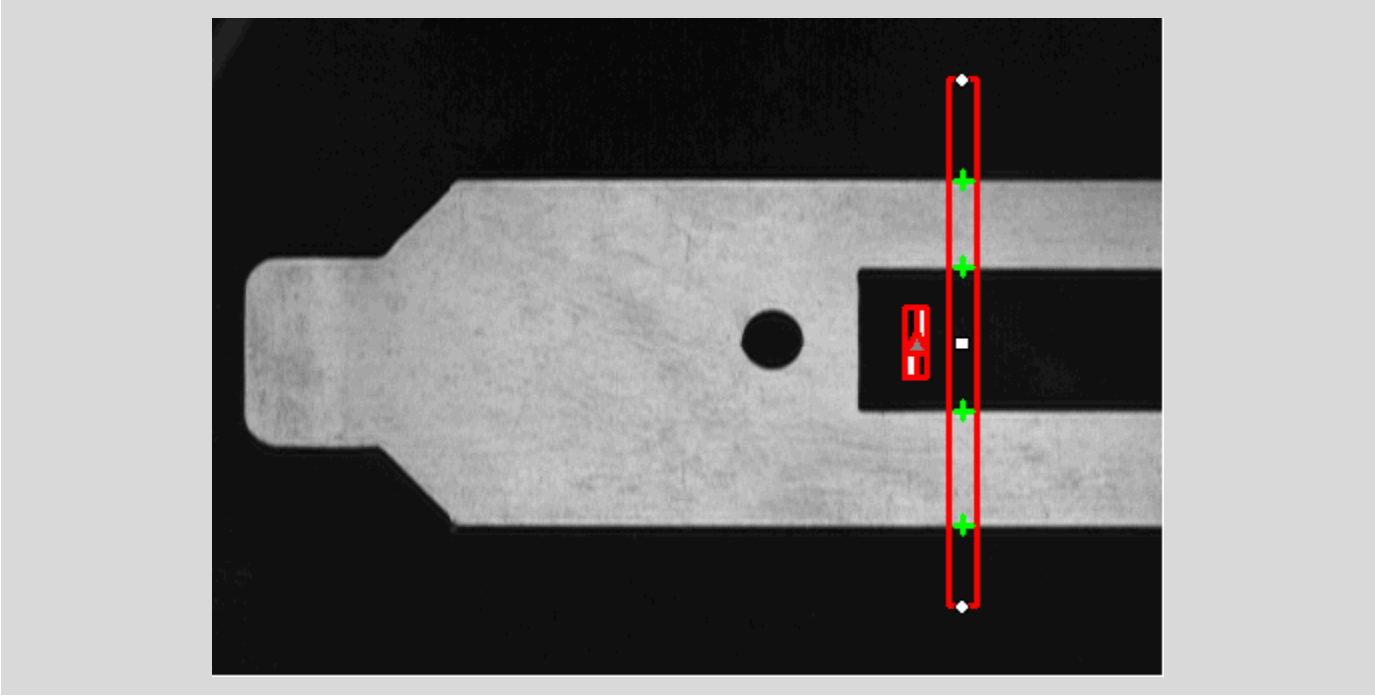
Closest and 3rd from begin choices

Peak selection can also be refined by choosing the transition polarity: raising or falling edge (i.e. positive or negative peak), or indifferent.



Black to white, white to black or indifferent polarities

To reduce the effect of noise, a local pre-filtering of the image values is possible. Transverse (lengthwise) filtering is achieved by averaging several parallel lines when sampling the image. Longitudinal (crosswise) uniform filtering can also be applied to the resulting profile curve.



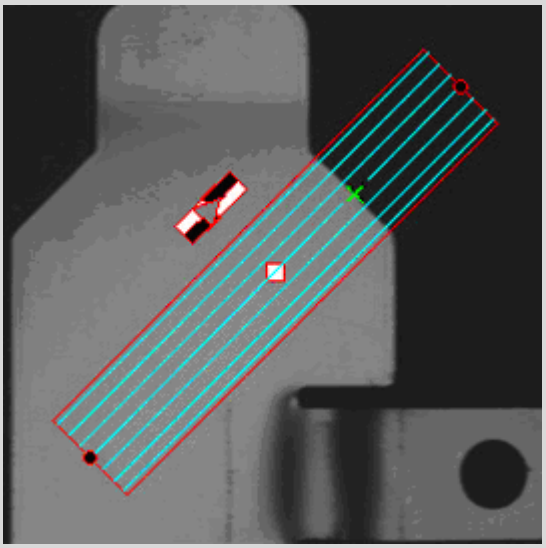
Thick point gauge for filtering

2.3 Transverse Filtering

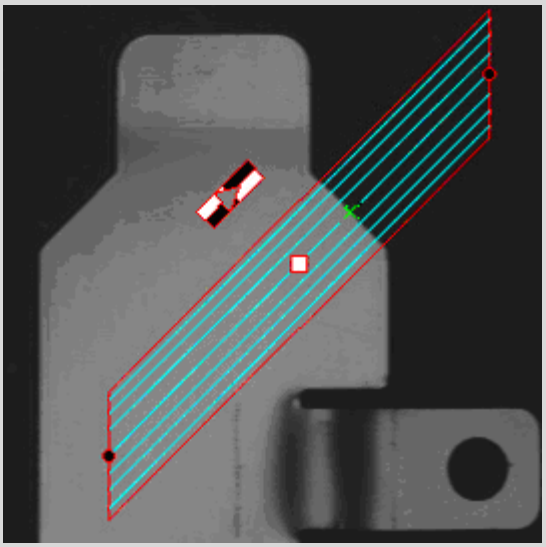
Two modes exist for the transverse filtering. The first one places the parallel line segments in a parallelogram, the second one in a rectangle. Note that the number of parallel lines depends on the thickness transition parameter. The smallest region containing all the parallel line segments is called the sampling area.

The rectangular sampling area is the default one. Previously, it was only possible to achieve the transverse filtering in a non rectangular sampling area. This behavior can be toggled.

When angle is close to 0° or 90°, or when thickness is small (less than 5), the non rectangular mode runs faster than the rectangular one without significant accuracy differences. Obviously, when the sampling area reduces to a single line segment (thickness equal to 1), no difference subsists between the two modes.



Rectangular sampling area



Non rectangular (parallelogram) sampling area

2.4 Point Probe Position

The position of a point gauge is fully specified by its **center**, its length, also called **tolerance**, and the orientation **angle** with respect to the X-axis.

This position is known as **nominal**, since it refers to the expected position of the point. By contrast, the tolerance indicates to what extent we "allow" the point position to vary.

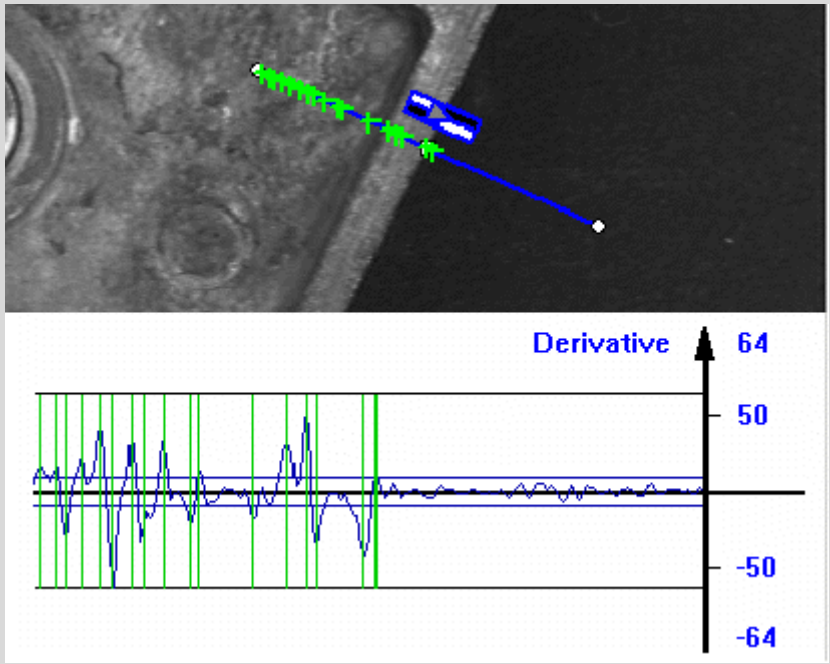
After measurement, the returned results are the coordinates of the located points. This position is known as the **actual** location. Additionally, the strength of the transition (amplitude and area) is computed. Low values indicate a weak edge, possibly corresponding to an unreliable or inaccurate measurement.

2.5 Tuning the Point Measurement Parameters

EasyGauge was designed in such a way that the default parameters and working modes are suitable for standard cases, when there is little ambiguity between possible edges. For more complex situations, anyway, it is useful to master a few adjustable parameters.

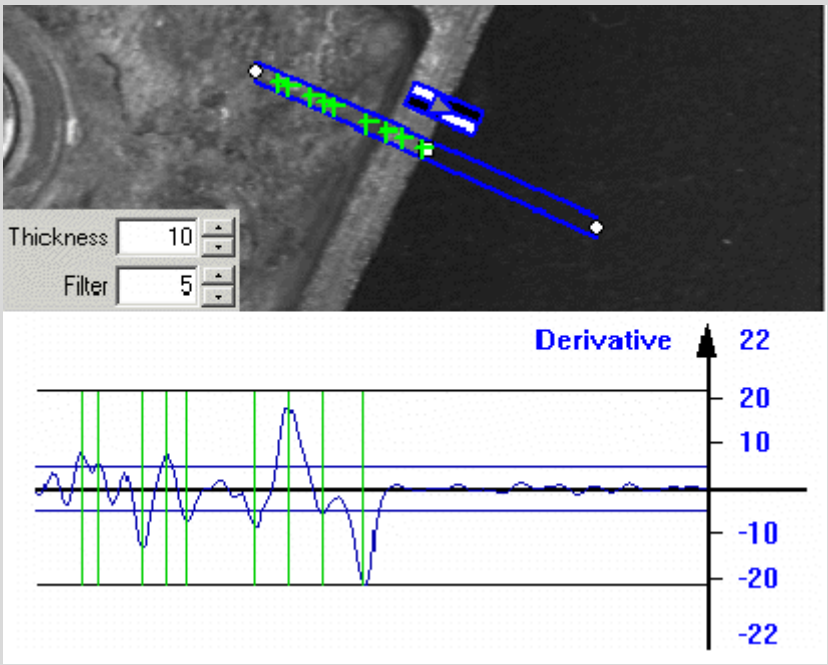
The first step in a gauging application, besides possible calibration, is setting the gauge location, including tolerance. The center position and orientation are easy to decide based on a sample image or on coordinate considerations.

The location tolerance is depending on the possible variations of the edge position. The larger the tolerance, the more likely the edge will be hit; on the other hand, the more likely hitting false edges or extraneous features.



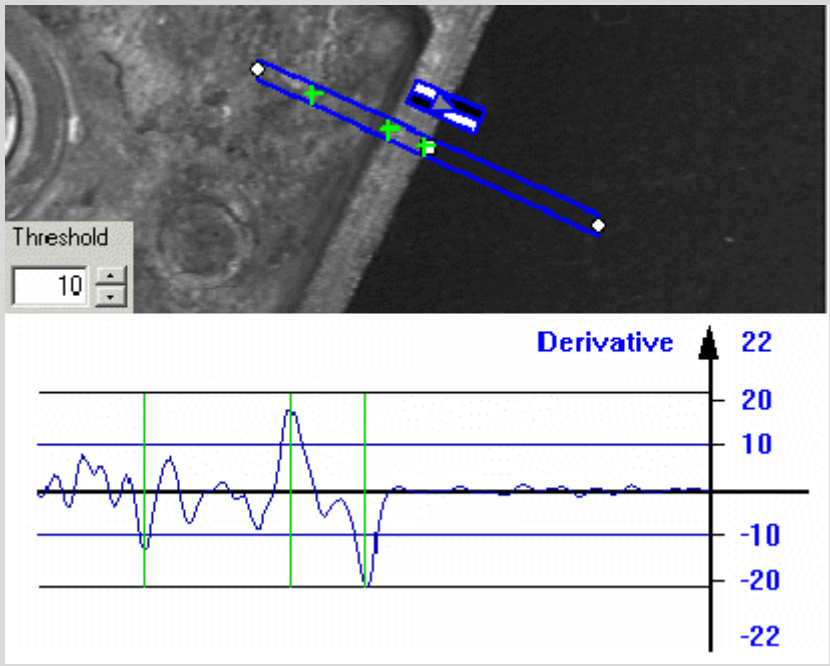
Placing and sizing the point gauge

After overlaying the gauge to the right location, one can observe the profile curve and its derivative. The curve regularity gives an indication of the spread of the gray-level values. It is a matter of judgment to decide whether filtering by projection or smoothing is required (playing with the filtering parameters while looking at the plotted curve can help).



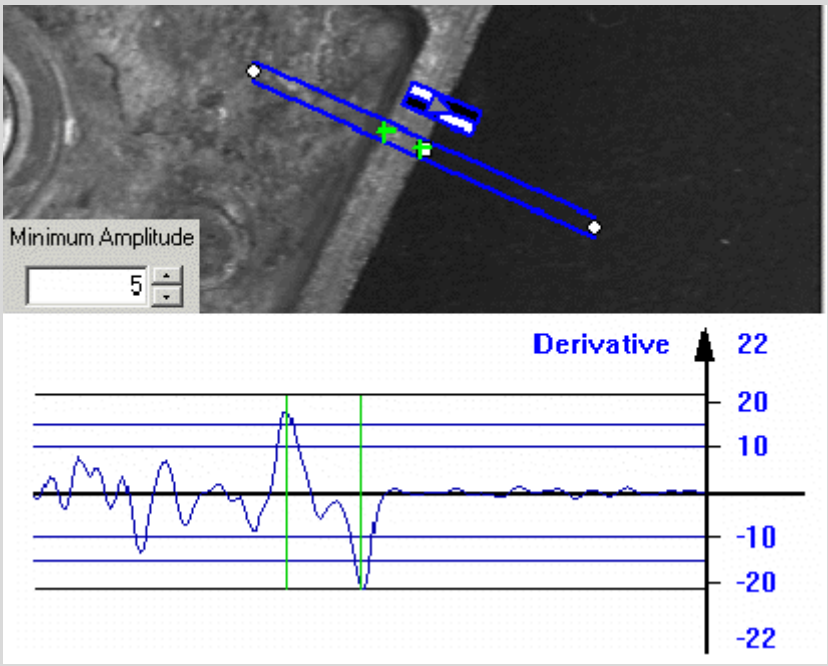
Projecting for noise reduction

When these coefficients are set, the gray-level profile will not change anymore. It is then appropriate to set the **threshold** value appropriately. A low value should be used to make sure that the useful parts of the peaks cover enough pixels (to achieve better sub-pixel accuracy), but not lower than the ambient image noise.



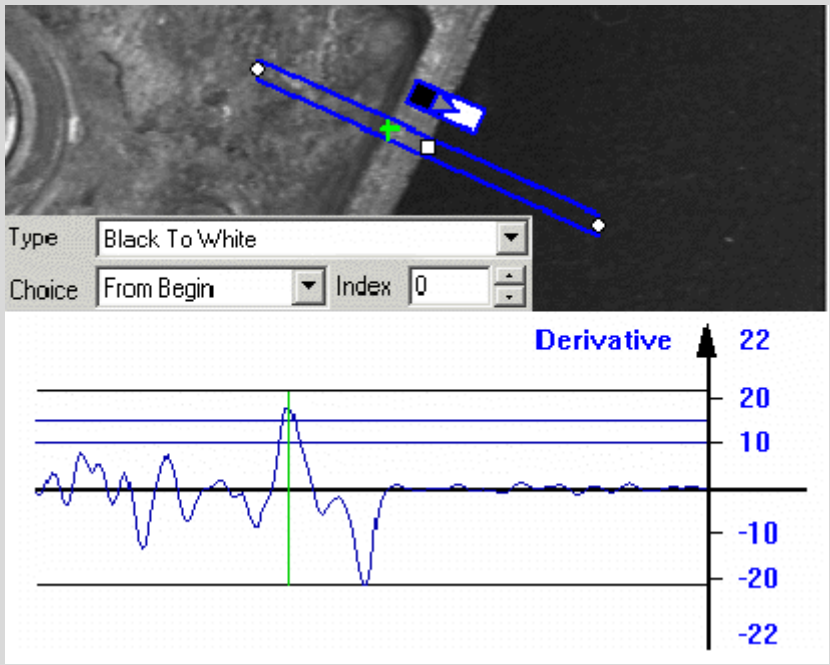
Setting the threshold

Looking at the list of amplitudes and areas of the distinct peaks helps getting rid of unwanted weak or false edges. Plotting these values along with good and extraneous peaks can help find appropriate peak rejection limits, if need be.



Getting rid of the weak transitions

The next and final decision regards whether all transition points are needed or a single one suffices (the most relevant one). If all are required, they can be queried one after another. Otherwise, a point selection strategy should be chosen. The selection can be based on transition polarity (black to white and/or conversely), and a strength or order criterion.



Using the most relevant edge

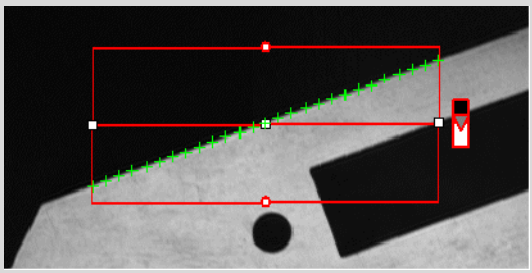
3. Model Fitting

Locating a point on an edge is a first step to gauging. It allows finding the position of an item with a single degree of freedom, such as a part gliding along a rail. Anyway, in many other situations, following edges to determine curvatures, orientations, thicknesses... is required.

EasyGauge is able to work by fitting a simple geometric shape such as a line segment or an arc to a given edge. To achieve this, the nominal position of the targeted edge is define, and points are sampled along it using regularly spaced point measurement gauges. When a number of such edge points are available, model fitting in the least square sense can be applied.

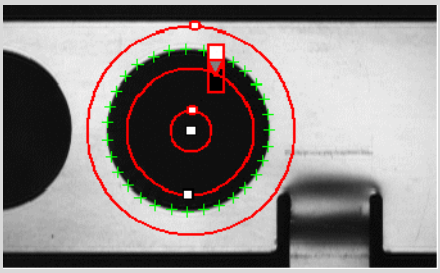
EasyGauge supports four types of models:

- **Line** segment: used to measure position and orientation of straight edges.



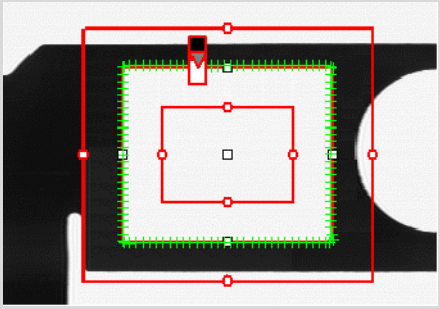
Line fitting

- **Circle**: used to measure position and curvature or curved edges. The circle can be full or limited to an arc.



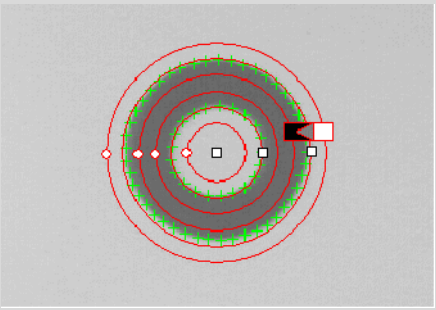
Circle fitting

- **Rectangle**: used to measure position, orientation and size of a rectangle. All four sides can take part or not to the measurement, allowing to measure thickness (between opposite sides) of straight items, or corner location (between adjacent edges).



Rectangle fitting

- **Wedge**: used to measure position, orientation and size of a ring, a disk sector or a curvilinear rectangle (two curved, concentric, sides). All four sides can take part or not to the measurement, allowing to measure thickness (between opposite arcs), angular aperture (between opposite line segments), or corner location (between adjacent edges).

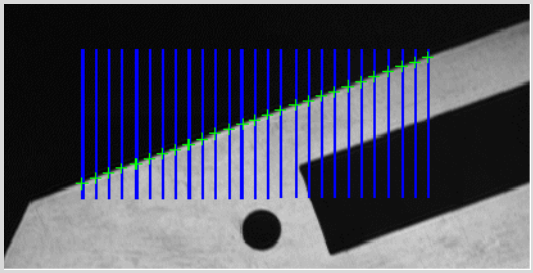


Wedge fitting

Common Features

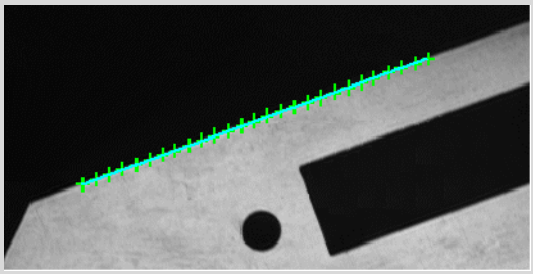
The four gauge types ELineGauge, ECircleGauge, ERectangleGauge, EWedgeGauge share the following features:

- **Point sampling**: point gauges are moved along the edges and point measurement carried out at regularly spaced spots. All the point measurement parameters and operating modes are available. They can even be adjusted differently per side in the case of rectangle and wedge gauges. The spacing of the point location gauges along the model is governed by **SamplingStep**, while **NumSamples** returns the number of points sampled during the model fitting operation.



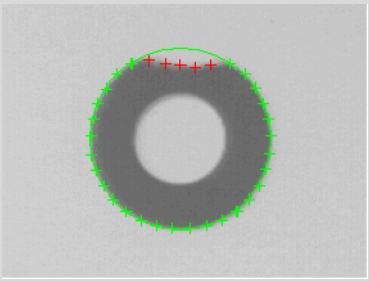
Sampling paths and sampled points

- **Model fitting**: the synthetic model is adjusted so as to minimize the error residue, this providing the best estimate of the edge parameters. In the case of rectangles and wedges, parallelism and concentricity constraints are enforced, allowing true thickness measurements.



Sampled points and fitted line

- **Outlier rejection**: after model fitting, some of the points can be found to lie too far away from the fitted model. These can have a harmful effect on location accuracy and should be ignored. EasyGauge allows applying a rejection phase that tags as outliers the sample points lying at a certain distance of the fitted model. Outliers elimination is performed by setting the **FilteringThreshold** property. This outlier elimination process can be repeated several times using **NumFilteringPasses**. The number of valid sample points remaining after a model fitting operation is retrieved by means of **NumValidSamples**. The average distance of these points to the fitted model is returned by **AverageDistance**.



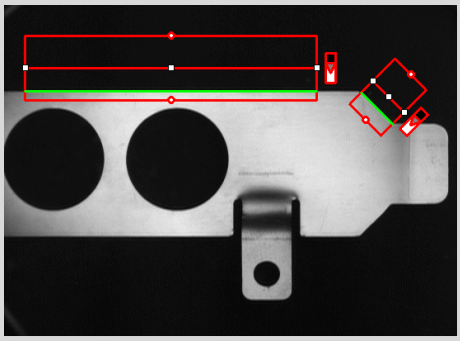
Outlier rejection

The Tolerance parameter determines the extent of the point location gauges to be used around the model.

3.1 Line Gauge

The placement of a line gauge is defined by its center coordinates, its length and its angle with respect to the X-axis.

Note. When fitting a line, the slope may be known or not. To constrain the line slope value, set **Angle** and **KnownAngle**.



Line fitting

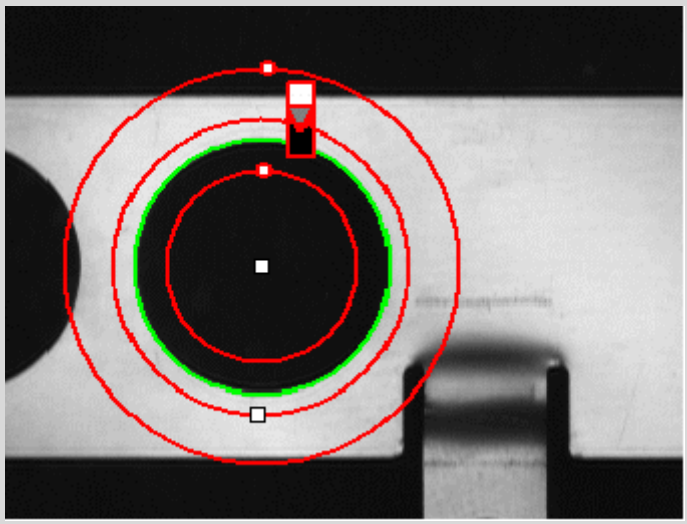
Once the gauge has been defined and positioned, use Measure to trigger the line fitting operation. To obtain the measurement results, set the ActualShape property to TRUE. The ActualShape mode determines whether an inquiry returns the fitted line (TRUE value) or the nominal line position (FALSE value, default). The requested information is then retrieved by means of the line properties.

Alternatively, MeasuredLine provides the results as an ELine object. Information pertaining to the fitted line is then retrieved through this ELine object.

3.2 Circle Gauge

The placement of a circle gauge is defined by its nominal position (given by the coordinates of its center), its nominal diameter (or radius), the angular position from where it extents and its angular amplitude.

Distinct overloads of the Set member are available to distinguish between a full circle and an arc. In the latter case, the arc amplitude must be specified too.



Circle fitting

Once the gauge has been defined and positioned, use Measure to trigger the circle fitting operation. To obtain the measurement results, set the ActualShape mode to TRUE. The ActualShape mode determines whether an inquiry returns the fitted circle (TRUE value) or the nominal circle position (FALSE value, default). The requested information is then retrieved by means of the circle properties.

Alternatively, MeasuredCircle provides the results as an ECircle object. Information pertaining to the fitted circle is then retrieved through this ECircle object.

3.3 Rectangle Gauge

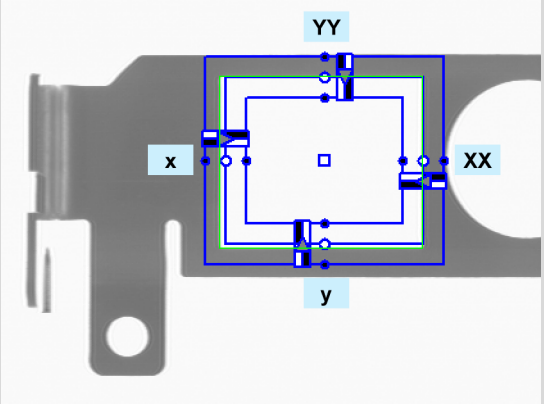
The placement of a rectangle gauge is defined by its nominal position (given by the coordinates of its center), its nominal size and its rotation angle.

Each side of a rectangle can have its own transition detection parameters. The four sides of the rectangle can be set to an active or inactive state by means of the ActiveEdges property. When a side is active, this means that:

- setting the value of a parameter only applies to the currently active sides;

- getting the value of a parameter yields a result only when the value of this property is the same for all active sides;
- only active sides are used for measurement and model fitting.

These rules allow setting different parameters for different sides, and measuring parallel sides or a corner point instead of the whole rectangle. The four sides are denoted by letters "x", "y", "XX" and "YY" respectively.



Naming conventions for the sides of a rectangle gauge

Once the gauge has been defined and positioned, use Measure to trigger the rectangle fitting operation. To obtain the measurement results, set the ActualShape mode to TRUE. The ActualShape mode determines whether an inquiry returns the fitted rectangle (TRUE value) or the nominal rectangle position (FALSE value, default). The requested information is then retrieved by means of the rectangle properties.

Alternatively, MeasuredRectangle provides the results as an ERectangle object. Information pertaining to the fitted rectangle is then retrieved through this ERectangle object.

3.4 Wedge Gauge

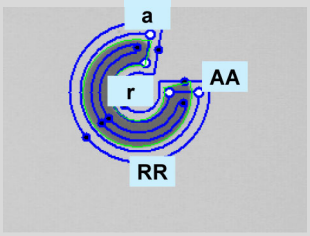
The placement of a wedge gauge is defined by its nominal position (given by the coordinates of its center), its nominal inner and outer radius (inner and outer diameter), its breadth (difference between radii), the angular position from where it extents and its angular amplitude.

Distinct overloads of the **Set** member are available to distinguish between a full ring and a sector of a ring. In the latter case, the sector amplitude must be specified too. Distinct overloads are also available to distinguish between a disk (null inner radius) and a ring.

Each side of a wedge can have its own transition detection parameters. The four sides of the wedge can be set to an active or inactive state by means of the ActiveEdges property. When a side is active, this means that:

- setting the value of a parameter only applies to the currently active sides;
- getting the value of a parameter yields a result only when the value of this parameter is the same for all active sides;
- only active sides are used for measurement and model fitting.

These rules allow setting different parameters for different sides, and measuring parallel arcs or oblique sides, or a corner point, instead of the whole wedge. The four sides are denoted by letters "a", "r", "AA" and "RR" respectively.



Naming conventions for the sides of a wedge gauge

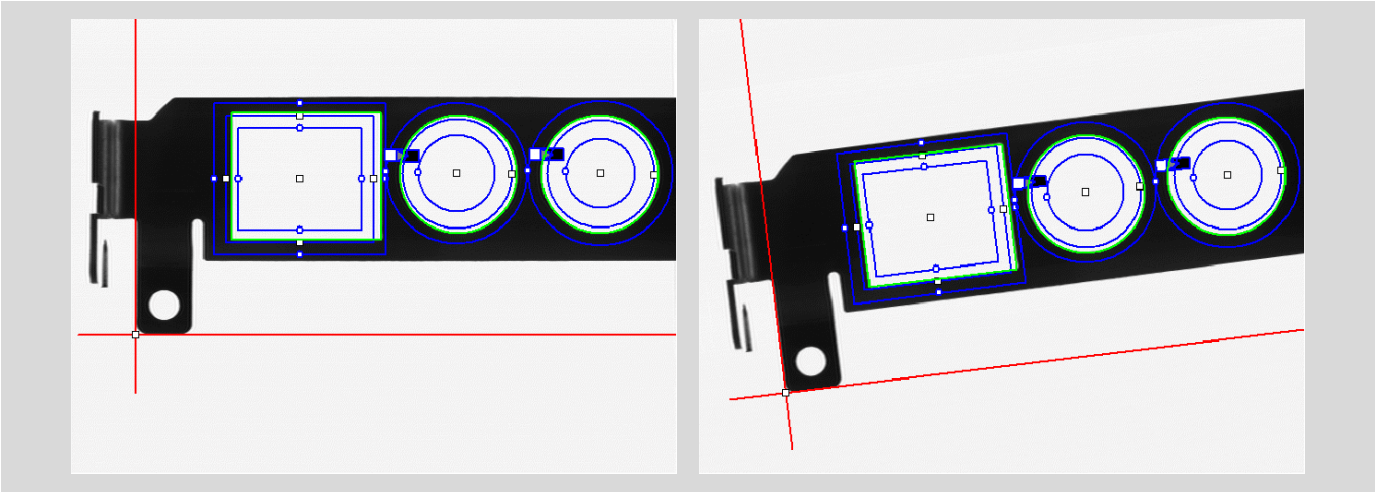
Once the gauge has been defined and positioned, use Measure to trigger the wedge fitting operation. To obtain the measurement results, set the ActualShape mode to TRUE. The ActualShape mode determines whether an inquiry returns the fitted wedge (TRUE value) or the nominal wedge position (FALSE value, default). The requested information is then retrieved by means of the wedge properties.

Alternatively, MeasuredWedge provides the results as an EWedge object. Information pertaining to the fitted wedge is then retrieved through this EWedge object.

4. Gauge Grouping

When a part inspection required more than one measurement site, several gauges can be tied together to form a dedicated measurement tool. Grouping is achieved by attaching the gauges to a common coordinate frame or to another gauge. Once the gauges are attached, their relative placement remains fixed, and they can be moved as a whole.

Use Attach to associate a gauge to a mother gauge or frame. Information relative to the attached daughters or the mother is retrieved by means of NumDaughters, GetDaughter, or Mother. To dissociate the daughters from a mother, use Detach, DetachDaughters.



Gauge grouping for object tracking

5. Graphical Interaction

EasyGauge provides means to graphically interact with the gauges to place and size them, combine them as a hierarchy of grouped items, and store/retrieve them and all working parameters to/from model files.

Drawing

The gauge classes **Draw** function gives a graphical representation of a gauge. Drawing is done in the device context associated to the desired window. The current pen is used. Depending on the allowed operations, specific handles are displayed.

Dragging

Dragging a gauge allows an operator to move it interactively on an image. Several dragging handles are available.

- **HitTest** method allows to determine whether the mouse cursor is over a handle. When this is the case, the cursor shape should be changed for feedback, and a dragging operation can take place.
- **Drag** method effectively moves the handle and the corresponding gauge accordingly.

Plotting

For parameter tuning purposes, EasyGauge can plot the gray-level values along the sampled paths and/or its derivative. Use method **Plot** for this purpose.

In case of a point measurement gauge, the plot is available after a call to the Measure method. In case of a model fitting gauge, the plot is available after a call to the MeasureSample method. This one takes an index argument that must lie between 0 and GetNumSamples-1 (included). To view the corresponding sampling path, refer to method Draw using mode EDrawingMode_SampledPath.

6. Calibration and Unwarping

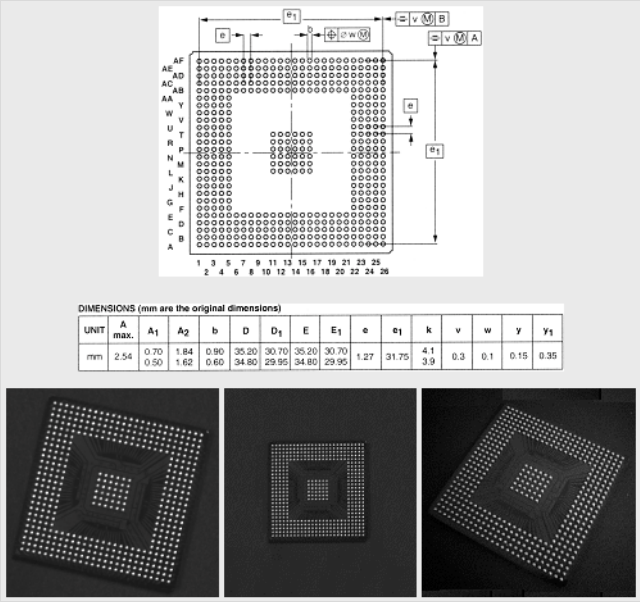
An **EWorldShape** object manages a field-of-view calibration context. Such an object is able to represent the relationship between world coordinates (physical units) and sensor coordinates (pixels), and account for the distortions inherent in the image formation process.

Image calibration is an important process in quantitative measurement applications. It establishes the relation between the location of points in an image (pixel indices) and the actual positions of those points in the real world, on the inspected item.

Calibration can be setup by providing explicit calibration parameters of the calibration model, or by identifying those parameters by means of a set of known points (landmarks), or by means of a calibration target.

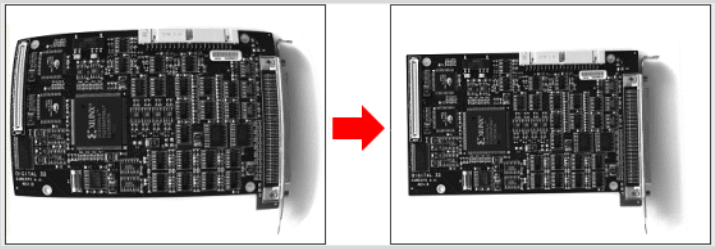
The goal of calibration is twofold:

- It allows to gain independence with respect to the viewing conditions (part placement in the field of view, lens magnification, sensor resolution, ...), letting you describe the inspected item once for all using absolute measurements.



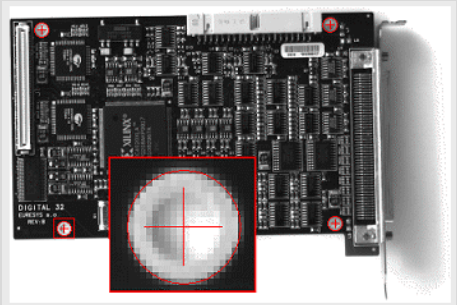
Single model versus multiple viewing conditions

- It can correct some sources of distortion related to the imaging process (perspective effect, optical aberrations, ...).



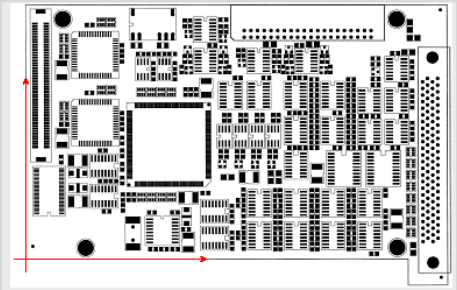
Removal of image distortion

The pixel indices in an image usually are integer numbers, but fractional values can be encountered when using sub-pixel methods. In what follows, these values will be termed sensor coordinates. Sensor values are normally obtained by somehow processing an image and locating known feature points in it.



Feature point in sensor space

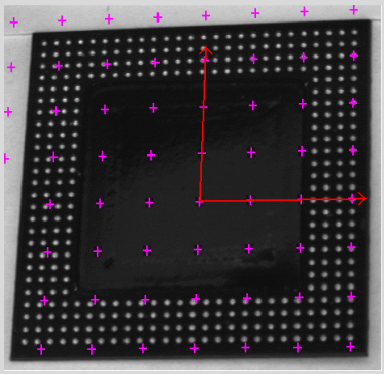
The world coordinates are used to describe location of points on the inspected item. They require a reference frame to be defined. The world values are expressed in an appropriate length measurement unit and correspond to actual dimensions. World values are usually gathered from design considerations (part drawing) or by mechanical measurements on a part.



Reference frame in world space

6.1 What Is Calibration?

Calibration establishes the relationship between point coordinates in the real world and pixels in the image. The calibration transform is handled by a specific device named EWorldShape, which provides means to compute the appropriate calibration coefficients. The transform is then implicitly applied to all measurement gauges tied to it.

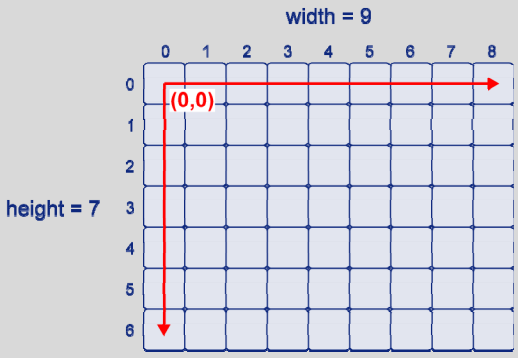


Field-of-view calibration

6.2 Sensor Coordinates (Pixels)

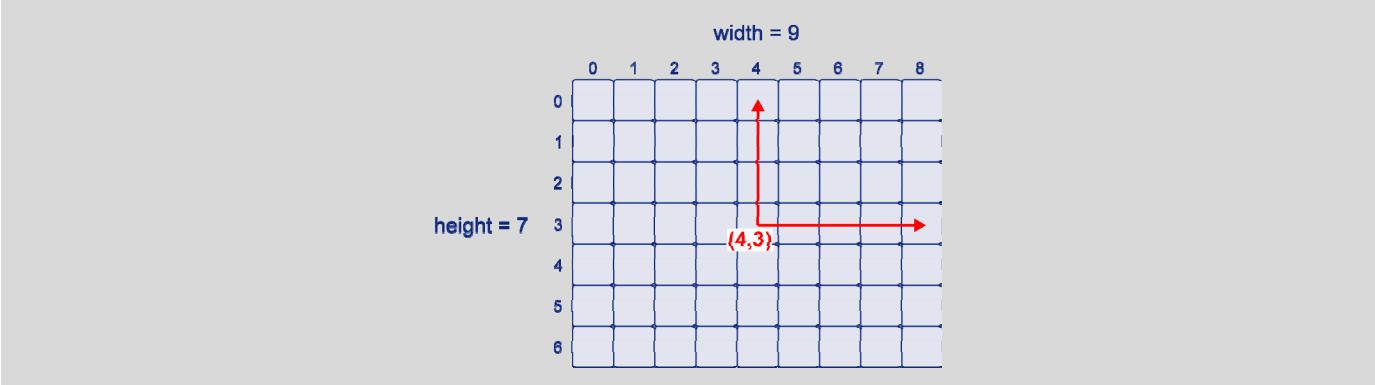
The sensor space is two-dimensional and directly maps to the array of sensitive sites of a CCD camera (or to places where the video signal is sampled by the frame grabber to yield a digital image).

The most natural coordinate system that is used on a sensor starts from the upper leftmost pixel and extends rightwards and downwards (this corresponds to the usual storage order of the pixels in a frame buffer). The range of abscissae is **0** to **width-1** and the range of ordinates from **0** to **height-1**. This definition assumes that the integer coordinate values correspond to the pixel centers.



Raw sensor coordinate system

Quite often, anyway, another setting is preferred: the origin point of the coordinates lies at the center of the sensor —**([width-1]/2, [height-1]/2)** in the previous system— the axis extending rightwards and upwards.

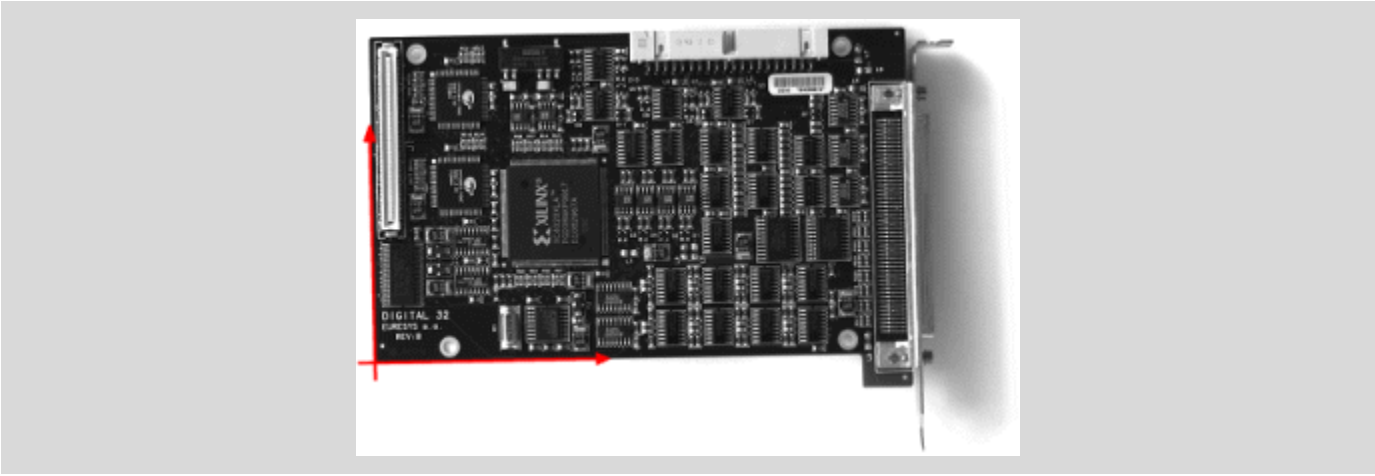


Centered sensor coordinate system

6.3 World Coordinates (Physical Units)

As you know, the true world is three-dimensional. An imaging system looses information because it acts as a projection device. A plurality of points project to the same location on the imaging plane. Anyway, for most applications, the field of view is a plane or can be regarded as a flat surface, the objects of interest being sufficiently flat and on the same level. This allows considering that the world coordinates are defined in a 2D reference frame tied to a reference plane, such as the work surface.

The origin and direction of the axis are most of the time aligned with major features (symmetry lines, corners, holes, ... and straight edges) of the inspected parts.



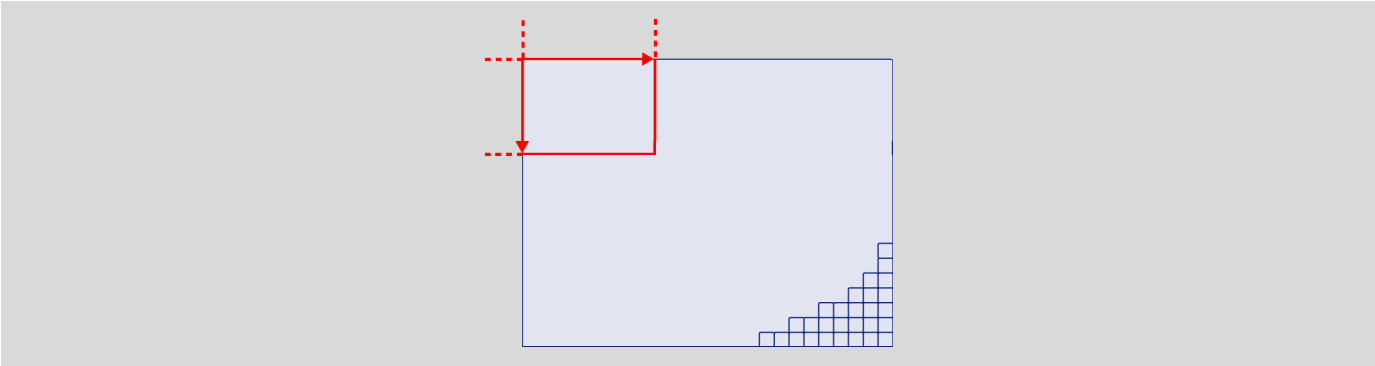
World coordinate system

6.4 World-to-Sensor Transform

When converting from world to sensor coordinates, several effects can be accounted for. The following figures show, in increasing order of complexity, how these effects combine and model the actual image formation process.

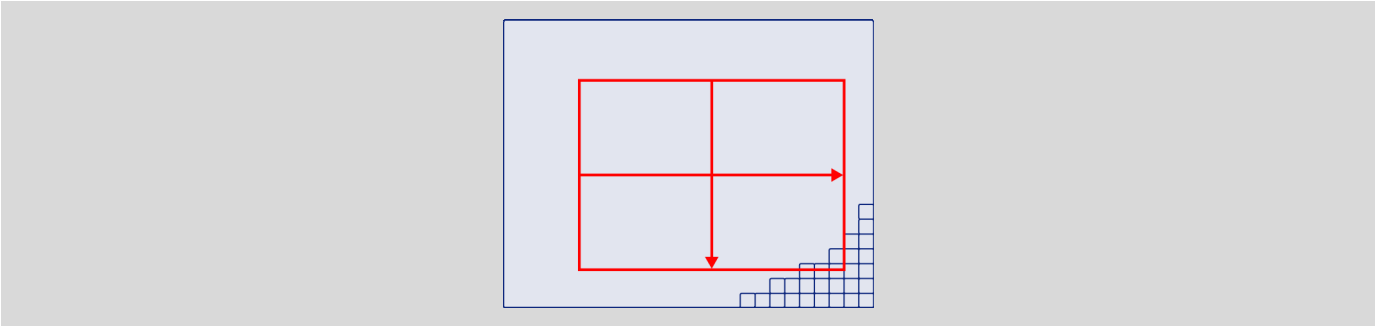
Simple Calibration

When the image is not calibrated, **the world and sensor coordinates are identical**.



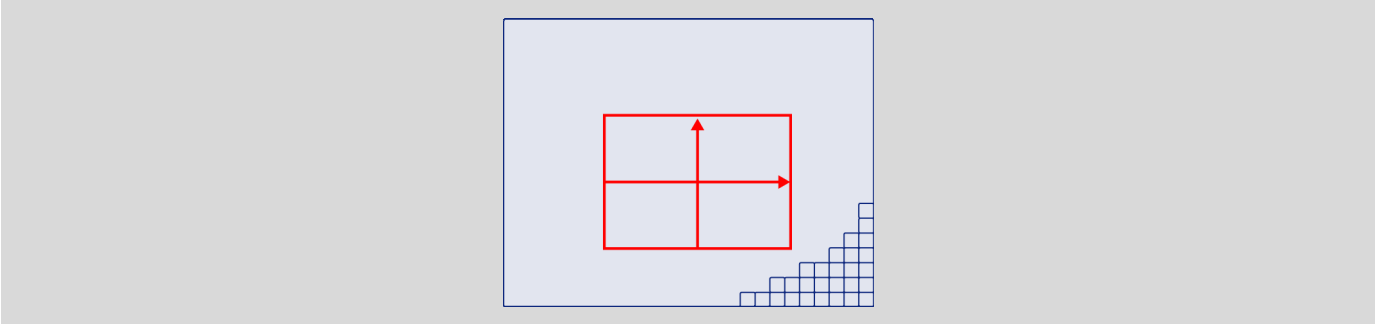
Simple calibration: uncalibrated

For convenience, the coordinate origin can be brought to another place than the upper left corner. This is a **translation** effect. World coordinates still correspond to pixel units.



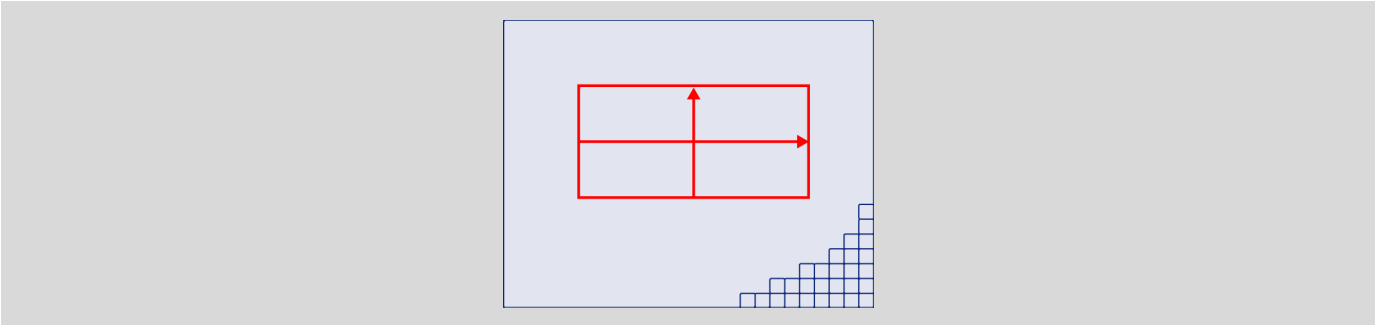
Simple calibration: translated

The next step involves converting the pixel values to physical measurement. This involves introducing an appropriate scale factor (f.i. in pixels per inch) for unit conversion. When this factor is the same in all directions, we say that the pixels are square. This is **isotropic scaling**.



Simple calibration: scaled (square pixels)

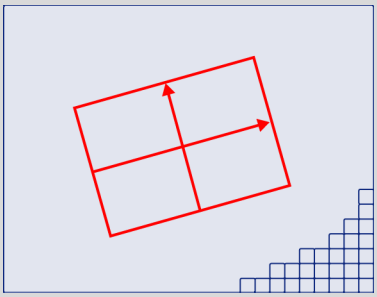
When they are not square (because of mismatching spatial sampling frequencies), two distinct scale factors are required. This is **anisotropic scaling**. Beware that pixels are always displayed as if they were square. The resulting visible effect is a stretching of the image in one direction. There is a restriction pertaining to the allowed image anisotropy. The corresponding pixels aspect ratio (X resolution/Y resolution) should be in the range **[-4/3, -3/4]** (or **[3/4, 4/3]**), otherwise the calibration could fail.



Simple calibration: scaled (non-square pixels)

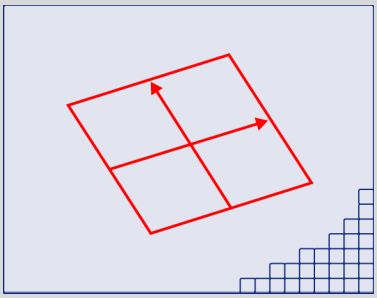
Affine Calibration

When the inspected part is rotated by some angle, it is convenient to keep the world axis aligned with it. In this case, the transform involves **translation, rotation and scaling**.



Affine calibration: scaled and skewed (square pixels)

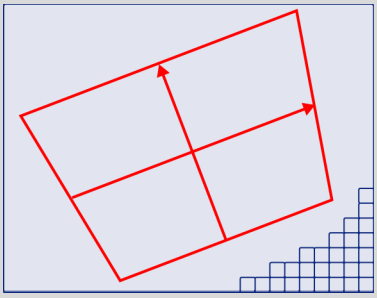
In the case of non-square pixels, distortion of the right angles becomes apparent. Non-square pixels frequently arise when using a linear-scan camera, when the scanning speed does not match the pixel spacing.



Affine calibration: scaled and skewed (non-square pixels)

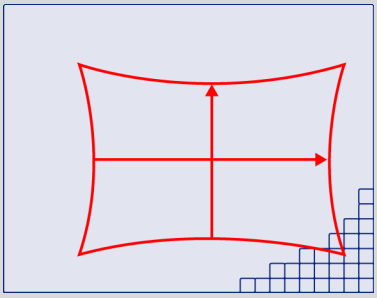
Advanced Calibration

Ideally, the field of view should be parallel to the sensor plane (both being perpendicular to the optical axis). When it is not the case, further distortion becomes apparent and the so-called **perspective** effect shows up: the further the objects, the smaller they look; lines remain straight but angles are not preserved. Such a situation should be avoided or minimized by a careful alignment of the camera.



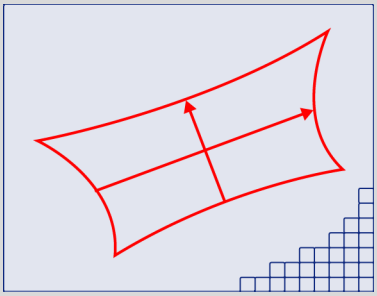
Advanced calibration: perspective distortion

Additionally to the perspective effect, the camera lens unavoidably introduces distortion, related to the so-called **optical** aberrations. This effect is such that the local magnification changes with the distance from the optical axis, resulting in cushion or barrel appearance of rectangles.



Advanced calibration: optical distortion

All effects combined result in a complex, non linear, transform from world to sensor spaces.



Advanced calibration: combined distortions

Distortion Sources Elimination

Whenever possible, the sources of distortion should be eliminated.

Non-square pixels may depend on some sweep frequency or scanning speed; it should be adjusted as accurately as possible.

Perspective effect can be reduced by improving the optical axis alignment.

Optical distortion is reduced by using longer focal distances and better quality lenses. In normal cases, these sources of distortion should remain invisible to the naked eye.

The scale factor is directly influenced by the lens magnification, itself depending on the observation distance and the focusing.

In all cases, try to minimize the skew and translation allowance by providing tight mechanical fixture or accurate synchronization between part movement and acquisition triggering.

Also keep in mind that the more repeatable the part position is, the easier it will be located for inspection. The calibration functions have been designed in such a way that the simpler the model, the faster the computation. For this reason, it is recommended to choose the calibration model of the appropriate complexity.

6.5 Calibration using EWorldShape

The world object allows you to calibrate the whole field of view once for all (in given imaging conditions -fixed camera placement and lens magnification). (The calibration step will have to be remade any time the optical setup is modified.)

Depending on the application's needs and calibration facilities available, several approaches can be taken:

- **By guesswork**: the calibration parameters, such as actual pixel resolution, can be provided explicitly. In this case, the user must measure them by his own means. This is mostly feasible for the simplest calibration modes.
- **Using landmarks**: sets of reference points, called landmarks, can be passed to a calibration function able to compute all required parameters. The landmarks can be chosen at will on the inspected item.
- **Using a calibration target**: a calibration target with an appropriate pattern (grid of dots) can be automatically analyzed to get an appropriate set of landmarks.

The **EWorldShape** class provides all necessary means to set the world-to-sensor transform parameters, perform conversions from and to either coordinate system, and determine unknown calibration parameters.

Additionally, you can save the parameters of a given transform to a file for later reuse.

Calibration by Guesswork

When no sophisticated distortion correction is required and when the accuracy requirements are low, calibrating by providing coefficient estimates is sufficient. Use the following procedure to obtain the pixel calibration coefficients:

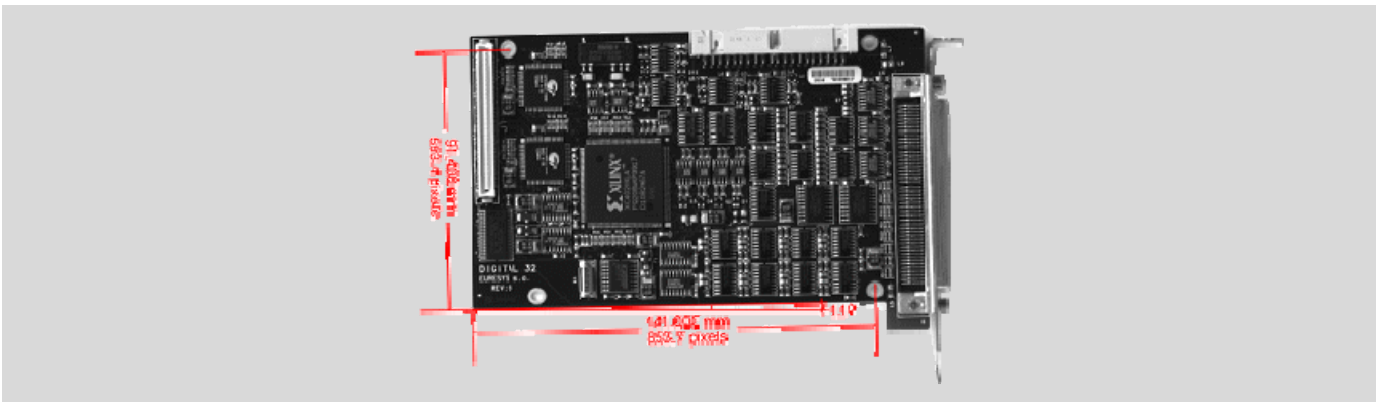
1. Take a picture of a known object such as the part to be inspected or a calibration target (chessboard, dot pattern, rectangle, ...). Locate known feature points such as corners in the image (by the eye) and determine their coordinates in pixel units —let **(i,j)**.

2. Use the Euclidean distance formula to derive the calibration coefficient: $$C = \frac{\sqrt{(i_1 - i_0)^2 + (j_1 - j_0)^2}}{D}$$ where **C** is a calibration coefficient, in pixels per unit, and **D** is the world distance between the corresponding points, in units.
3. Repeat this operation for pairs of horizontal and vertical points in case of non-square pixels.

Another convenient way is to locate the limits of the field of view in the actual scene and obtain the calibration coefficients by dividing the image resolution by the actual field of view size.

It is also possible to estimate the skew angle, if any, by means of the following formula applied to two points on the X-axis in the world system:

$$\theta = \arctan \frac{j_1 - j_0}{i_1 - i_0}$$



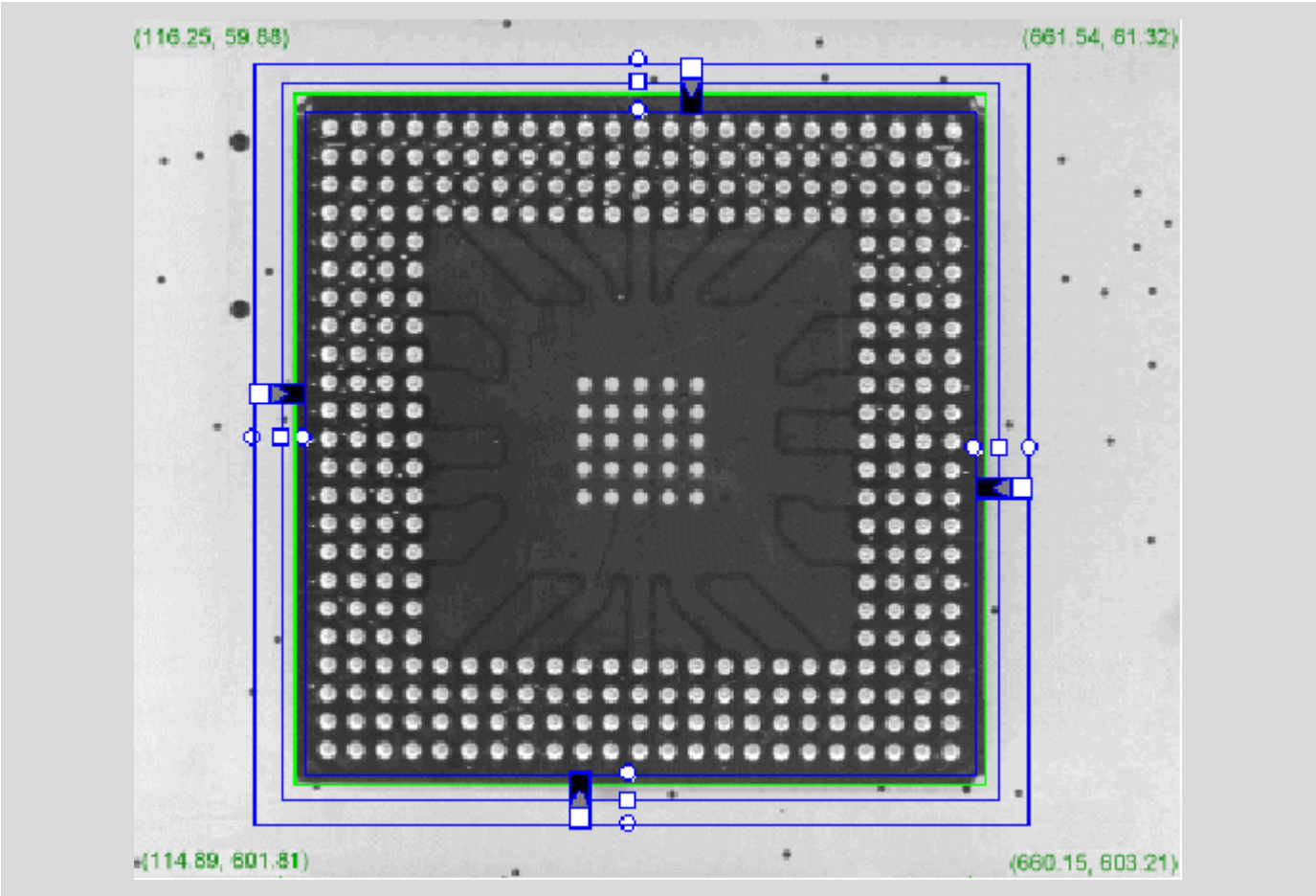
Estimating scale factors and skew angle

When the calibration coefficients are available (obtained from the above mentioned procedure or by any other means), use **SetSensor** to adjust them and set the corresponding calibration mode.

Calibration coefficients can also be programmed one at a time using: **SetSensorSize**, **SetFieldSize**, **SetResolution**, **SetCenter**, **SetAngle**.

Landmark-Based Calibration

Assume you have at disposal an object of known dimensions such that a number of points can be located on it by means of image processing methods. For instance, you can accurately locate the four corners of a rectangle using a rectangle fitting gauge, as can be done with **EasyGauge**. Such feature points are commonly called *landmarks*.



Locating a rectangle's corners

Whatever the target shape and structure, the procedure always amounts to locating the desired feature points and providing their coordinates in both the sensor and world coordinate systems. The sensor coordinates are obtained by image processing means, and corresponding world coordinates are derived from actual measurements on the part or by computation from a blueprint.

The more landmark points are used, the more accurate calibration will be. At least 4 of them are necessary in most calibration modes. Anyway, there is absolutely no constraint on their relative placement, so that any calibration pattern or shape can be used.

However, there is a restriction pertaining to the allowed image anisotropy. The resulting pixels aspect ratio (X resolution/Y resolution) should be in the range [-4/3, -3/4] (or [3/4, 4/3]), otherwise the calibration could fail.

Dot-Grid-Based Calibration

In the previous section, landmark-based calibration constitutes an easy way to achieve automatic calibration, provided an appropriate procedure is available to extract the desired landmark point coordinates.

Open eVision provides an improved method that allows using the landmark approach with straightforward image processing techniques. It relies on the use of a specific target holding a rectangular grid of dots (the dots need not be circular; they just have to be symmetrical).

The image can contain dark dots on a light background or light dots on a dark background. Notice that no other object may lie on the grid part. Here is such an example.

The procedure is as follows:

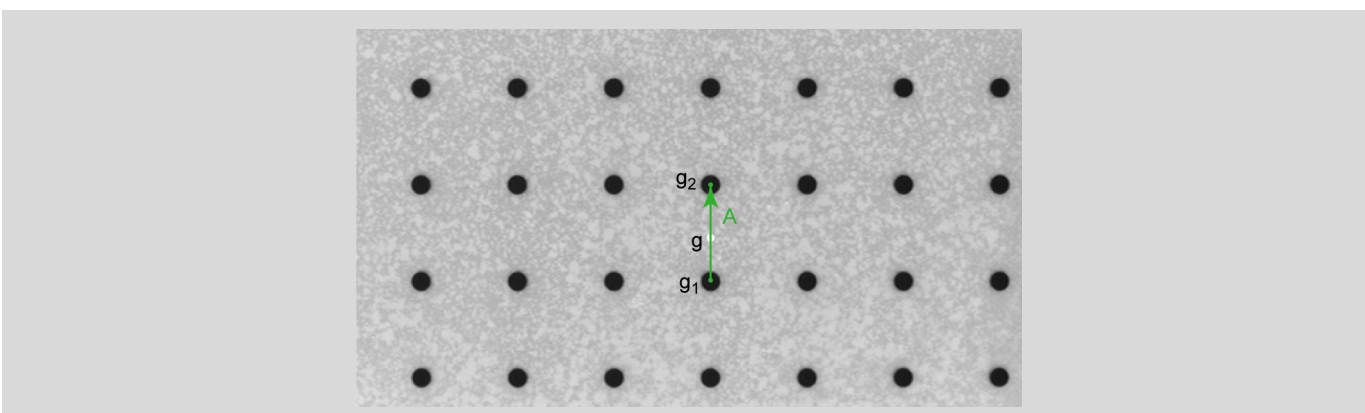
1. Grab an image of the calibration target in such a way that it covers the whole field of view (if this is not feasible, restrict the image of view to an ROI where only dots are visible).
2. Apply the standard tools of blob analysis to extract the coordinates of the centers of the dots, as can be done by EasyObject.
3. Pass all points detected to **AddPoint** (sensor coordinates only.)
4. Call **RebuildGrid** The latter will fully reconstruct the grid geometry from the given coordinates and so be able to compute the world coordinates of each dot.
5. Call **Calibrate**the landmark approach.

Grid construction algorithm

The goal of **RebuildGrid** is to reconstruct the topology of a grid, to calibrate a field of view. The grid construction uses an iterative algorithm that works as follows:

Step 1

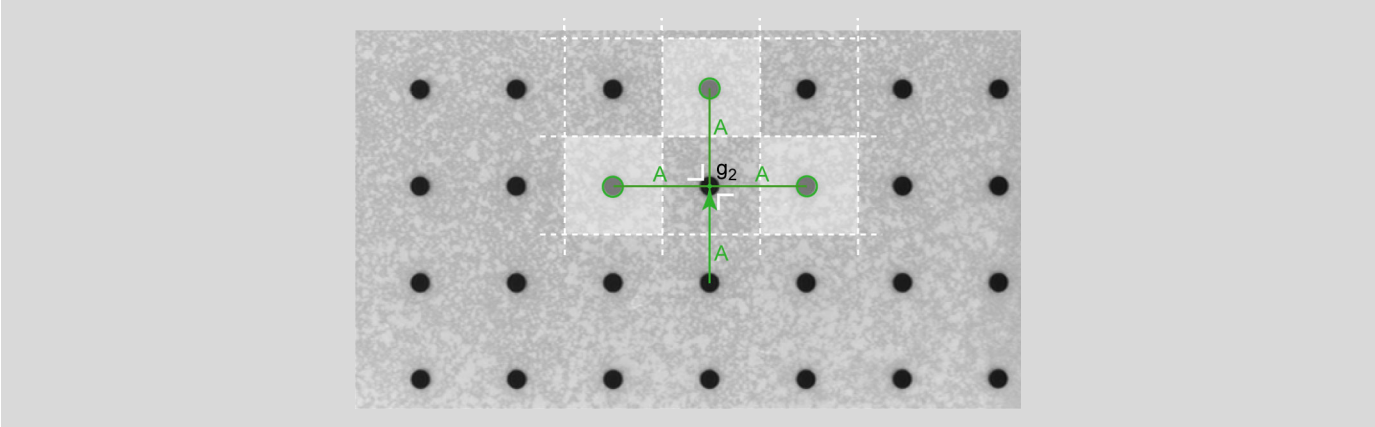
The two grid points (g_1 and g_2) nearest to the gravity center (g) of the grid points are selected to form the first reference oriented segment, of length A.



Grid construction: step 1

Step 2

Starting from the extremity of the reference segment (g_2), the algorithm determines 3 tolerance areas (white squares in the figure), in perpendicular directions. The tolerance areas are centered at a distance A (length of the reference segment) from the reference segment extremity (g_2). They are square, with a side-length of A. The algorithm searches for 1 neighboring point, in each of the 3 tolerance areas. The grid will be correctly calibrated if the 3 neighbors are located in the 3 tolerance areas.



Grid construction: step 2

Step 3

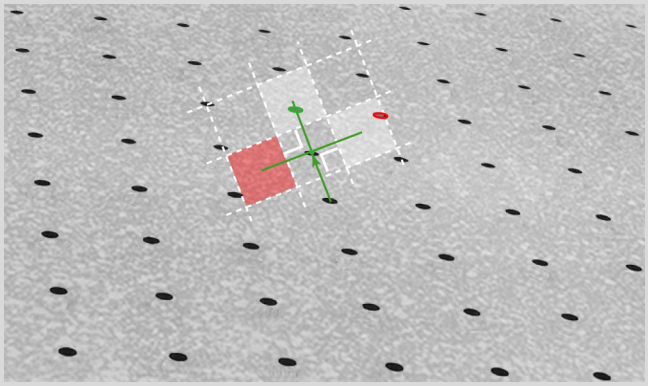
The 3 perpendicular segments are the references of the next iterative searches. The algorithm goes back to step 2.

Remark

When the grid exhibits too much distortion, the following errors could happen:

1. One or more neighboring points fall outside the tolerance areas (red square in the figure).
2. More than one neighboring points fall inside a tolerance area.
3. The point in the tolerance area is not the correct one. For instance, the point might be diagonally connected (red point in the figure).

In these three offending cases, the grid reconstruction does not work as expected.



Bad grid construction on a too distorted image

Calibration Coefficients

In order to fully describe the field-of-view calibration model, a set of parameters is used, with the following meaning:

Sensor Width and Height

The sensor width and height give the full image logical size, in pixels. This is always an integer number, for obvious reasons.

Field-of-View Width and Height

The field-of-view (f-o-v) width and height give the full image physical size, in length units, i.e. the size of the rectangle corresponding to the image edges in the world space. These values are related to the pixel resolution by the following equations:

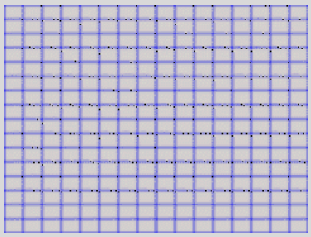
```
 f-o-v width = pixel width * sensor width
f-o-v height = pixel height * sensor height
```

or

```
sensor width = f-o-v width * horizontal resolution
sensor height = f-o-v height * vertical resolution
```

By default, i.e. when the height is not specified, the pixels are assumed to be square (pixel width = pixel height).

Ratio



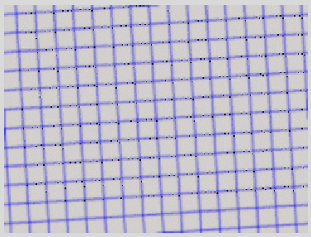
Anisotropic aspect ratio

Center Abscissa and Ordinate

The center abscissa and ordinate indicate where the origin point (world coordinates (0,0)) falls in the image. By default, this is the image center.

Skew Angle

The skew angle indicates the angle formed by the reference frame (X-axis) in the world space and the image edge (horizontal). By default, there is no skew, i.e. the world axes are well aligned.

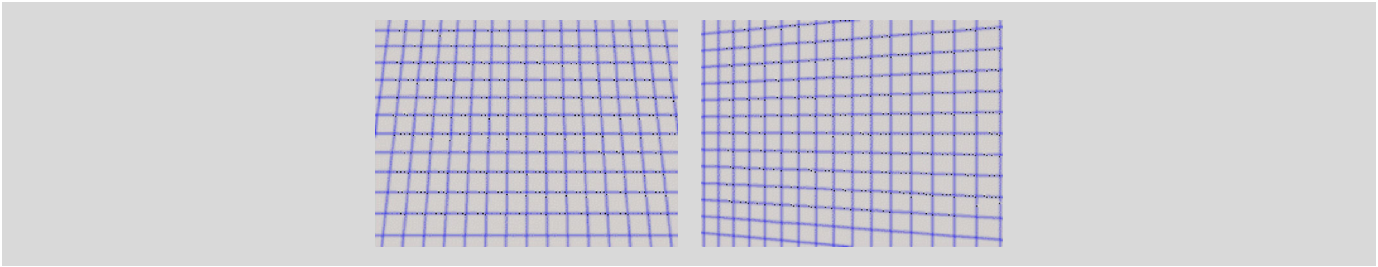


Skew angle

Note. When the pixels are not square, the skew angle does not appear as such in the image. The **EWorldShape** object provides an appropriate method to convert between the angle in world and sensor spaces.

X and Y Tilt Angles

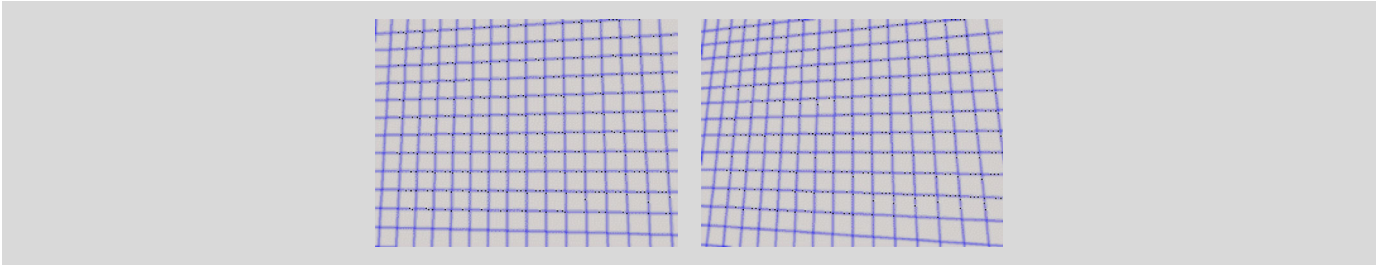
The X and Y tilt angles describe the direction of the viewing plane. The field of view cannot be perfectly perpendicular to the optical axis. The two tilt angles correspond to the required rotations around the X and Y axis that bring the Z axis parallel to the optical axis.



Tilt X and tilt Y angles

Perspective Strength

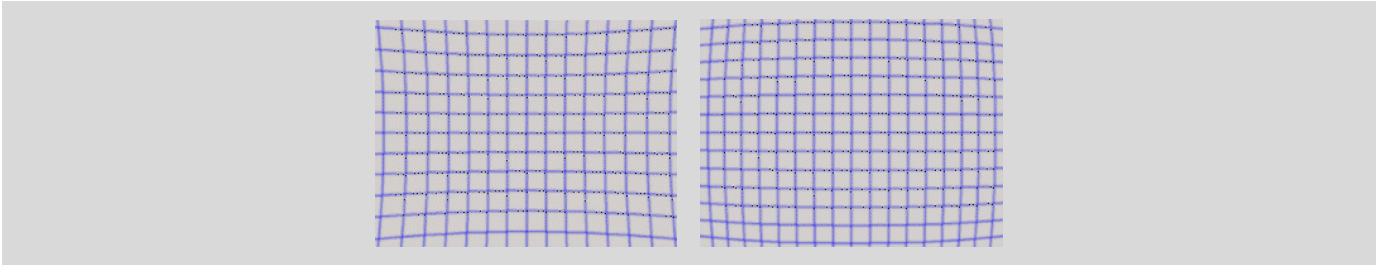
The perspective strength gives a relative measure of the perspective effect. The shorter the focal length, the larger the value.



Weak and strong perspective

Distortion Strength

The distortion strength gives a relative measure of radial distortion at the image corners, i.e. the ratio of the image diagonal length with and without distortion.



Positive and negative distortion

Parameters 1 through 4 correspond to simple (linear) calibration modes. Parameter 5 and 6 relate to the perspective distortion. Parameter 7 relates to optical distortion.

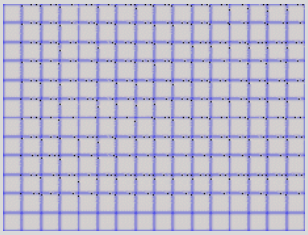
Additionally, the calibration modes, expressed as a combination of options,can be set and get. Refer to ECalibrationMode for the possible options.

- **SensorWidth/SensorHeight**
- **FieldWidth/FieldHeight**

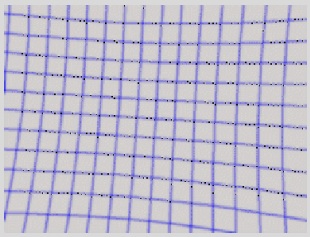
- **Scale**
- **Ratio**
- **CenterX/CenterY**
- **Angle**
- **TiltXAngle/TiltYAngle**
- **DistortionStrength, GetDistortionStrength2**
- **PerspectiveStrength**

Additionally, the calibration mode, expressed as a combination of options, can be accessed via CalibrationModes.

Effect of the Calibration Coefficients



No calibration coefficient



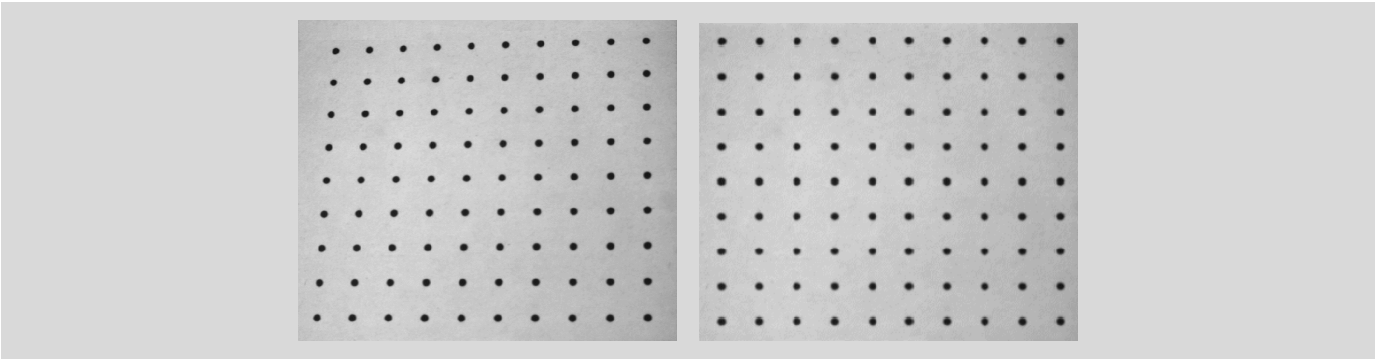
All coefficients combined

Coordinate Transform

When the calibration has been achieved, the EWorldShape object is ready to perform coordinate transform for arbitrary points by means of methods SensorToWorld and WorldToSensor.

Image Unwarping

After the calibration has been performed, the calibration coefficients can be used to compensate for their corresponding effect and thus unwarp the image provided to the calibration function.



Distorted vs. unwarped image

Use Unwarp to perform the operation. Additionally you can build a lookup table before the unwarping to speed up the process, call SetupUnwarp and UnwarpAfterSetup.

EasyOCR

1. What Is EasyOCR?

EasyOCR is a font-dependent printed character reader, based on a template matching algorithm. This library supports gray-level images.

EasyOCR allows training on the font to be recognized by showing sample images of all possible characters. For this reason, it is able to read any kind of short text (serial numbers, labels, ...) such as those found in industrial environments.

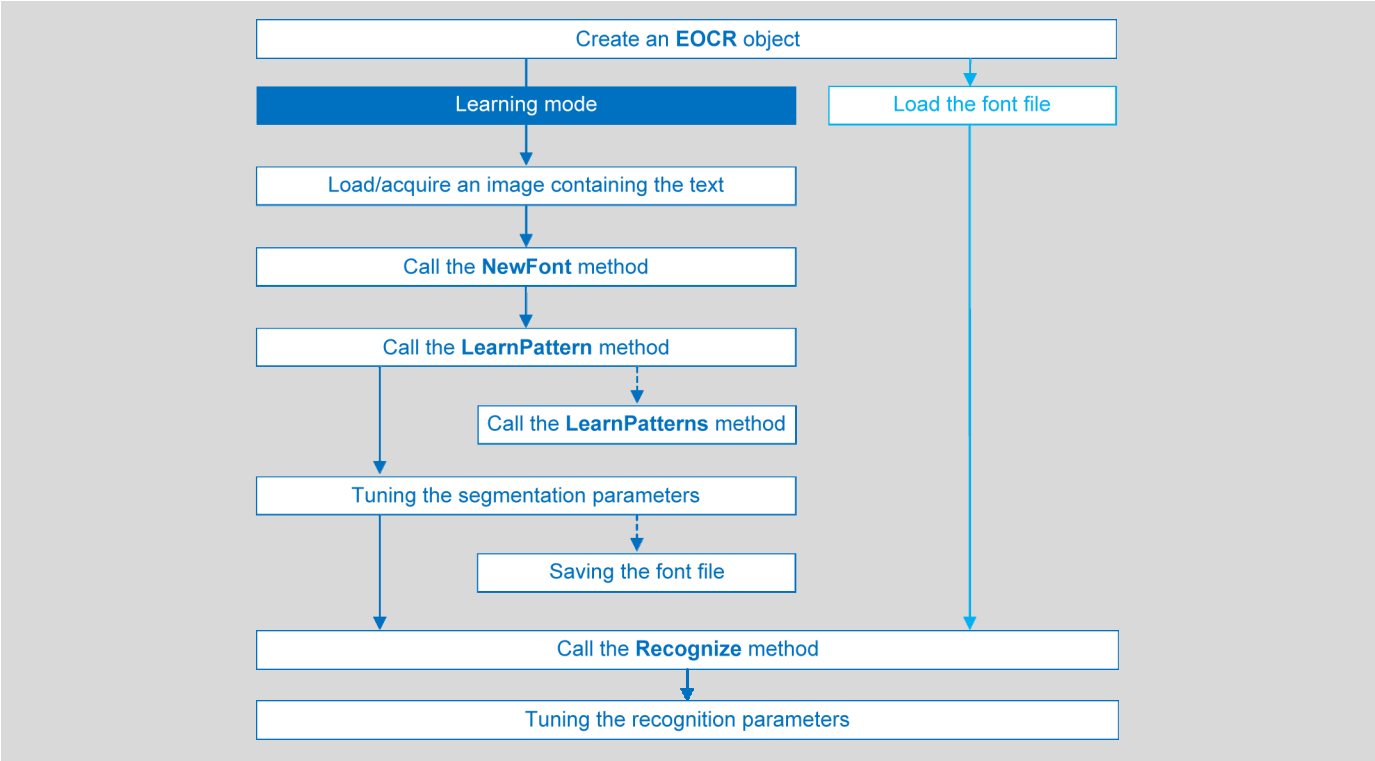
The training phase uses an interactive utility to show samples of the characters and store them in a font file. Such a font editor is available, as an EasyOCR sample program. Moreover, EasyOCR provides the means for you to write your own, custom, font editor.

EasyOCR also provides three standard fonts: OCR-A, OCR-B and Semi. This avoids to collect samples when having to recognize a text using one of these fonts.

EasyOCR uses blob analysis functions to segment the image and extract the characters constituting the text to be read. Blobs are elected as characters based on tunable size and shape criteria. Moreover, EasyOCR is able to deal with characters split into several blobs. When the exact position of the characters in the image is unknown, EasyOCR functions will process the entire image and locate the characters.

2. EasyOCR Workflow

Note that the workflow ONLY shows the generic steps required to work in EasyOCR.



3. Learning Characters and Creating a Font File

EasyOCR is a multi-font character recognition library. This means that EasyOCR functions are able to recognize text printed using any character font, once it has been taught. Practically, during the learning process, characters are presented one by one to the system which analyzes them and builds a database called a **font**.

Only a few data are stored for each new character, they represent distinctive features of the character's shape. This small database may be saved to disk and restored when needed.

During the learning process, each pattern gets an associated numerical value call its code (usually its ASCII code). A pattern also belongs to a character class. The class information can be used later during the recognition process to restrict the set of patterns to compare to.

The learning process is achieved by means of the following calls:

1. **NewFont** clears the current font.
2. **LearnPattern** or **LearnPatterns** add the patterns from the source image to the font. Patterns are ordered by their index value, as assigned by the **FindAllChars** process.
3. Possibly, an erroneously learnt pattern might be removed from the font using **RemovePattern**.
4. **Save** writes the contents of the font to a disk file.

The patterns in a font are stored as a small array of pixels, by default 5 pixels wide and 9 pixels high. This default size can be changed before learning, using parameters PatternWidth and PatternHeight.

Note. The characters used through the learning phase should be as representative as possible of those to be read. Furthermore, the segmentation parameters (color, spacing, ...) *must* be the same for both learning and recognition process.

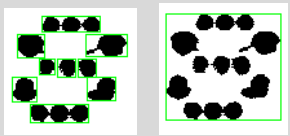
During the learning process, characters are presented one by one to the system, which analyzes them and builds a database called a *font*. Only few data is stored for each new character, it represents distinctive features of the character's shape. This small database may be saved to disk and restored when needed.

For learning as well as for recognition, EasyOCR will segment the characters, i.e. locate the characters and determine their bounding box. This is done by means of blob analysis (thresholding followed by a grouping of pixels of the same color, as is done by EasyObject). After blobs have been found, they can be filtered out to remove unwanted features (small blobs of noise, large extraneous objects, ...).

A second segmentation phase will consider the blobs and form characters out of them, using one of two segmentation modes:

- In the simplest case, one blob always corresponds to one character (the **keep objects** mode).
- In the other one, a built-in algorithm takes decisions to group the blobs in such a way that the characters remain within nominal size (the **repaste objects** mode). This is very convenient when the characters appear to be broken because of marking degradation or poor thresholding, and when characters are made up of several parts.

Note. Furthermore, the segmentation parameters (color, spacing, ...) must be the same for both learning and recognition process



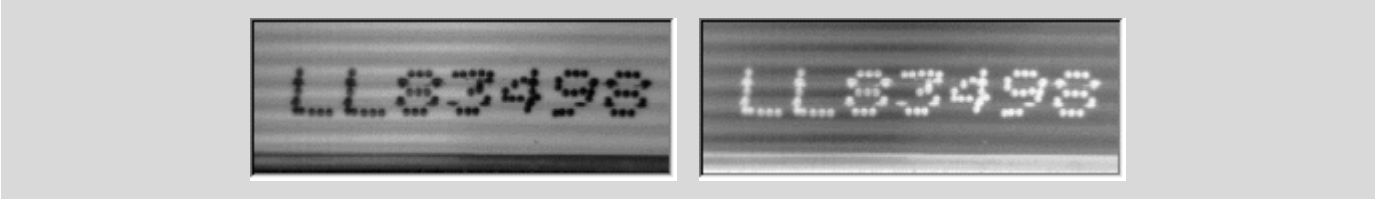
Character segmentation by blob grouping

When a blob is too large to be considered a single character, it can be split automatically, using CutLargeChars.

3.1 Segmentation Parameters

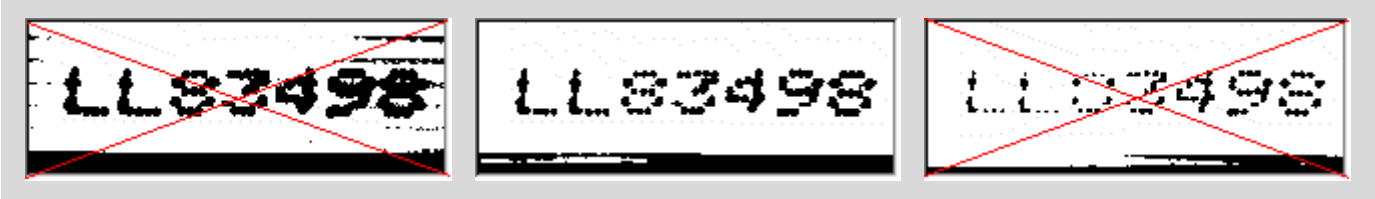
The following parameters are used for blob segmentation:

- **TextColor**: The text should be dark on a lighter background, or conversely. If necessary a threshold can be applied to obtain such a result.



Dark on lighter contrast, and light on darker contrast

- **Threshold**: Threshold value used to separate the text from the background. The threshold value should be chosen such that the characters are well separated. The correct thresholding level must be chosen with care. For black on white characters, a too high value will thicken the characters and possibly make them merge together or merge with other objects in the field of view. A too small value will thin the characters and make parts become invisible and character split. If the lighting conditions are too variable, automatic thresholding should be considered.



Threshold adjustment

The following size parameters are used for character segmentation:

- **NoiseArea**: if a blob has an area smaller than this value, it is considered as noise and discarded. The noise area should be chosen such that the noise blobs are discarded but small character features are preserved (for instance, the dot over an "i" letter).
- **MaxCharWidth**, **MaxCharHeight**: if a blob does not fit within a rectangle with these dimensions, it is not considered as a possible character (too large) and it is discarded. Furthermore, if several blobs fit in a rectangle with these dimensions, they are grouped together, forming a single character. The outer rectangle size should be chosen such that it can contain the largest character from the font, enlarged by a small safety margin.
- **MinCharWidth**, **MinCharHeight**: if a blob or a group of blobs does fit in a rectangle with these dimensions, it is not considered as a possible character (too small) and it is discarded. The inner rectangle size should be chosen such that it is contained in the smallest character from the font, shrunk by a small safety margin.

- **CharSpacing**: if two blobs are separated by a vertical gap wider than this value, they are considered to belong to different characters. This feature is useful to avoid the grouping of thin characters that would fit in the outer rectangle. Its value should be set to the width of the smallest gap between adjacent letters. If it is set to a large value (larger than **MaxCharWidth**), it has no effect.

After the character segmentation phase, EasyOCR will process the character image, to extract relevant features and store them once for all in the font file. In particular, this involves size normalization, i.e. stretching the character so that it fits in a bounding box of known size. This allows size-independent matching.

4. Recognizing Characters

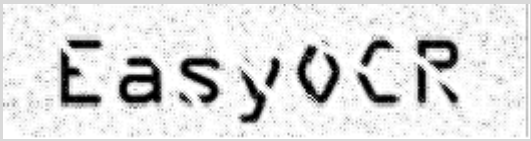
EasyOCR follows a few steps in the recognition process.

First, **the image is segmented** and decomposed into **objects** or blobs (connected components), in the same way as EasyObject does.

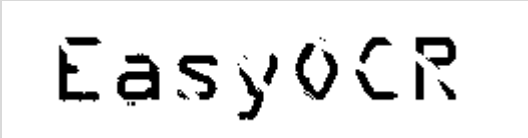
Note. The segmentation parameters (color, spacing, ...) must be the same for both learning and recognition process.

Then the objects are filtered according to the size and possibly grouped together ("repasted") to form distinct **characters**. This is called **character isolation** or segmentation. (This step can be bypassed when the exact position of the characters is known beforehand). Blobs are elected and grouped to form characters based on tunable size and shape criteria. EasyOCR is able to deal with characters split into several blobs. Even if the character size is not the same as at learning time, EasyOCR can retrieve them and match them seamlessly against the trained samples. However, the segmentation parameters (color, spacing, ...) should be the same than those used during the learning phase.

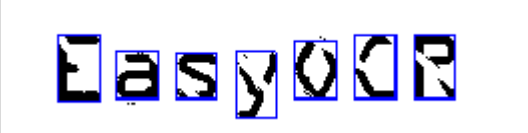
The characters are compared to a set of patterns, called a **font**. A character is recognized by finding the best match between a character and a pattern in the font. After the character has been located, it is normalized in size (stretched to fit in a predefined rectangle) for matching. The normalized character is compared to each normalized template in the font database and the best matches are returned.



Raw image



After objects segmentation



After character isolation

After character recognition

Character Classes and Filtering

Additionally to the recognition process, a **character set filter** can be used to restrict the range of characters from the font to be compared to the characters in question. For instance, assume you know a marking always consists of two uppercase letters followed by five digits, the last of which is always even. In such a case, it is possible to assign each character in the font to one of the following three classes: uppercase letters, even digits, odd digits. Then, at recognition time, the character filter will allow the following classes: twice uppercase, four times even digit or odd digit, once even digit.

To achieve this effect, you must assign each character a class at learn time. 32 different classes can be defined. Then, at recognition time, you describe the text content by listing the expected classes for all character positions.

Using character filter has positive effects on both the recognition reliability and running time.

The recognition process is achieved by means of the following calls:

1. **Load**: reads a pre-recorded font from a disk file.
2. **BuildObjects**: segments the image to locate the characters.
3. **FindAllChars**: selects the objects considered as characters and sorts them from left to right.
4. **ReadText**: performs the matching.

Steps 2 to 4 can be repeated at will to process other images or ROIs. The Recognize method can be used as well.

If additional information, such as the geometric position of the detected characters, is required, some query functions may be used: CharGetOrgX, CharGetOrgY, CharGetWidth, CharGetHeight, ...

4.1 Recognition Parameters

The recognition process is governed by a few parameters that need to be fine tuned to obtain the most reliable results.

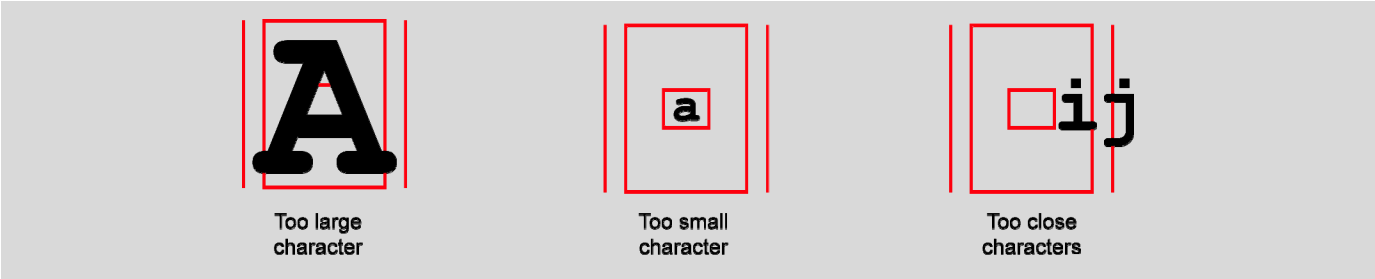
The following two parameters are used during segmentation:

- **TextColor**: black text on a white background, or conversely, with or without thresholding.
- **Threshold**: threshold value used to separate the text from the background. The threshold value should be chosen such that the characters are well separated.

The following geometric parameters are used during character isolation:

- **RemoveBorder**: most of the time, blobs that are found along the image/ROI edges are spurious and cannot be exploited for character recognition. By default, they are discarded for character isolation.
- **NoiseArea**: if a blob has an area smaller than this value, it is considered as noise and discarded. The noise area should be chosen such that the noise blobs are discarded but small character features are preserved (f.i., the dot over an "i" letter).

- **MaxCharWidth**, **MaxCharHeight**: if a blob does not fit within a rectangle with these dimensions, it is not considered as a possible character (too large) and is discarded. Furthermore, if several blobs fit in a rectangle with these dimensions, they are grouped together, forming a single character. The outer rectangle size should be chosen such that it can contain the largest character from the font, enlarged by a small safety margin.
- **MinCharWidth**, **MinCharHeight**: if a blob or a group of blobs does fit in a rectangle with these dimensions, it is not considered as a possible character (too small) and is discarded. The inner rectangle size should be chosen such that it is contained in the smallest character from the font, shrunk by a small safety margin.
- **RemoveNarrowOrFlat**: by default, small characters are discarded when they both narrow and flat. This behavior can be changed so that they are discarded when either condition is met.
- **CharSpacing**: if two blobs are separated by a vertical gap wider than this value, they are considered to belong to different characters. This feature is useful to avoid the grouping of thin characters that would fit in the outer rectangle. Its value should be set to the width of the smallest gap between adjacent letters. If it is set to a large value (larger than **MaxCharWidth**), it has no effect.
- **CutLargeChars**: when a blob or grouping of blobs is larger than the maximum allowed width, it is considered as clutter and discarded. When the **CutLargeChars** mode is enabled, the blob is split in as many parts as necessary to fit. This is an attempt to separate touching characters.
- **RelativeSpacing**: when the **CutLargeChars** mode is enabled, setting this value allows specifying the amount of white space that should be inserted between the split parts of the blobs.



Invalid recognition settings

If the character isolation process is bypassed, the following members must be used to specify the known locations of the characters:

- **AddChar** and **EmptyChars**.

Remarks

1. When objects are larger than **MaxCharWidth**, they can be split into as may parts as needed using vertical cutting lines. This behavior is corresponds to the option **CutLargeChars**.
2. A font file stores the learnt patterns, as well as the following parameters: **NoiseArea**, **MaxCharWidth**, **MaxCharHeight**, **MinCharWidth**, **MinCharHeight**, **CharSpacing**, **TextColor**. When a font file is loaded, these parameters are set to the recorded value.
3. The patterns in a font are stored as a small array of pixels, by default 5 pixels wide and 9 pixels high. This default size can be changed before learning, using parameters **PatternWidth** and **PatternHeight**.
4. When the character and learnt pattern are compared, the comparison can be made sensitive to a discrepancy between the aspect ratios (height over width). This enforces the difference between narrow and wide characters. This corresponds to the option **CompareAspectRatio.**
5. Two character isolation modes are available, as defined by **ESegmentationMode**. The characters isolated using either method may not be the same:

- **Keep objects** mode: in this mode, a character is made up of a single blob; no attempt is made to group blobs corresponding to damaged characters. For this reason, non connected characters cannot be handled. Small features such as accents and dots will be discarded by the inner rectangle criterion.
- **Repaste objects** mode: the blobs are grouped as long as they fit in the outer rectangle and are not separated by a vertical gap. For this reason, the accents and dots will be preserved.

Tuning the Recognition Parameters

When segmentation behaves correctly, few parameters have to be adjusted for the recognition process:

- **CompareAspectRatio**: when this setting is on, EasyOCR becomes less tolerant on character sizes and takes into account the measured aspect ratio. Using this mode improves the recognition performance when characters look like each other after size normalization.
- Filtering the characters (in the **ReadText** method), can be used if the marking structure is fixed.

When objects are larger than the MaxCharWidth property, they can be split into as many parts as needed, using vertical cutting lines.

A font file stores the learnt patterns, as well as the following parameters: NoiseArea, MaxCharWidth, MaxCharHeight, MinCharWidth, MinCharHeight, CharSpacing, TextColor. When a font file is loaded, these parameters are set to the recorded value.

The patterns in a font are stored as a small array of pixels, by default 5 pixels wide and 9 pixels high. This default size can be changed before learning using PatternWidth and PatternHeight properties.

When the character and learnt pattern are compared (enabling the CompareAspectRatio property), the comparison can be made sensitive to a discrepancy between the aspect ratios (height over width). This enforces the difference between narrow and wide characters.

Two character isolation modes are available. The characters isolated using either method may not be the same:

- **Keep objects** mode: in this mode, a character is made up of a single blob; no attempt is made to group blobs corresponding to damaged characters. For this reason, non-connected characters cannot be handled. Small features such as accents and dots will be discarded by the inner rectangle criterion.
- **Repaste objects** mode: the blobs are grouped as long as they fit in the outer rectangle and are not separated by a vertical gap. For this reason, the accents and dots will be preserved.

EasyOCV

1. What Is EasyOCV?

Optical Character Verification, also known as *mark inspection*, is an inspection technique by which a geometric pattern, a *sample*, is checked for similarity with a predefined model, called a *template*. For instance, when a part number is printed on an integrated circuit package, one may need to control the quality of the printing: correct placement with respect to the component body, sufficient contrast, good shaping of the characters, absence of inking defects... All these factors may affect readability of the printing. EasyOCV works by comparing the template and sample images while taking into account relative displacement of the constituent parts.

OCV first requires training on the particular marking in question. During this phase, a good quality template is presented to the system. An interactive utility allows defining the structure of the template and related acceptance criteria, in details. The template learnt can be saved for later use.

When a template model is available, inspection may take place. The sample image is processed and the system first locates the marking, allowing it to be translated, rotated and re-scaled or even sheared with respect to the template. After location, geometric comparison is performed and some matching scores, called quality indicators, are computed. When these fall outside of given acceptance intervals, a defect warning is reported.

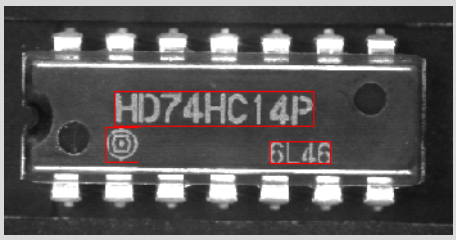
EasyOCV is the Open eVision component that provides all means to quickly build OCV applications.

In order for EasyOCV to provide maximum flexibility, several working modes are available. Anyway, for ease of reading, the more useful features are presented first, while others are highlighted as being advanced. Also, one can distinguish between the functionalities related to the learning phase (model definition) and to the inspection phase.

Note. In a simple application, the learning phase can be skipped by providing a ready-made template model learnt separately and stored in a file. Only the inspection phase has to be implemented. To generate the required model file, Open eVision Studio provides a comprehensive model editor.

2. Template Model Structure

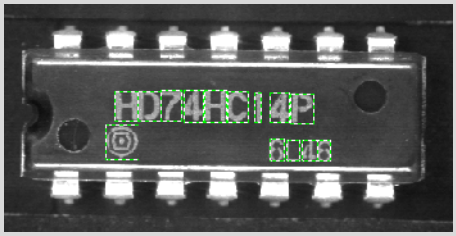
It is important to understand how the template model is organized. Inspection takes place in a rectangular ROI. In this region, several pieces of text can be defined. A piece of text is a set of characters rigidly tight to each other. In practice, all the marks that are printed simultaneously should be considered a single text. During inspection, the different texts are allowed to move independently of each other.



Texts are sets of characters

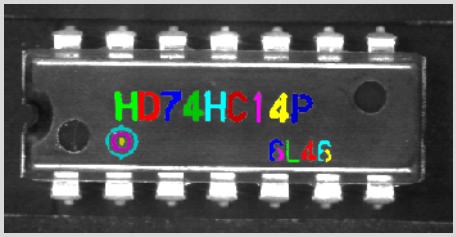
Note. A text item needs not to be defined for every row of text or every word of a sentence. The true goal of defining more than one text is to allow several parts of the marking to be inspected separately, i.e. have different inspection parameters and quality acceptance criteria. In most cases, just defining a single piece of text is enough.

A character is the smallest part of a marking that can be inspected in isolation. The characters are attached to a text but a small displacement from their nominal position can be allowed and measured.



Characters are sets of blobs

A character can be composed of several blobs grouped together. A blob is a connected component of a thresholded image, as handled by EasyObject (legacy).



Blobs are the "atomic" constituents of markings

Remember this top-down organization: the marking is a set of texts, each being a set of characters, each being a set of blobs. In typical applications, a single text is used; there is usually a one-to-one correspondence between characters and blobs, unless accented letters or logos appear. Also note that the characters may have any shape and size and need not to be members of an existing font.

3. Inspection Process

Inspection is the combined process of locating the components of the model and assessing their similarity in the template and sample. Inspection takes place in an ROI that does not need to be the same as that at learning time.

The inspection process involves two operations.

1. First, the ROI is scanned and the best matching is found between the reference template model and the sample image, using all of the desired displacement degrees of freedom, as described below. This is the **location** process.
2. Then, every sample character is compared to the corresponding template character and the quality indicators are computed. The results are collated to result in global quality indicators for the texts. This is the **scoring** process.

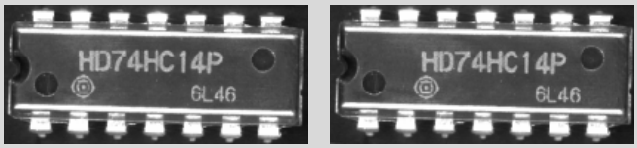
When all scores are obtained, they are tested for inclusion in the acceptance intervals. When a value falls outside an allowed range, the corresponding character/text is flagged and a diagnostic code is generated. Also, global diagnostics summarizing all text and character defects are issued.

For most applications, text-level inspection —i.e. assessing text quality globally— is sufficient. If finer details are wanted, character-level inspection —i.e. assessing the quality of each individual character— is also possible, though more complex.

3.1 Degrees of Freedom for Location

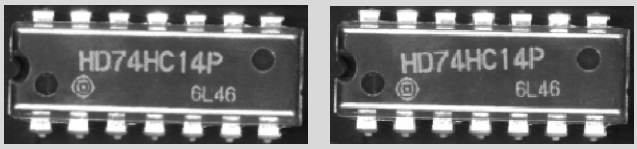
During inspection, the best match between the template model and the sample image is found during a search phase called *location*. The following degrees of freedom can be allowed and their amplitude specified:

- **Text translation**: all texts can be moved horizontally (**ShiftX**) and vertically (**ShiftY**) in a specified range.



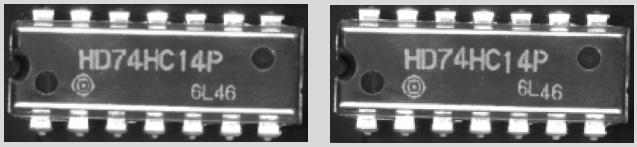
Text translation

- **Text skewing**: all texts can be rotated about the center of their bounding box. The corresponding parameter is an angle called **Skew**.



Text skewing

- **Character translation**: after the text position has been adjusted using the aforementioned degrees of freedom, the individual characters can again be moved horizontally and vertically with respect to their nominal position. The corresponding parameters are also called **ShiftX** and **ShiftY**.

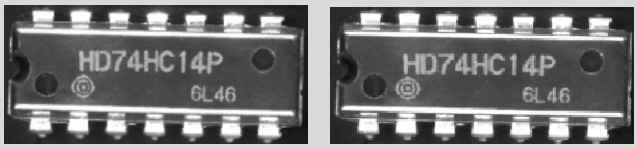


Character translation

Advanced Degrees of Freedom for Location

The former degrees of freedom are often met in practical applications. The next ones have been implemented for completeness but should be used only when there are good reasons to do so.

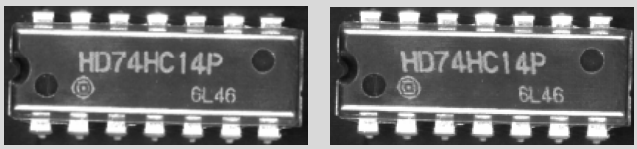
- **Text X/Y-scaling**: all texts can be re-scaled horizontally and vertically, while the center of their bounding box remains fixed. The corresponding parameters are ratios called **ScaleX** and **ScaleY**. Re-scaling can be isotropic (both scale factors remaining identical) or anisotropic.



Anisotropic text scaling

Note. By default, text scaling is supposed to be isotropic (scaling along X- and Y-axes are proportional). If you think your application might generate anisotropic scaling for a text, you must set the **IsotropicScaling** parameters of the **EOCVText** corresponding object to **FALSE** before inspecting. Working with isotropic scaling results in an effective time saving during inspection.

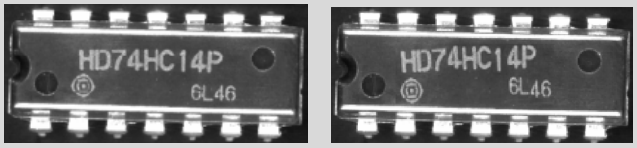
- **Text shearing**: all texts can be sheared, i.e. become italicized, while the center of their bounding box remains fixed. The corresponding parameter is an angle called **Shear**.



Text shearing

The purpose of these degrees of freedom is to compensate for misalignment and distortion: translation and skewing often arise because of non repeatability of the parts placement, f.i. because of imperfect mechanical guiding. Character translation may be due to a lack of stiffness of the printing device, but also provides a convenient way to cope with small amplitude skew or scale changes. Isotropic changes in scaling occur when lens magnification (or scene/lens distance) is changed. Anisotropic scaling and shearing are much less frequent and may occur because of relative movement between the marking device and the medium during printing.

Since the addition of a degree of freedom increases the running time, it is advisable to use them sparingly. In many cases, text and character translation are sufficient. When large amplitude skewing is possible, text translation + skewing can be used. Care must be exercised when combining the other degrees of freedom.



All degrees of freedom combined

Specifying the Degrees of Freedom

The degrees of freedom for location are specified with respect to the *nominal position*, that is the position at learning time. The text centers are remembered relative to the template image/ROI center, the skew and shear angles are set to **0** and scale factors to **1**. The characters are remembered relative to the corresponding text center.

The parameters are allowed to vary in an acceptance interval, specified as a **Tolerance** around a **Bias** value: the range bounds are computed as **Bias ± Tolerance**. When a bias parameter is set to **0**, the range is centered around the nominal value. When a tolerance parameter is set to **0**, the corresponding degree of freedom is only tried using the bias value.

The number of positions tried for each degree of freedom is specified differently for translation than for the other transforms:

- **Translation**: in principle, every integer value in the ranges **ShiftXBias ± ShiftXTolerance** and **ShiftYBias ± ShiftYTolerance** is tried. However, for efficiency reasons, the **ShiftXStride** and **ShiftYStride** parameters can be set to a value larger than **1**, so that a gross location pass with the specified stride is followed by a finer one with unit stride. (The expected speed-up is on the order of the square of the **Stride** parameter.) Anyway, choosing too large a value may cause mismatches when local maxima are present. (A value on the order of a fraction of the character size is recommended.)
- **Skewing, scaling and shearing**: the **SkewCount**, **ScaleXCount**, **ScaleYCount** and **ShearCount** parameters indicate the exact number of values tried for each degree of freedom in the allowed ranges, bounds included. The execution time increases as the product of these counts. When a degree of freedom is not used, its count must be left as **1**.

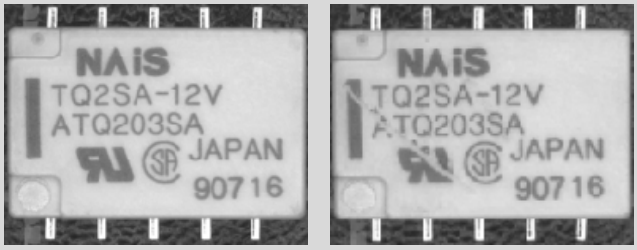
Note. When the **SkewCount** value is set to **0**, EasyOCV automatically chooses an appropriate count value.

3.2 Image Comparison with EChecker

EChecker is a part of the EasyOCV component. It is an inspection tool based on image comparison, i.e. finding visible differences between representative sample items (parts free of defects), and another sample to be inspected.

It starts by a learning step during which a plurality of images is provided and serve as a reference. The images in this set are averaged to provide a noise-free template. Then, for each image to be inspected, a comparison is made pixel by pixel between the sample and the template, and significant differences are highlighted. The result of the comparison is a defect map. All the standard tools of blob analysis (as provided by EasyObject (legacy)) can be used to locate those defects and qualify them in terms of extent, orientation, lightness, ...

The EChecker object is ideally suited when explicit inspection is wanted. It assumes that the inspected items are rigid (their shape does not vary) and that the illumination remains uniform. This ensures that the visual appearance of the image is repeatable enough, so that point to point image comparison makes sense. Anyway, EChecker is capable of dealing with movement of the inspected part in the field of view and global illumination changes.



Reference image vs. sample image with defects

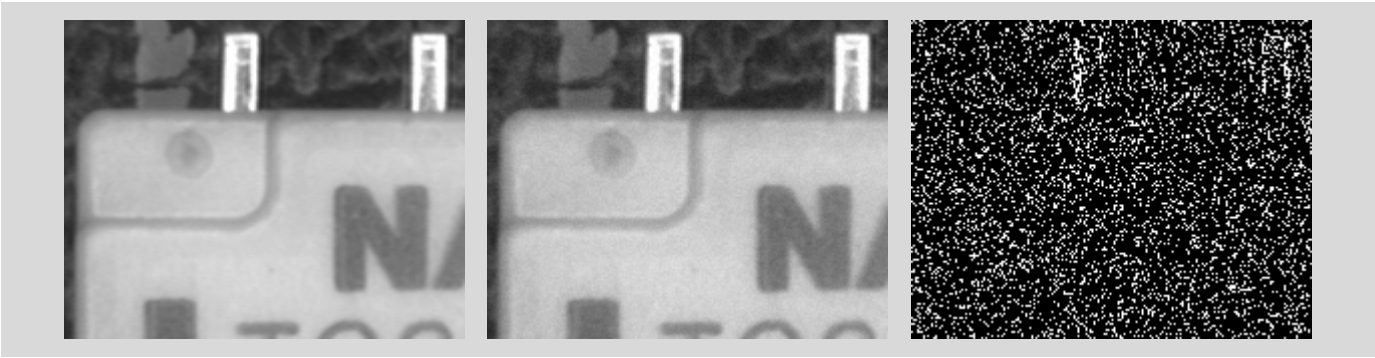
Image Comparison

Image comparison is a straightforward and natural inspection technique. Assume you have a few samples of a part to be checked. In principle, all of them will look alike when placed in similar conditions. If one of them has visible defects, on the other hand, its image will significantly differ at places from the other ones.

Comparing two images is achieved by subtracting them and taking the absolute value of the result (See Arithmetic and Logic functions in EasyImage), so that dark and light spots are seen. Where images differ, the result will be non zero. Thresholding the difference image can enhance the non-zero pixels.

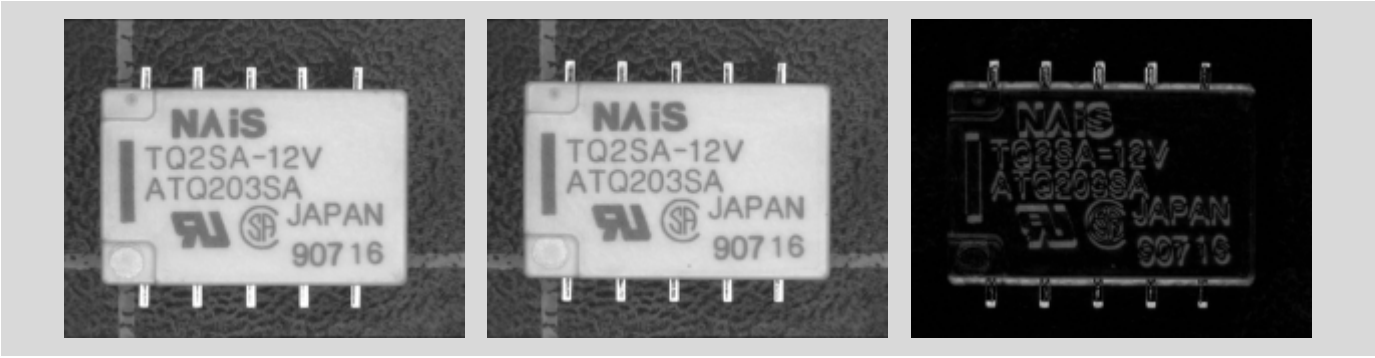
In all practical situations, this simple approach badly fails because of three shortcomings:

- Acceptable variations are always present: image acquisition inevitably introduces some amount of noise. Such noise makes consecutive images of the same scene differ by a few gray-level values in an unpredictable way. Also, slight variations in color or shape between seemingly identical parts always occur. These sources of variations are unavoidable, and some tolerance has to be introduced. Comparison for strict equality is a too naive approach.



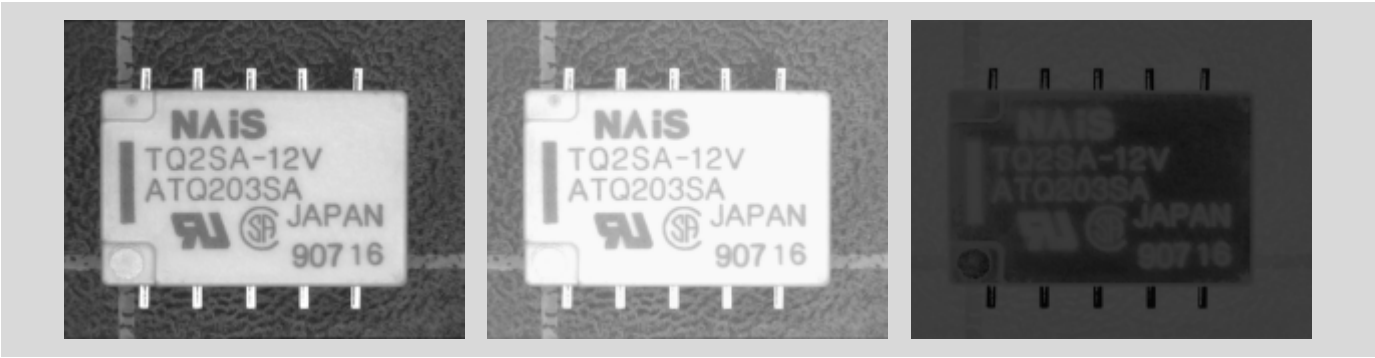
Comparing two noisy images for strict equality.
Black pixels of the result image are non equal pixels

- Most of the time, placement of the inspected parts is not totally repeatable because of mechanical play and imperfect fixture. When this occurs, misalignment results so that the pixels to be compared do not correspond anymore to the same areas on the inspected parts. Comparing then makes little sense. The resulting effect is especially noticeable in the vicinity of edges, where the gray-level values change rapidly.



Comparing misaligned images

- If the lighting change, either globally (uniformly over the whole scene) or locally (differently at different places), the visual appearance of the parts does change even though the parts themselves do not. (Keep in mind that the image formation process is the result of the interaction of the incident light and the object surfaces.) You inspect at the same time a part and the light source that illuminates it. There is no simple way to separate both contributions.



Comparing images in different lighting conditions

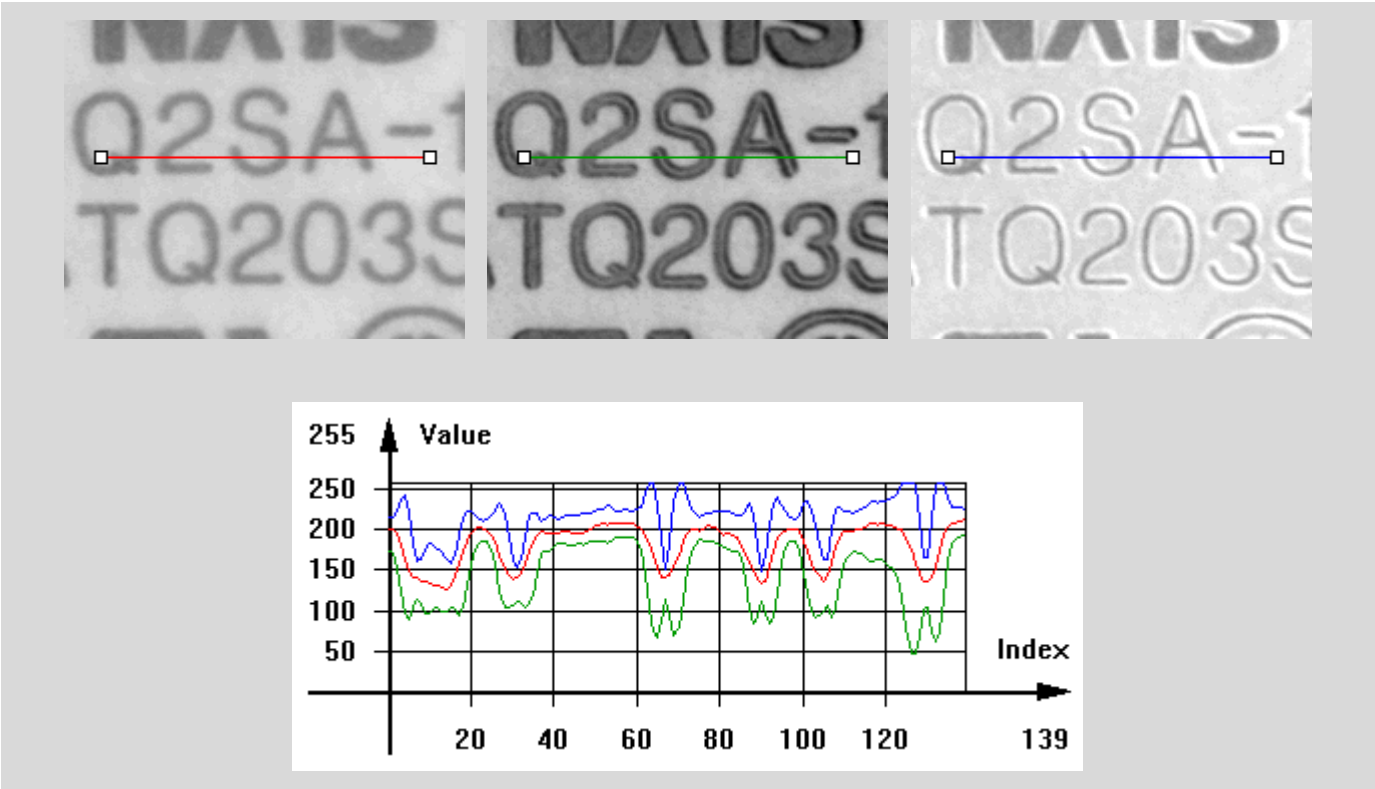
By a clever combination of pre-processing operations applied to a single reference image, one can deal, to some extent, with these shortcomings. Security margins can be introduced both in terms of gray-level values and pixel positions. These allow tolerating noise, slight misalignment and slight illumination changes. Anyway, choosing appropriate values for the security margins is not a trivial task and can only be achieved by a difficult trial and error process.

The EChecker tool provides a better answer to these three issues:

- statistical training,
- realignment,
- gray-level normalization.

Statistical Training

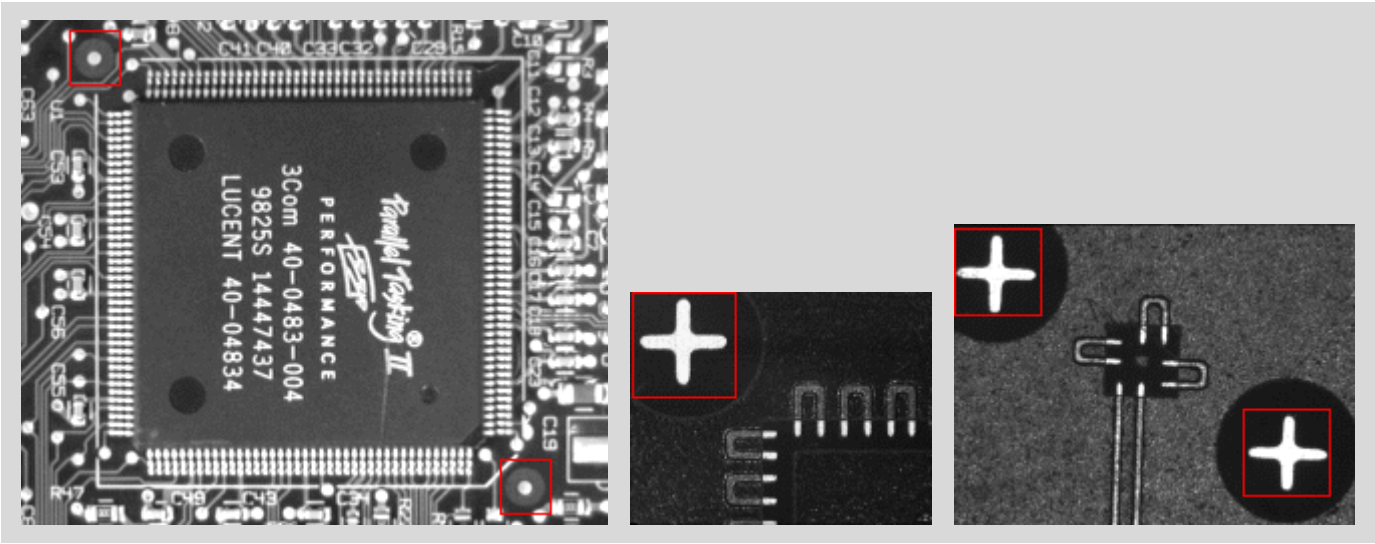
Rather than using a single reference image, considering several images altogether allows assessing the normal gray-level variations and determining an acceptance interval for the gray-level values. When grabbing consecutive images of the same part without any change (static test), the distribution of the gray-level values simply corresponds to the distribution of noise. Grabbing consecutive images of different defect-free parts reveals the variations due to the parts themselves. EChecker combines a set of images and automatically determines the range of acceptable values at each pixel.



Accepted gray-level ranges

Realignment

The possible movement of the inspected part can be measured in terms of translation, rotation and/or scaling by means of pattern matching (as described in the EasyMatch chapter). For instance, when fiducial marks are available, they can be used as landmarks for accurate and repeatable location. EChecker hosts one or two matching contexts internally. A single matching pattern easily handles translation. Two matching patterns provide better accuracy for rotation measurement. When the patterns have been located, the image is realigned so that the inspected part is brought to the same position as in the reference images.



Realignment using fiducial marks

Gray-Level Normalization

As explained above, separating a local change in the illumination from a change in the surface reflectance (color) is not possible. Anyway, one can compensate a global change in intensity and/or contrast by means of gray-level normalization (see Normalization in the EasyImage section). The idea is that by adjusting constant gain and offset parameters, the intensity and contrast of the inspected and reference images are made equal, canceling the change that occurred.

4. EChecker Concept

4.1 What Is EChecker?

EChecker requires two distinct processing steps to be implemented: the training step consists in presenting the set of reference images and pre-processing them to compute the acceptance ranges at each pixel and store them in two threshold images for use with EasyObject (legacy). The training can be done once for all and the results archived in a model file for later use.

The inspection step handles a sample image, realigns it by means of the fiducial(s) in use, normalizes the gray-level values and detects those pixels that fall out of the acceptance intervals. Further handling of the defective pixels is performed by means of the standard EasyObject (legacy) functions.

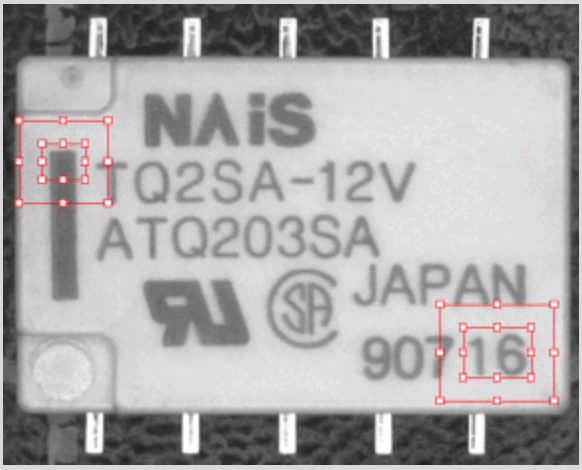
The following sections present the relevant API functions for use in the training and inspection steps. These two operations are totally independent and can be programmed in separate applications.

4.2 Training with EChecker

Training uses a set of reference images. (Two modes of operations are provided, depending on whether the images have been stored to disk or are acquired and processed on the fly.)

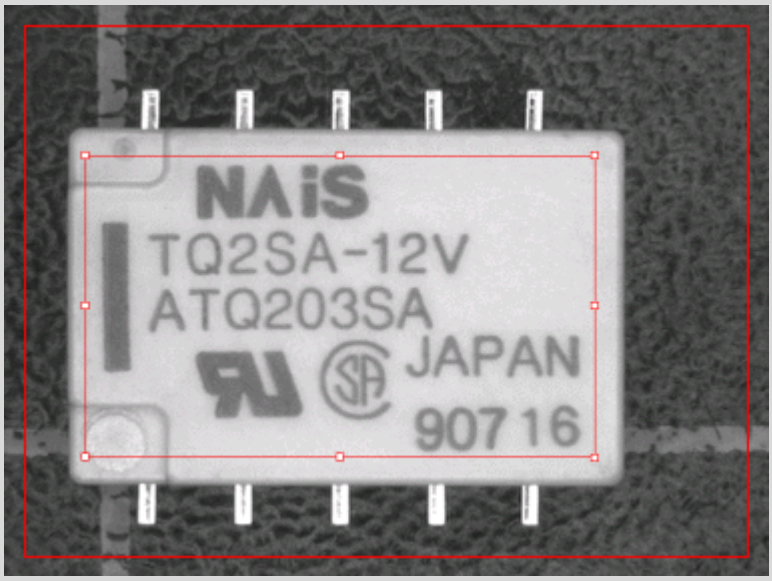
To define a model, several operations are performed, in this order:

- On the first reference image, one or two ROIs are placed to define the location patterns (fiducial marks or landmarks). These ROIs are surrounded by two other ones defining the possible movement freedom. They will be used as search areas for pattern matching. Placement of these ROIs can be done interactively by means of dragging handles. **EChecker** manages the dragging operations.



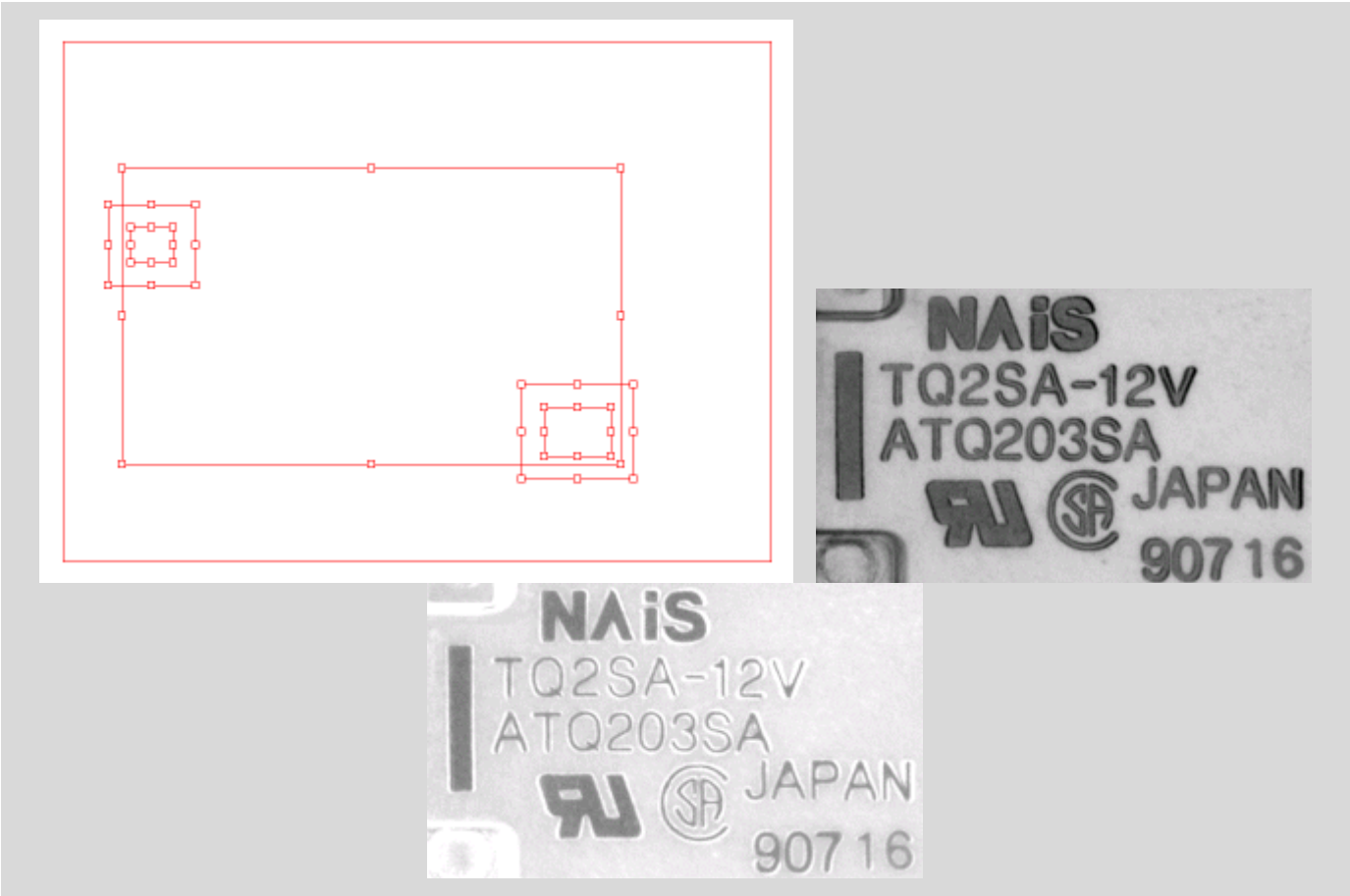
Pattern ROIs and search ROIs

- If the training set resides on disk, the list of file names can be registered at that time. This list can be used later to execute learning in a single command. This is called *batch learning*, as opposed to *on-the-fly learning*.
- Another ROI is defined to delimit the area to be inspected. It is important to adjust this area so that it only includes pixels belonging to the rigid part at hand (that move with the fiducial marks), and none of the background.



Inspected ROI

- All images are processed and undergo realignment, gray-level normalization and statistical analysis.
- Eventually, the low and high threshold images are computed. The user has the possibility to strengthen or loosen the acceptance intervals by means of a global tolerance parameter.
- The model can then be saved to a single file including all relevant information, i.e. placement of the various ROIs, fiducial pattern images, gray-level normalization parameters and the threshold images.



Components of an EChecker model

4.3 Defining the Pattern and Search ROIs

The pattern ROIs can be interactively located by means of handles, as usual. Member function Draw performs the graphical rendering of the required ROIs and handles. Member function HitTest detects the presence of the cursor over one of the handles. Member function Drag allows moving the corresponding handle. These functions operate on all three kinds of ROI (pattern, search and inspected).

You will quickly notice that dragging a pattern ROI will cause the corresponding search ROI to adjust itself automatically so that the search tolerances remain constant. Dragging a search ROI causes its size to adjust symmetrically with respect to the pattern. Adjusting a search area also sets the inspected ROI to the largest available space in the image.

4.4 Trying the Reference Images

After the alignment ROIs have been set, they should be checked for reliability of fiducial location. The best way is to load the training images, display them and locate the patterns in them, by means of member function Register. If location fails, different corrective actions can be taken, depending on the problem:

- The choice of the pattern is poor. Define other ROIs with more stable contents.
- The search areas are too tight, so that occurrences of the pattern are found along edges. Enlarge the search area.
- The image is insufficiently representative (it has defects). It is better to withdraw it from the learning set.

Note. The first call to member **Register** will invoke the **Learn** members of the alignment EasyMatch contexts, i.e. training on the patterns is achieved. Unless member **Learn(ELearningMode_Reset)** is called, these patterns will be used for all subsequent alignment operations. The first image to be used serves both as a reference for defining the alignment pattern and contrast measurement. It is called the *mother image*.

EChecker provides a convenient way to keep track of the learning images when they are saved to files on disk: the EChecker object can keep a list of file pathnames. Every time a file is successfully loaded and is accepted as a reference image, its name can be added to the list. Later, all files can be processed in a single command.

4.5 Defining the Inspected ROI

Note. The inspected ROI must be positioned on the mother image, otherwise misalignment can result.

The inspected ROI can be interactively positioned in the same way the pattern and search areas are. This ROI can be set at the same time as the others. Anyway, changing the search tolerance will reset the inspected ROI to the largest available area.

4.6 Learning Passes

After the ROI placement and pattern learning have been performed (**Register** operations), training still requires two passes:

- The **"average" pass** is needed to compute an ideal, noise-free, image that reveals the central tendency of the part image.
- The **"deviation" pass** measures variations around the average image.

In principle, the images shown during those two passes should be the same. Anyway, if you do not want to archive them at all, two distinct sets of images can be used (on-the-fly learning). These sets need not even be of the same size.

A learning set size of at least 16 images is recommended.

Computing the Average Pixel Values

For each image, realign and normalize (Register). If the operation is successful (good pattern location), call Learn(ELearningMode_Average) for immediate processing (on-the-fly learning), or AddPathName for deferred processing (batch learning).

Computing the Pixel Values Deviations

For each image, realign and normalize (Register). If the operation was successful, call Learn(ELearningMode_RmsDeviation) or Learn(ELearningMode_AbsDeviation) for immediate processing.

Alternatively, calling BatchLearn will perform the two required passes for all images added to the file list.

Two rules for computing the deviations are available. The first one, **Rms**, enhances the large deviations. The other, **Abs**, is more robust and is recommended in most circumstances.

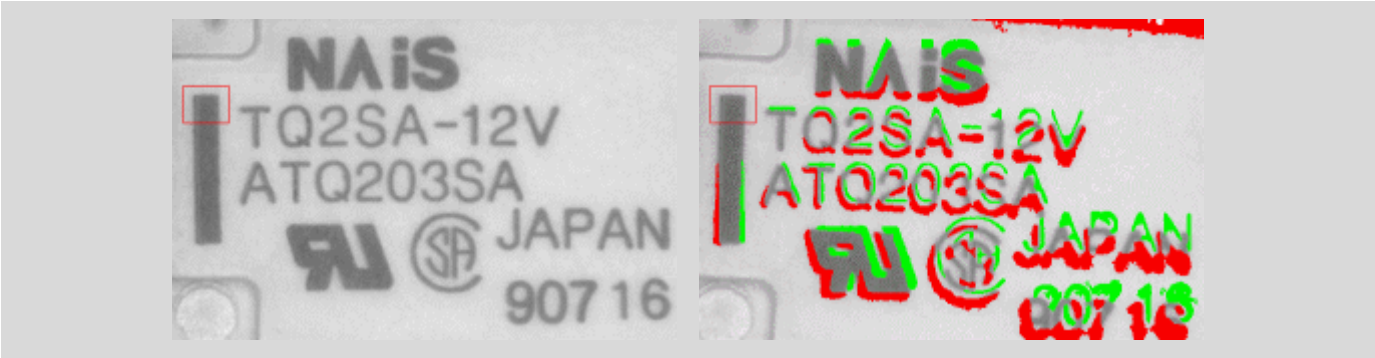
4.7 Building the Threshold Images

Call Learn(ELearningMode_Ready). Property RelativeTolerance can be adjusted at that time to adjust the acceptance ranges.

4.8 Tuning the Learning Process

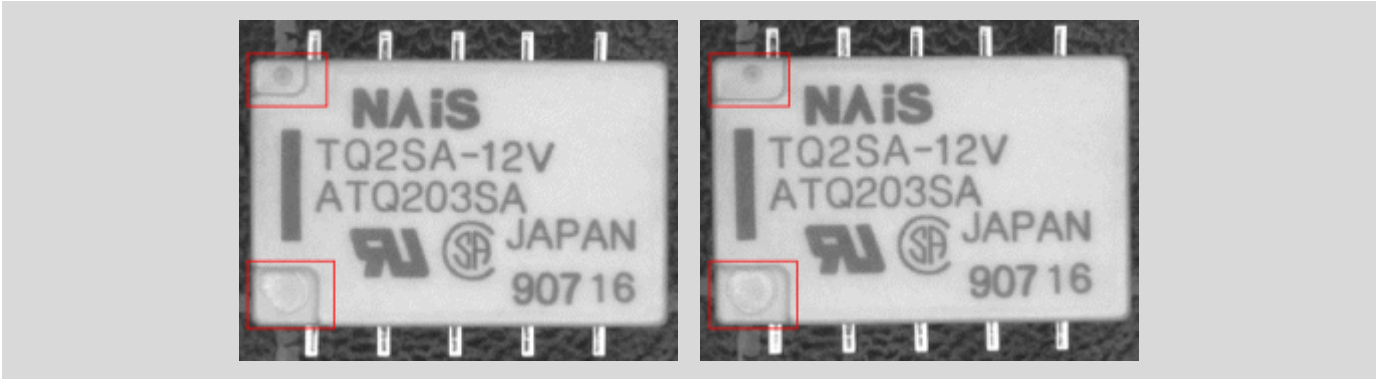
When designing an EChecker model, a few early decisions must be considered with care, to ensure success:

- The placement repeatability must be evaluated. Often, when the part is handled by a conveyor or travels along a guide, only translation can occur. When the part is dropped randomly or not fixtured tightly, rotation also happens. When the lens magnification is variable, scaling also shows up. Translation alone can be handled by a single alignment pattern (fiducial). Rotation and/or scaling are preferably dealt with by means of two alignment patterns.



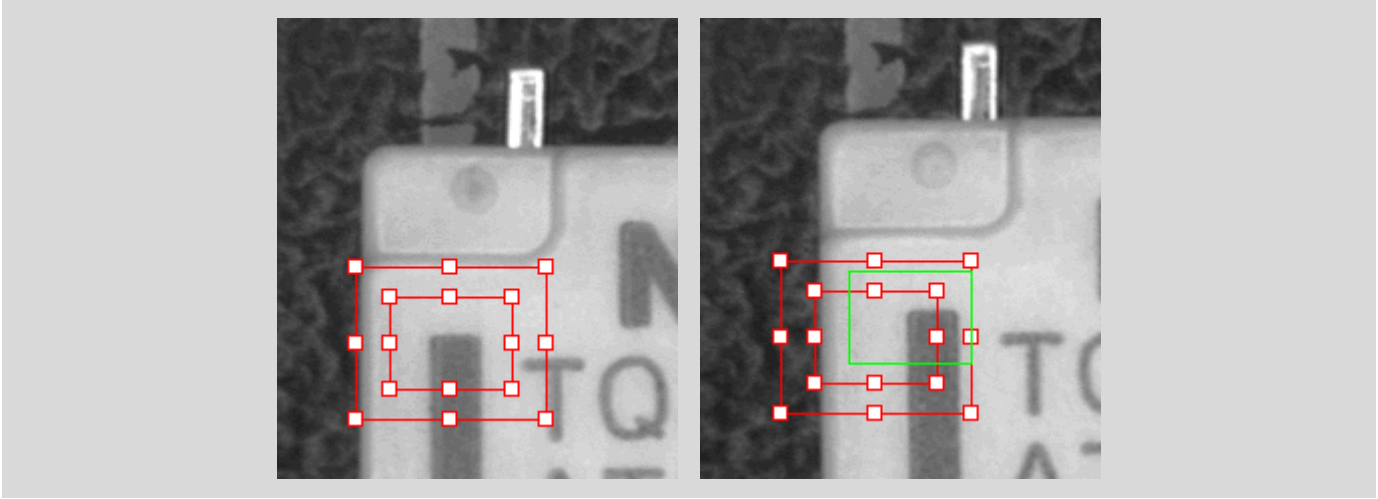
Bad: rotation not handled

- When choosing the alignment pattern(s), usual care must be taken (refer to EasyMatch). The patterns must be fixed, well-contrasted features not subject to degradation. They have to move rigidly with the inspected part. When two of them are used, they should be located as far apart as possible (in opposite corners), to achieve optimal accuracy. The pattern ROIs should not contain extraneous features likely to change from sample to sample. The patterns should be small so that rotation and scaling has little impact on them, but large enough to contain information at different scales.



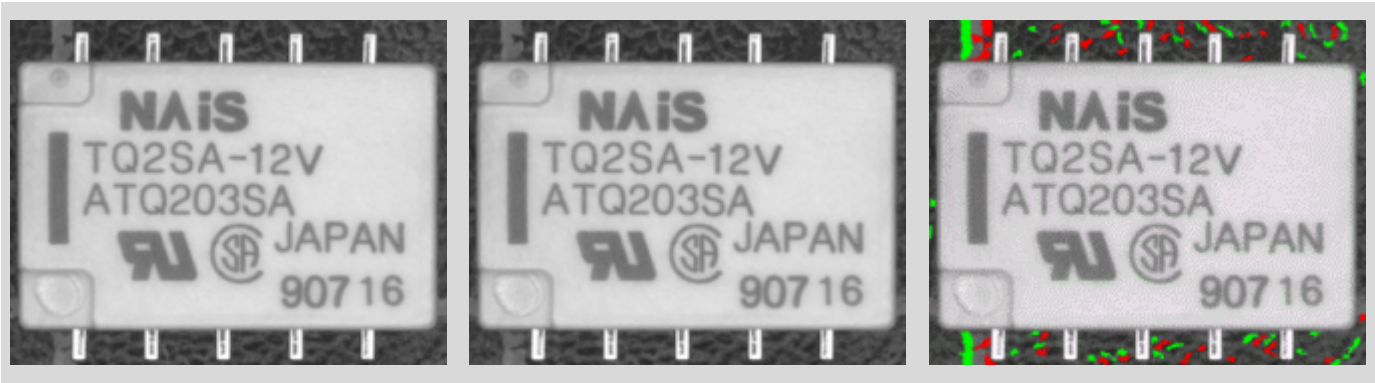
Bad: the location pattern is not repeatable

- After the patterns have been chosen, the search area around them must be defined. This area should be as tight as possible to reduce the search time and avoid possible false matches. Anyway, in normal operation, the pattern should never be found close to the edge of a search area; if this occurs, no guarantee can be given that the true match lies within the search area. The search area must provide a sufficient security margin. It is advisable to check on a representative set of good images that location by pattern matching never fails by touching on the search area edges.



Bad: too tight search areas

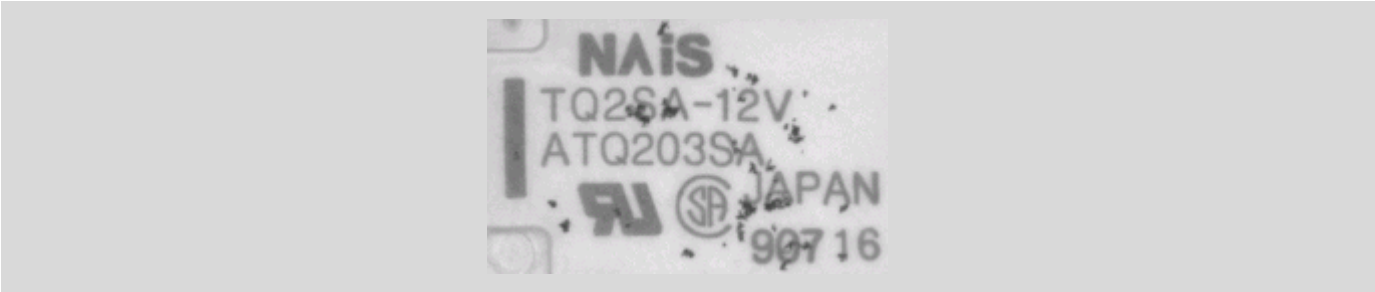
- The inspected ROI must then be defined in such a way that it surrounds all areas where defects are to be detected. Never forget, anyway, that only area belonging to the part should be inspected; otherwise, inspecting the background may raise false alarms because the movements of the part and of the background are independent.



Bad: undue inspection of the background

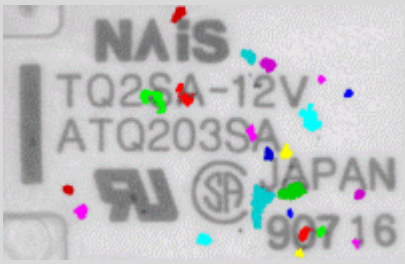
4.9 Inspecting with EChecker

Inspection is a straightforward task: after model set-up (as described in the learning phase above, or merely by loading a model file), the sample image is processed for realignment and gray-level normalization. The resulting image is passed to an ECodedImage object for blob analysis, and compared to the lower and upper reference images.



Inspected image, ...

... high and low threshold images, ...



... and detected blobs.

The blob analysis mechanisms allow grouping neighboring pixels to form blobs, discard the smaller of them (these are usually benign and correspond to noise), measure geometric characteristics on them (location, size, orientation, ...) among others. A whole chapter is devoted to this technique (see EasyObject (legacy)).

4.10 Tuning the Inspection Parameters

In principle, the learning operations are (relatively) time-consuming and cannot be used during on-line operation. Also, graphical interaction and a systematic sequence of operations are involved.

EChecker allows you to tune the global RelativeTolerance property at inspection time. This results in new contents for the lower and upper threshold images used by EasyObject (legacy). This is the only EChecker parameter that can be changed after learning.

5. Quality Indicators

Template/Sample, Foreground/Background

To understand this section, always keep in mind that the parameters computed once for all on the **template** image serve as a reference and are compared to those computed on the current **sample** image.

After accurate location of the model, the inspection process compares the contents of the sample image with the contents of the template, and rates the resemblance at the character level.

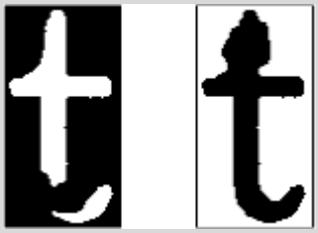
When the template image is binarized, the marking appears as white pixels on a black background. The foreground region of a character is formed by all the white pixels. The background region is formed by all remaining pixels in the character's bounding box. The bounding box is the tightest rectangle that wholly contains an item, with a possible safety margin.

Template image, foreground and background (white pixels)

Sample image, foreground and background (white pixels)

Area-Based Quality Indicators

The foreground and background template areas are the pixel counts of the foreground and background regions of a character. When the sample image is rated, thresholding also separates white and black pixels. The foreground and background sample areas are defined as the count of the white [black] sample pixels falling in the foreground [background] region. This is not the same as the total count of white and black pixels in the sample. As a fact, the difference between the template and sample areas measures the area of the defects detected in the foreground [background] of the character.

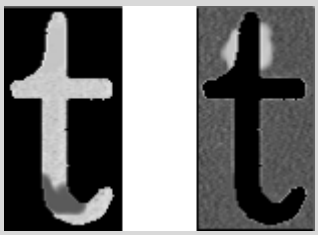


Foreground and background areas (white pixels)

Note. The area-based indicators rely on thresholding of the sample image. If necessary, the threshold level must be compensated for a change in intensity (automatic thresholding).

Gray Sum-Based Quality Indicators

A different measure of the amount of light reflected by the marking is given by sums rather than counts: the foreground [background] sample sum is defined as the sum of the gray-level values of all pixels of the foreground [background] region of the sample image. The foreground [background] template sum is the same feature computed on the template image to provide a reference value. Optionally, the sums are normalized with respect to the reference foreground and background average gray-levels to compensate for possible changes in gain (contrast) and offset (intensity).



Foreground and background sums

Note. The sum-based indicators do not rely on thresholding of the sample image, but the reference foreground and background gray-levels may take into account changes in gain and offset. With these indicators, acceptance/rejection of the characters is decided by comparing the values related to the sample and the template: if the difference is larger than the specified tolerance, a defect is reported. Too small, a foreground value indicates under-printing or missing character parts. Too large, a background value indicates over-printing or spurious character parts. Character mismatches provoke both kinds of anomalies.

Correlation-Based Quality Indicators

Normalized correlation, as used in pattern matching techniques (EasyMatch), is a way to rate the mismatches between two images. The correlation parameter is a global score in range **0** to **1**, which is implicitly corrected for a change in gain and offset.

Note. The correlation-based indicator is a global measure of mismatch that should be as close as possible to **1**. It is not sensitive at all to changes in gain or offset.

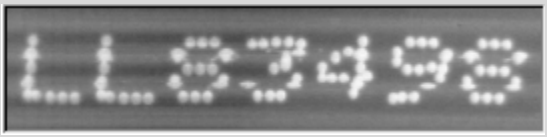
Reporting

After computation of the quality indicators, the detected defects are reported in three ways:

- explanatory diagnostics are associated to each inspected character and text;
- the items for which diagnostics are reported are automatically selected so that they can be immediately highlighted on the display;
- the relevant items are drawn as a box crossed by its main diagonal.

6. Template Model Definition

A very careful preparation of the template is the key to successful OCV. The Open eVision package includes a ready-made model editor (integrated in Open eVision Studio) that provides all means to interactively build and modify the structure of the template and then archive it. Writing your own template editor is also possible, though this is not a trivial task.



Raw image

From the raw image to the final model, the template definition follows a logical sequence of steps.

- First, the image is thresholded to separate the foreground (marking) from the background (printing medium). (If necessary, the thresholded image can be improved manually to join split characters or separate touching ones.)

Thresholded image

- Blob analysis (as done by EasyObject (legacy)) is performed and the detected objects may be selected depending on their size to generate a set of candidates. (If necessary, some blobs can be manually selected or unselected, to add features of unusual size or delete spurious objects.) At that stage, the blobs are said to be *free*.

Free objects

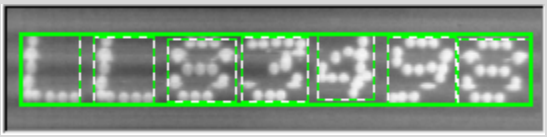
- The blobs are then aggregated automatically and enclosed into boxes corresponding to the characters. (If necessary, the grouping can be improved manually to merge several characters together or on the opposite separate blobs that were erroneously put together.) At that stage, the characters are said to be *free*.

Free characters

- The characters are then aggregated automatically and enclosed into boxes corresponding to the texts. (If necessary, this grouping can be improved manually to merge several texts together or on the opposite separate characters that were erroneously put together.)

Single text

- When the resulting structure is satisfactory, the actual learning process takes place. The system will analyze the shape of the template and generate the required data structures that represent it and allow fast inspection. It will also perform some quality measurement on the template for later comparison with the sample. At this stage, only texts and their constituent characters are stored. The free characters and free objects are no more relevant.



Final model

- Lastly, the inspection parameters must be defined. These include:
  - the allowed ranges for the location parameters of the texts with respect to their nominal position in the inspected ROI. These parameters correspond to translation and rotation (scaling in both horizontal and vertical directions, and shearing).
  - the allowed ranges for the location parameters of the characters with respect to their nominal position in the inspected texts. These parameters correspond to translation only.
  - the allowed ranges for the quality ratings with respect to their nominal values. These parameters can be chosen among the area of the character background and foreground, the accumulated gray level of these areas, and a similarity coefficient.

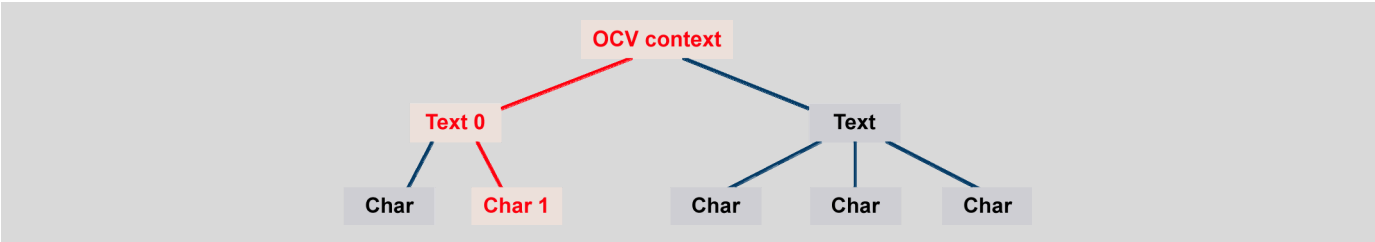
When the allowed ranges are specified, per text and per character, the template can be saved.

Note. Before commencing the interactive definition of the template, one should consider the inspection problem in question and decide:
- whether the marking should be considered as a single piece of text or as several independent ones, depending on the possible relative movements;
- to what extent the text position can vary in terms of translation, rotation [and possibly scaling and/or shearing];
- how the texts should be decomposed into several characters, if at all, depending on the desired level of detail for quality measurement;
- whether the individual characters are allowed to move with respect to their containing text.

7. Accessing the Model Components

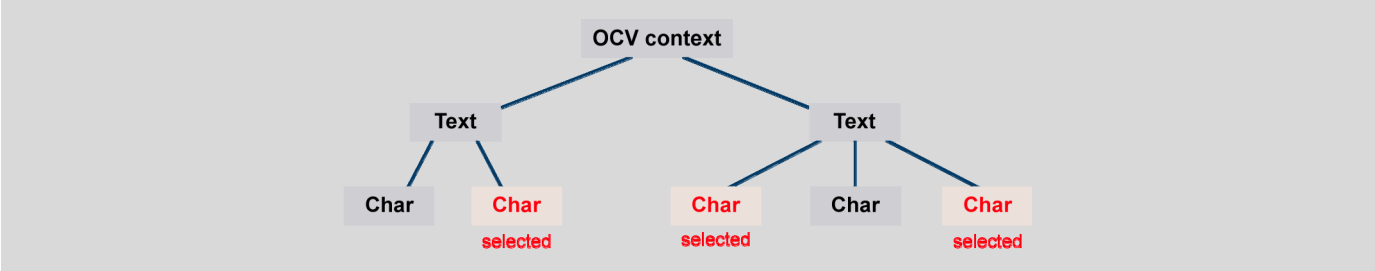
The OCV context contains a list of texts. Each text itself contains a list of characters.

To access a specific text, you provide its **index**, traversing the hierarchy from the OCV context. To address a particular character, you provide both the index of the text that contains it, and the index of the character within this text.



Access by index (second character of first text)

You can also access and modify several components collectively. Every text and every character has a boolean **selected** property, so that in a single operation, you can read/write the value of some property for all currently selected elements. All the selected components will be accessed at one time. When reading a property, a value is only returned if it is identical for all selected elements.



Access a group of selected components

Scattering operations on selected components is very handy for implementing interactive editors that allow to list and modify parameters for single items, or groups of items.

8. Statistics

The template image used during learning serves to compute the reference value of the quality indicators. Using a single image has drawbacks:

- the chosen image may itself have small, unnoticed defects and not be fully representative of the whole population.
- a single image gives no insight on the random variations between acceptable samples, and give no way to adjust the tolerances on the quality indicators.
- the default quality tolerances may be adapted to the sharpness of the default you would like to detect.

EasyOCV allows to accumulate results on a series of consecutive inspections: the average value of the quality indicators can be reported, as well as their standard deviation. These parameters allow you to choose appropriate values for the acceptance ranges. The library also carries necessary functions to automatically adjust position as well as quality tolerance parameters from statistics. It is customary to select a tolerance value that is a small multiple of the observed standard deviation (±2 sigma or ±3 sigma criterion).

In this way, better control of the manufacturing process can also be gained. The average shows the long-term trend behavior of the system and allows tp detect drift in the marking device working. The standard deviation is related the repeatability of the process and allows to detect the appearance of slack.

9. Programming with EasyOCV

9.1 Introduction

Writing an inspection application for the production line can be made very simple. If no operator intervention is required to adjust parameters, and if no *in situ* learning phase is necessary, few operations are used:

The model can be loaded once for all.

- For every acquired image inspection takes place.
- The results of inspection are graphically displayed on top of the image.
- A diagnostics report can be generated, including relevant quality indicators, and possibly statistics on them.

A slightly more advanced inspection application may allow the operator to modify the inspection parameters. This means that dialog boxes must be provided to allow editing the parameters. In the simplest form, the parameters are changed globally (same value everywhere). Otherwise, the operator must have the ability to select the texts and characters on which to work.

In a full fledged application, model edition before learning must be made possible. This requires the capability to select and unselect items, display parameters and modify them, load and save to a model file. This leads to more complex programming.

This section presents required steps, from simplest to more advanced.

9.2 Setting the ROI for Inspection

The ROI should be placed so that it is roughly centered on the marking. The ROI should be positioned in a repeatable way with respect to the marking substrate. This is achieved either when the position of the inspected object is known and stable, or when the object has been located by appropriate means such as pattern matching or edge measurement.

The ROI is not used to evaluate the global contrast of the sample marking. Instead, it is used to:

- Define the effective search area for each text thanks to their centers and their location parameters (**ShiftX**, **ShiftY**, ...). This way, inspection is by no means confined to the inspected ROI.
- Locate the marking. After the marking is located, its global contrast is evaluated in an ROI centered on the marking, and whose dimensions are those of the learning time.

9.3 Inspecting

A threshold level is required to determine the global marking contrast. Automatic thresholding can be used.

9.4 Drawing the Inspected Items

The inspected texts and characters can be represented by their bounding box, at the position determined by the location process. Drawing can be done selectively, allowing the selected and unselected items appear in a different color.

Items with detected defects will appear crossed by their main diagonal.

9.5 Retrieving the Diagnostics and Quality Indicators

After inspection, global inspection diagnostics are available. They summarize all defects found on the marking and allow raising alarms.

If a more detailed reporting is necessary, diagnostics are also available for each text and character. Furthermore, all measured quality indicators can be retrieved.

Retrieving the text diagnostics and parameters involves a loop where all texts are visited. Retrieving the character diagnostics and parameters involves a double loop where all texts and all characters of all texts are visited.

9.6 Setting Inspection Parameters

During operation, it may be necessary at times to adjust the value of some working parameter. This may be done in different ways:

- **Global change**: the same value of a given parameter may be set for all texts and/or all characters. This is straightforward and requires a single call to **ScatterTextsParameters**, **ScatterTextsCharsParameters**.
- **Selective change**: the values can be set for those texts or characters that are in the selected state. This requires mastering the interactive selection of text and characters. The description of the selection operations appears below.
- **Custom change**: the values can be adjusted individually using a user-defined rule. This approach is similar to the retrieval of parameters using indexed access.

Note. Before using one of the **Scatter** parameters function, make sure that the parameters you don't want to change are set to an undefined value.

9.7 Selecting Items Interactively

When the operator wants to retrieve or modify parameters, either individually or grouped, he must have the ability to select them using the mouse cursor. EasyOCV provides support for such operations by means of a general selection/de-selection mechanism: several selection functions allow toggling (selected become unselected and conversely) the selection state of all items (partially) contained in a given rectangle. This toggling can be applied to all items or only on the currently selected or unselected ones.

Note. The rectangle is usually obtained as the result of a dragging operation. A degenerate rectangle (reduced to a single point) can be used as well to handle point clicking.

Since the toggling mechanism combined with the possible rectangle extent and current selection mode is tricky, let us give a few examples.

Assume a model of three texts in the following states: **Selected**, **Selected**, **Unselected**.

- Using a rectangle that contains all three of them will set them to states **Unselected**, **Unselected**, **Selected** (SSU -> UUS).
- Using a rectangle that touches the first of them will set the states to **Unselected**, **Selected**, **Unselected** (SSU -> USU).

Now consider the same operations applied to the selected texts only.

- Using a rectangle that contains all three texts will set them to states **Unselected**, **Unselected**, **Unselected** (SSU -> UUU).
- Using a rectangle that touches the first of them will set the states to **Unselected**, **Selected**, **Unselected** (SSU -> USU).

Now consider the same operations applied to the unselected texts only.

- Using a rectangle that contains all three texts will set them to states **Selected**, **Selected**, **Selected** (SSU -> SSS).
- Using a rectangle touches the first of them will leave their states unchanged (SSU -> SSU).

Note. This selection mechanism applies to texts and characters at inspection time (**SelectSample...**). It also applies to free objects, free characters and texts during the model edition phases (**SelectTemplate...**).

9.8 Computing Inspection Statistics

Using EasyOCV, gathering statistical information on the process is possible. For each measured parameter (location parameters and quality indicators), the average and standard deviation can be estimated from a number of samples.

The procedure is straightforward: after an image has been inspected, one can request that the measured parameters be taken into account as valid samples by calling UpdateStatistics. (If, for any reason, the sample is to be rejected, just do not call UpdateStatistics.) After at least two sample images have been processed, the average and standard deviations can be obtained. The standard mechanisms for text and character parameters retrieval can be used.

To compute the statistics afresh on new samples, start by calling ClearStatistics. The number of samples accumulated so far is given by StatisticsCount.

9.9 Adjusting Inspection Parameters from Statistics

Statistics may also be used to adjust location and quality indicators. When adjustments should be made on individual texts or characters, the selection mechanism described above is applicable.

When performing quality range adjustment, the bias value of each used indicator is assigned the average value of the inspected samples (provided they had been added to statistics), whereas the indicator tolerance is assigned s times the standard deviation, where s is a security factor to provide. If you wish not to adjust quality range of a particular indicator, you should first de-activate it by setting UsedQualityIndicators before proceeding.

When performing location parameters adjustment, the minimum and maximum values are used. A security factor may also be specified.

To perform location parameters adjustment, call AdjustTextsLocationRanges, AdjustCharsLocationRanges. To perform quality ranges adjustment on used indicators, call AdjustTextsQualityRanges, AdjustCharsQualityRanges.

9.10 Interactively Editing a Model

Writing an OCV model editor requires a good understanding of windowed applications design. In particular, it is important to know how to manage the mouse cursor movements, when and how to refresh the display, handle dragging of selection rectangles and the like. It is out of the scope of this documentation to explain these features which are deeply related to Windows programming. Also note that the level of functionality, from blind -no operator intervention- to full fledged editing is a matter of taste and of programming skill.

Recall the steps in defining the model structure:

1. An **ECodedImage** object is used to segment the image into blobs (**BuildObjects** method).
2. Possibly, blob selection by all means provided in EasyObject (legacy), is performed (**SelectObjectsUsingFeature** or **SelectObjectsUsingPosition**). In particular, small blobs generated by noise should be unselected.
3. The selected blobs are passed to the OCV object and enter the free objects list (EasyObject (legacy)'s blobs become OCV's free objects, or **TemplateObjects**).
4. At this point, the objects in the free list can be selected/unselected interactively.
5. The (selected) free objects are then used to generate free characters, using one of the available grouping policy (free objects become free characters, or **TemplateChars**).
6. At this point, the free characters can be selected/unselected interactively.
7. The (selected) free characters are then used to generate texts. The default policy is to group all free characters in a single piece of text (free characters become texts, or **TemplateTexts**; these texts now contain embedded characters, or **TemplateTextChars**).

In the simplest form of a model editor, steps 4 and 6 can be skipped, meaning that all free objects and all free characters will enter the model. A better editor will allow withdrawal of unwanted items and explicit grouping (steps 4 and 6). An even more powerful editor should allow grouping as well as ungrouping (backwards from 3 to 2, from 5 to 4, from 7 to 6).

At any time, the following operations can be handled:

- The model components can be selected/unselected interactively (using **SelectTemplateObjects**, **SelectTemplateChars**, **SelectTemplateTexts**).
- New items can be grouped to form new higher level items (using **CreateTemplateObjects**, **CreateTemplateChars**, **CreateTemplateTexts**).
- Items can be ungrouped by destroying the higher level item (using **DeleteTemplateTexts**, **DeleteTemplateChars**, **DeleteTemplateObjects**).

A clean way to organize the editor is to define a sequence of separate phases dealing with objects, free objects, free characters and texts.

9.11 Advanced Features

Contrast Parameters

Image contrast is an important factor during both learning and inspection.

The background and foreground information must be separated using an appropriate threshold, possibly determined automatically. After a threshold is given, the average gray level of the background and foreground areas are computed separately. These serve as reference gray levels to measure the image contrast and normalize the gray level quality indicators if needed.

The background and foreground reference gray levels are computed for both the template and sample images. See the EOCVChar properties. The sample global contrast may then be compared to the template contrast reference value, in order to diagnose an over-contrasted or under-contrasted image before diagnosing further.

Note. The template threshold directly influences the thickness of the blobs in the model. It also has an effect on the reference gray levels of the template image.
The sample threshold influences the reference gray levels of the sample image. When gray level normalization is used, this influence is perceived on the location score and gray-level sums. It also has an effect on the thickness of the blobs in the sample image, with immediate consequences on the sample areas. No other quality indicators depends on it.

Location Modes

For location of the model components, a search process is used during which EasyOCV tries to find the character edges in the sample image, possibly transformed. Four location modes are provided: raw, binarized, gradient and Laplacian. See enumeration constants ELocationMode.

Experience reveals that the best location reliability is achieved by the binarized and gradient modes. Additionally, the gradient mode is not sensitive to the choice of a threshold level. Use of the Laplacian mode is not recommended.

In case you experience location problems, the first action to take is to try another location mode. No definite guidelines can be given.

Location Score

It is highly recommended to keep the default reduction of location scores option turned on. Reduction consists in dividing a raw location score by the number of points its computation required. Thus, the calculated scores do not depend on the number of used points any longer; the number of used points may be decreased without degrading localization to save potential time, if necessary.

Location scores may also be normalized. This option is useful in case the lightening conditions of sample images are not the same from a sample to another, or when template and sample images have obviously different reference gray levels. The action of this option is equivalent to performing a global contrast correction on the sample image (or ROI). By the way, location scores do not depend on the reference gray levels, and are so more reliable.

Finally, AccurateTextsLocationScores provides an alternative way to compute text location score. During location process, EasyOCV tries first to locate texts using their contour, which are sets of fixed points one from each other. This rigid definition of text contour has the draw back to make the library return poor location score values if the sample characters have moved from their nominal positions, and may result in a false alarm.

If the AccurateTextLocationScores property is turned on, a text location score will be computed as sum or average of the characters location scores that form the text (depending on the state of the ReduceLocationScore property). This way, texts location scores become independent from the position of the characters they contain.

Used Quality Indicators

For a given inspection case, not all quality indicators are relevant. For instance, it sometimes suffices to use the location scores alone to detect absence of a given marking.

To avoid false alarms raised by unused quality indicators for which the tolerances have not been adjusted, and to avoid unnecessary processing, it is important to activate only the quality indicators in use through the UsedQualityIndicators property.

Character Resampling

Normally, when text is inspected with rotation, scaling and/or shearing, some resampling must be performed to compute the quality indicators on the separate characters. If the angles remain small and the scale factors remain very close to unity, this resampling can be avoided by setting the ResampleChars parameter to FALSE.

EasyBarCode

1. What Is EasyBarCode?

EasyBarCode is a library dedicated to automatically locate and read bar codes. It is also able to identify and read a wide range of standard commonly used symbologies, as well as additional symbologies.

EasyBarCode automatically locates the bar code symbol in the image. Moreover, for prototyping or special cases, an advanced manual location mode is also available.

Specifications

- EasyBarCode supports numerous symbologies.
- The bar code may be printed with direct contrast (black ink on white background) or inverse contrast (white ink on black background).
- The bar and space sequence should be preceded and followed by a quiet zone of, at least, ten times the smallest bar or space thickness.
- Bars should also be surrounded below and above by a quiet zone of a few pixels.
- Bars and spaces width must be greater or equal to 2 pixels.

1.1 What Is a Bar Code?

A bar code is a 2D pattern made of a sequence of parallel bars and spaces of varying thickness. Bars and spaces are arranged according to the rules of a particular encoding convention, called the **symbology**, that specifies the allowed characters set and the encoding rules. The whole sequence represents a short character string.

Bar code (EAN 13 symbology)

Bar codes are widely used for marking and identifying goods.

Bar code reading is an essential task in many industrial vision applications. Printed bar codes are generally used to ensure traceability of manufactured goods; they can carry serial/lot numbers, manufacturer/product identification. By contrast with printed characters, bar codes are machine-readable and have been specifically designed for this purpose. In most cases, bar codes are printed in black ink on a white paper label, most of the time with an excellent contrast.

1.2 Supported Symbologies

Here follow the EasyBarCode supported symbologies, in two distinct groups.

Standard Symbologies

The standard symbologies are considered as major types. They are enabled by default.

- Code 128
- Code 39
- EAN 128
- Codabar
- Code 2/5 5 Interleaved
- EAN 13*
- MSI
- UPC A*
- UPC E

* EAN 13 and UPC A only differ by the layout of surrounding digits.

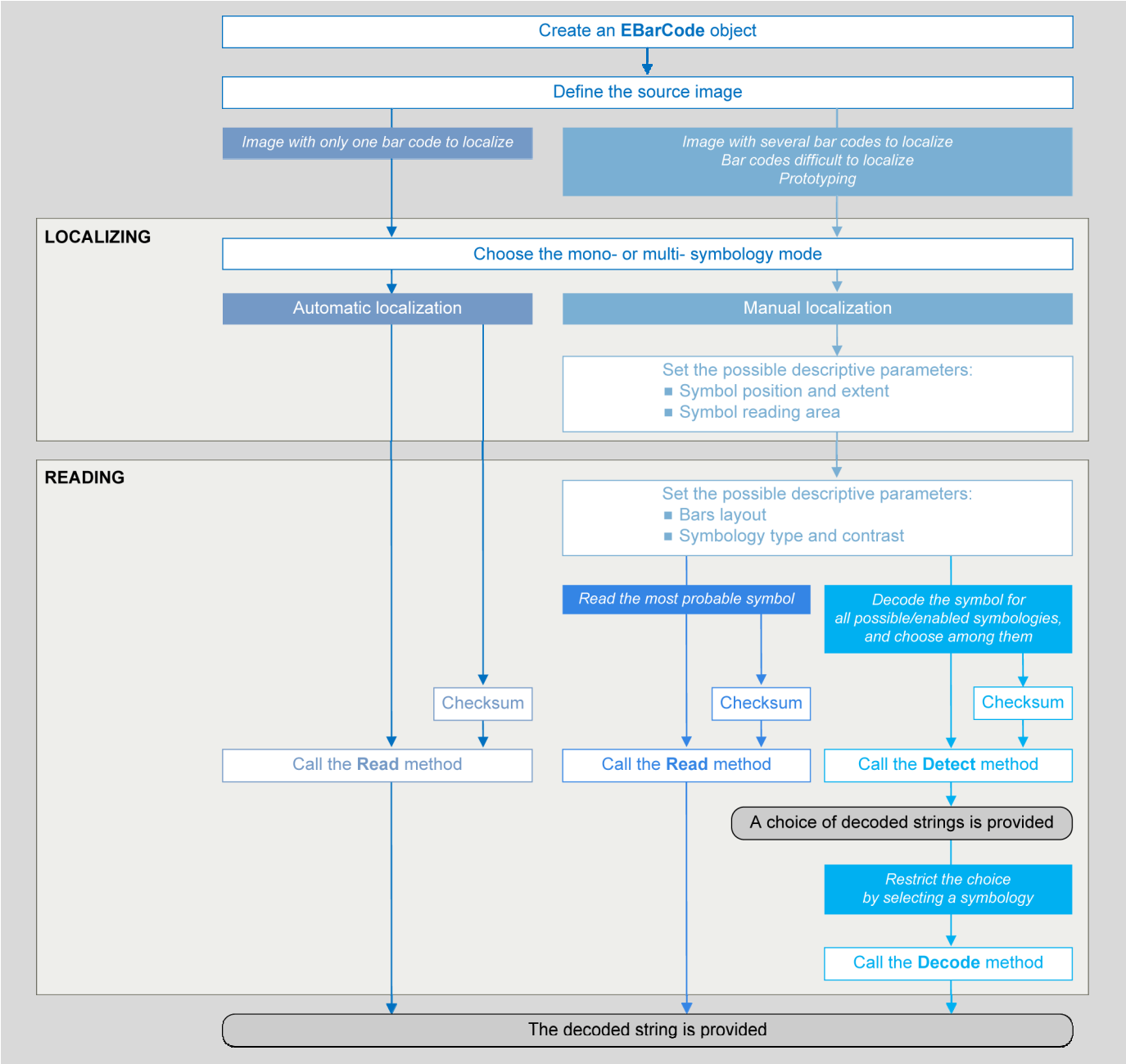
Additional Symbologies

- Code 32
- Code 39 Extended*
- Code 39 Reduced*
- Code 412 SEMI
- Code 93
- Code 93 Extended
- Code STK
- IBM Delta Distance A
- Plessey
- Telepen
- ADS Anker
- Binary code
- Code 11
- Code 13
- Code 2/5 3 Bars Datalogic
- Code 2/5 3 Bars Matrix
- Code 2/5 5 Bars IATA
- Code 2/5 5 Bars Industry
- Code 2/5 5 Compressed
- Code 2/5 5 Inverted
- Code BCD Matrix
- Code C.I.P.
- EAN 8

* Code 39 Extended is a super-set of Code 39. Code 39 Reduced is a subset of Code 39.

2. EasyBarCode Workflow

Note that the workflow ONLY shows the generic steps required to work in EasyBarCode.



3. Mono- or Multi-Symbology Modes

For a known symbology, the sequence of bars and spaces that encode a given string is unique and well defined.

Conversely, several different character strings could be encoded with the same sequence of bars and spaces but in different symbologies.

This remark points out the problem of symbology ambiguity in bar codes reading.

So, there are basically two modes of operation for bar codes reading when symbology choice is concerned: mono-symbology and multi-symbology.

In mono-symbology mode, the user knows perfectly the kind of expected bar code symbology or a limited number of them is allowed (without ambiguity). The reading process provides the string interpretation based only on this symbology. If the localized bar code is not readable within the framework of this symbology, an error occurs. In this mode, they are no ambiguity; the bar code is readable or not, and if readable, there is one and only one interpretation for the character string.

In multi-symbology mode, the user doesn't know exactly the expected symbology or a lot of them is allowed. The reading process provides the interpretation (character string or error condition) of the bars and spaces sequence for each enabled symbology. One expects that valid interpretations will be found for some symbologies and error conditions will be raised for the others.

The reading process orders enabled symbologies in the following way:

- Decoded symbologies (with a valid character string interpretation) come first.
- Amongst decoded symbologies, standard ones come first.
- Symbologies with error conditions come after all decoded symbologies, and are listed by increasing order of error severity (the least severe comes first).

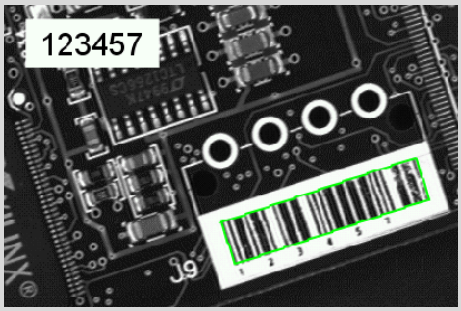
Remarks

- The reader is able to help a non-aware user to determine the symbology of a given symbol.
- Code 39 extended is a super-set of code 39 and code 39 reduced is a subset of code 39.
- Code EAN13 and UPC A only differ by the layout of surrounding digits.

4. Localizing Bar Code Automatically

EasyBarCode supports two approaches for the symbol location; automatic (which is the preferred mode) and manual (which is an advanced mode used for prototyping or when the automatic localization fails).

In the automatic mode, the reading algorithm locates the bar code in the field of view on its own. If several bar codes are present in the field of view, one of them is located. Once the symbol has been found, it is decoded. In this mode, EasyBarCode can be seen as a straightforward hand-held bar code reader.



Decoded bar code

The reader then reports the encoded information (if the bar code was readable) or the reason for failure (if it was not readable).

If the automatic localization fails or for prototyping purposes, the manual location mode can be more suitable. It requires the user to provide the bar code position.

5. Localizing Bar Code Manually

If the automatic localization fails or for prototyping purposes, the manual location mode requires the user to provide the **bar code position**.

Bar code position and extent

This can be done graphically by adjusting a rectangle around the symbol or by explicitly specifying its descriptive parameters. In this mode, if several symbols appear in the image, they can be processed one after the other.

In this mode, only part of the bar code will be used to compute the character string. This is the reading area.

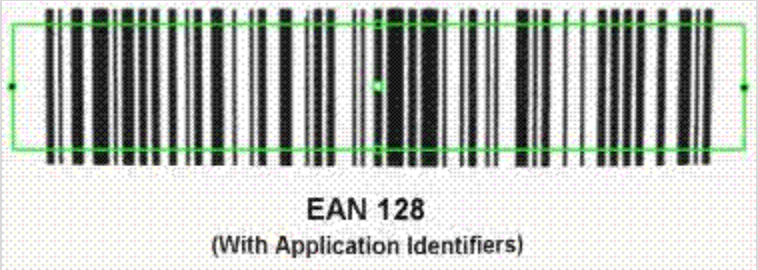
Bounding box — graphical appearance (manual location)

Reading area

The reading area is a rectangular portion of an image that is directly used to decode a bar code. This reading area is specified (graphically or not) respectively to the bar code bounding box.

The reading area has the following features:

- Its horizontal extension is wider than the bar code bounding box width to be sure not to miss the external bars.
- Its height is less than the bar code bounding box height to avoid problems coming from character strings under the bar code, bar occlusion, strong deformations or bar slant.
- It may also be rotated relatively to the bar code bounding box to take into account the slant of bars relatively to the bar code axis normal direction. (Advanced mode!)



Reading area — graphical appearance (manual location)

6. Reading Bar Code

The reader is able to help a non-aware user to determine the symbology of a given symbol.

The reader reports the encoded information (if the bar code was readable) or the reason for failure (if it was not readable).

The running time performance and the robustness are enhanced by using prior information on the symbology. The following descriptive parameters can be determined:

- **Bars layout**
- **Symbology contrast**
- **Symbology type**: Before performing any reading operation, the set of symbologies used for the decoding have to be specified. To enable symbologies, set the **StandardSymbologies**, or **AdditionalSymbologies** properties, depending on the group the symbologies belong to.

Read the most probable symbol

If only the most likely meaning matters or in case of the mono-symbology reading mode, use Read.

Decode the symbol for all enabled symbologies and choose among them

When ambiguities do arise (a given symbol might have a meaning under different symbologies), the reading of a bar code is triggered by means of Detect. It reports the number of possible symbologies by the means of the NumEnabledSymbologies property, and lists the data contents by decreasing likeliness. To retrieve the decoded data pertaining to a specific symbology, call the Decode method in a loop, using GetDecodedSymbology to walk through the list of successful symbologies in decreasing order of likelihood.

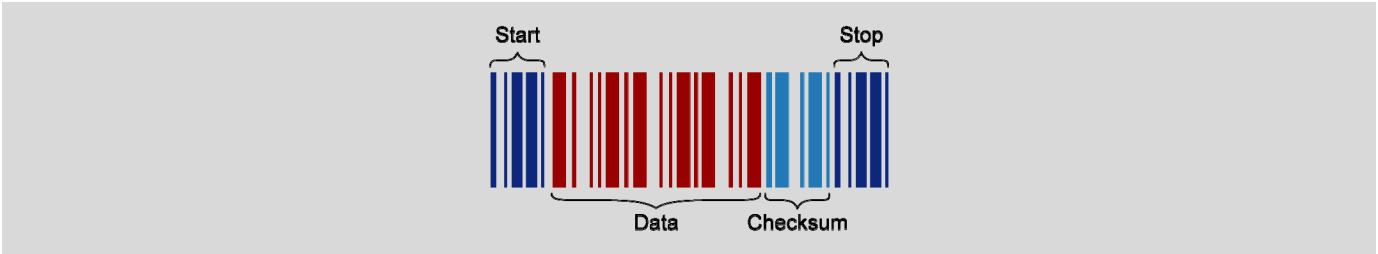
7. Checksum

The checksum is an additional character that is a function of all others encoded characters. It enables the reader to check the decoded character string coherence.

Symbologies take into account (or not) the checksum and its verification in several ways:

- The checksum is mandatory for a given symbology and it must be checked by the reader.
- The checksum is mandatory for a given symbology but its verification by a reader is optional.
- The checksum and its verification are optional for a given symbology.

EasyBarCode allows the user to verify or not the checksum for all enabled symbologies. By default, checksum is not controlled. To enable or disable checksum verification for all enabled symbologies, set the VerifyChecksum property.



Bar code structure (Code 39)

8. Dealing with Strings that Contain NULL Characters

When the bar code contains \0x00 characters, the **std::string::c_str** method should not be used (since C-strings are terminated by the \0x00 character). An iterator over the characters should be used instead of a C-string.

EasyMatrixCode

1. What Is EasyMatrixCode?

EasyMatrixCode is a fully automatic reader of 2D Data Matrix codes. It recognizes and decodes a symbol of any size, contrast, location and orientation in a single reading operation. EasyMatrixCode supports gray level images.

As an automatic library, a learning phase is not required, but is an option. In addition, characteristics of the Data Matrix code may be set manually, allowing to restrict the search, and thus to accelerate the process. The characteristics are for example: family of the Data Matrix code, number of cells, black cells on white background or reverse, flipping allowed or not.

Print quality control indicators based AIM standard values are available as well. The matrix code reader can be configured to compute, when reading, the matrix code grading. Enabling the grading computation slightly increases the reading time (about 5 %). By default, this feature is disabled.

Error detection and correction algorithms are used to provide a faithful reading.

Specifications

- Minimum quiet zone (blank zone around the matrix code) width: 3 pixels.
- Minimum cell (= module) size: 3x3 pixels.
- Maximum stretching (ratio between cell width and cell height): 2.

1.1 What Are Data Matrix Codes?

A Data Matrix code is a two-dimensional symbol made up of a set of dots (or cells) arranged in a specific way.

Data Matrix code (ECC 200, 26x26 cells)

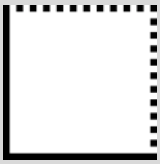
The data contents of the Data Matrix code is a string of characters, chosen from several possible character sets (digits, letters and possibly special characters), encoded using various encoding schemes to achieve maximum packing. Presence or absence of a dot at a given position corresponds to a single bit of information. In addition to the data bits, redundant bits are added to allow error correction for robust reading of degraded symbols.

Data Matrix codes are widely used for parcel tracking and part identification in the semiconductor, pharmaceutical and mechanical industries. They can be considered as a generalization of bar codes, enabling storage of a much larger volume of information.

Note. The definition of Data Matrix codes is provided by AIM International Inc. (PA). It is available as document *AIM USA Uniform Symbology Specifications: Data Matrix (Item X5-11)*, and has been approved as standard ANSI/AIM BC11-1997.

A Data Matrix code is characterized by its logical size (number of cells) and its encoding type, corresponding to several levels of error correction capability, namely ECC 000, ECC 050, ECC 080, ECC 100, ECC 140 (odd symbol sizes) and ECC 200 (even symbol sizes). See also Matrix Codes Sizes and Data Capacity.

A Data Matrix code can be easily recognized as a square or rectangular array of elementary cells. There are two kinds (classes) of cells: the "black" ones and the "white" ones. The bottom and left edges contain only black cells, while the top and right edges have alternating cells. This arrangement is known as the **finder pattern**.



Data Matrix code finder pattern

In some circumstances, the symbol can be inverted by a mirroring effect. Specifically, if image acquisition is made in such a way that the image is upside down (bottom-up), or if the symbol is looked at from the back on a transparent medium, it is said to be **flipped**. EasyMatrixCode is able to handle flipped matrix codes.

When a symbol is readable, the potential abnormal cells can be located in the array, and the quality of the printing can be assessed by a set of standardized parameters.

Matrix Codes Sizes and Data Capacity

ECC 000

Symbol Size Rows	Data Capacity		
	Numeric	Alphanumeric	Byte
9	3	2	1
11	12	8	5
13	24	16	10
15	37	25	16
17	53	35	23
19	72	48	31
21	92	61	40
23	115	76	50
25	140	93	61
27	168	112	73
29	197	131	86
31	229	153	100

33	264	176	115
35	300	200	131
37	339	226	148
39	380	253	166
41	424	282	185
43	469	313	205
45	500	345	226
47	560	378	248
49	596	413	271

ECC 050

Symbol Size Rows	Data Capacity		
	Numeric	Alphanumeric	Byte
11	1	1	Not possible
13	10	6	4
15	20	13	9
17	32	21	14
19	46	30	20
21	61	41	27
23	78	52	34
25	97	65	42
27	118	78	51
29	140	93	61
31	164	109	72
33	190	126	83
35	217	145	95
37	246	164	108
39	277	185	121
41	310	206	135
43	344	229	150
45	380	253	166
47	418	278	183
49	457	305	200

ECC 080

Symbol Size Rows	Data Capacity		
	Numeric	Alphanumeric	Byte
13	4	3	2
15	13	9	6
17	24	16	10
19	36	24	16
21	50	33	22
23	65	43	28
25	82	54	36
27	100	67	44
29	120	80	52
31	141	94	62
33	164	109	72
35	188	125	82
37	214	143	94
39	242	161	106
41	270	180	118
43	301	201	132
45	333	222	146
47	366	244	160
49	402	268	176

ECC 100

Symbol Size Rows	Data Capacity		
	Numeric	Alphanumeric	Byte
13	1	1	Not possible
15	8	5	3
17	16	11	7
19	25	17	11
21	36	24	15
23	47	31	20
25	60	40	26

27	73	49	32
29	88	59	38
31	104	69	45
33	121	81	53
35	140	93	61
37	159	106	69
39	180	120	78
41	201	134	88
43	224	149	98
45	248	165	108
47	273	182	119
49	300	200	131

ECC 140

Symbol Size Rows	Data Capacity		
	Numeric	Alphanumeric	Byte
17	2	1	1
19	6	4	3
21	12	8	5
23	17	11	7
25	24	16	10
27	30	20	13
29	38	25	16
31	46	30	20
33	54	36	24
35	64	42	28
37	73	49	32
39	84	56	36
41	94	63	41
43	106	70	46
45	118	78	51
47	130	87	57
49	144	96	63

ECC 200

Symbol Size Rows	Data Capacity		
	Numeric	Alphanumeric	Byte
10	6	3	1
12	10	6	3
14	16	10	6
16	24	16	10
18	36	25	16
20	44	31	20
22	60	43	28
24	72	52	34
26	88	64	42
32	124	91	60
36	172	127	84
40	228	169	112
44	288	214	142
48	348	259	172
52	408	304	202
64	560	418	278
72	736	550	366
80	912	682	454
88	1152	862	574
96	1392	1042	694
104	1632	1222	814
120	2100	1573	1048
132	2608	1954	1302
144	3116	2335	1556

Symbol Size		Data Capacity		
Rows	Columns	Numeric	Alphanumeric	Byte
8	**18**	10	6	3
8	**32**	20	13	8
12	**26**	32	22	14

12	**36**	44	31	20
16	**36**	64	46	30
16	**48**	98	72	47

1.2 Print Quality Control

When the print quality indicators are needed, the matrix code reader can be configured to compute, when reading, the matrix code grading. Enabling the grading computation slightly increases the reading time (about 5 %). By default, this feature is disabled.

After the read operation, the AIM standard values are available through the following matrix code properties:

- **Contrast**
- **PrintGrowth**
- **AxialNonUniformity**
- **UnusedErrorCorrection**

Each property has its corresponding grading (from 4 to 0, i.e. 'A', 'B', 'C', 'D' and 'F' respectively).

- **ContrastGrade**
- **PrintGrowthGrade**
- **AxialNonUniformityGrade**
- **UnusedErrorCorrectionGrade**

See also Normalized Print Quality Indicators.

Normalized Print Quality Indicators

This informational topic is taken from the AIM "International Symbology Specification - Data Matrix".

N.2.2 Symbol Contrast

Within the gray-scale image, all of the image pixels which fall within the area of the test symbol, extending outward to the limits of any required quiet zones, shall be sorted by their reflectance values to select the darkest 10 % of the pixels and the lightest 10 % of the pixels. Calculate the arithmetic mean of the reflectance of the darkest 10 % and the arithmetic mean of the reflectance of the lightest 10 %. The difference of the two means is the *Symbol Contrast* (**SC**).

The **SC** grade is determined by:

```
A (4.0) if SC > 70 %
B (3.0) if SC > 55 %
C (2.0) if SC > 40 %
D (1.0) if SC > 20 %
F (0.0) if SC < 20 %
```

SC tests that the two reflective states in the symbol, namely light and dark, are sufficiently and consistently distinct throughout the symbol.

N.2.3 "Print" Growth

Calculate a reflectance threshold halfway between the dark and light means from Annexe N.2.2. Create a secondary binary image distinguishing dark and light regions using the threshold.

The print growth parameter, the extent to which dark or light markings appropriately fill their module boundaries, is an important indication of process quality which affects reading performance. The particular graphical structures most indicative of element growth or shrinkage from nominal dimensions will vary widely between symbologies, and shall be defined within their specifications, but will generally be either fixed structures or isolated elements whose *dimension(s)* (**D**) is(are) determined by counting pixels in the binary digitized image. More than one dimension, for example both horizontal and vertical growth, may be specified and checked independently. Each checked dimension shall have specified both a nominal value **DNOM** and maximum **DMAX** and minimum **DMIN** allowed values. Each measured **D** shall be normalized to its corresponding nominal and limit values:

$$D' = (D - D_{NOM})/(D_{MAX} - D_{NOM}) \text{ if } D > D_{NOM}$$
$$= (D - D_{NOM})/(D_{NOM} - D_{MIN}) \text{ otherwise.}$$

Print growth is then graded according to:

```
  A (4.0) if -0.50 <= D' <= 0.50
  B (3.0) if -0.70 <= D' <= 0.70
  C (2.0) if -0.85 <= D' <= 0.85
  D (1.0) if -1.00 <= D' <= 1.00
F (0.0) if D' < -1.00 or D' > 1.00
```

Print Growth tests that the graphical features comprising the symbol have not grown or shrunk from nominal so much as to hinder readability with less optimum imaging conditions than the test condition.

N.2.4 Axial Non-Uniformity

2D matrix symbols include data fields of modules nominally lying in a regular polygonal grid, and any reference decode algorithm must adaptively map the enter positions of those modules to extract the data. *Axial Non-uniformity* (**AN**) measures and grades the spacing of the mapping centers, i.e. the sampling points, in the direction of each of the grid's major axes.

The spacings between adjacent sampling points are independently sorted for each polygonal axis, then the average spacing **XAVG** along each axis is computed. **AN** is a measure of how much the sampling point spacing differs from one axis to another, namely:

$$AN = abs(X_{AVG} - Y_{AVG})/[(X_{AVG} + Y_{AVG})/2]$$

where **abs()** yields the absolute value. If a symbology has more than two major axes, then **AN** is computed for those two average spacings which differ the most. **AN** is then graded as:

```
A (4.0) if AN < 0.06
B (3.0) if AN < 0.08
C (2.0) if AN < 0.10
D (1.0) if AN < 0.12
F (0.0) if AN > 0.12
```

AN tests for uneven scaling of the symbol which would hinder readability at some non-normal viewing angles more than others.

N.2.5 Unused Error Correction

The correction capacity of Reed-Solomon decoding is expressed in the equation:

```
e + 2t <= d - p
```

where:

- **e** is the number of erasures,
- **t** is the number of errors,
- **d** is the number of error correction codewords,
- **p** is the number of codewords reserved for error detection.

Values for **d** and **p** are defined by symbology specification (often depending on symbol size), while **e** and **t** are determined during the successful reference decode. The amount of *Unused Error Correction* (**UEC**) is computed as

```
UEC = 1.0 - (e + 2t)/(d - p)
```

In symbols with more than one (e.g., interleaved) Reed-Solomon block, **UEC** shall be calculated for each block independently, then the lowest value graded as:

```
A (4.0) if UEC > 0.62
B (3.0) if UEC > 0.50
C (2.0) if UEC > 0.37
D (1.0) if UEC > 0.25
F (0.0) if UEC < 0.25
```

The **UEC** parameter tests the extent to which regional or spot damage in the symbol has eroded the reading safety margin that error correction provides.

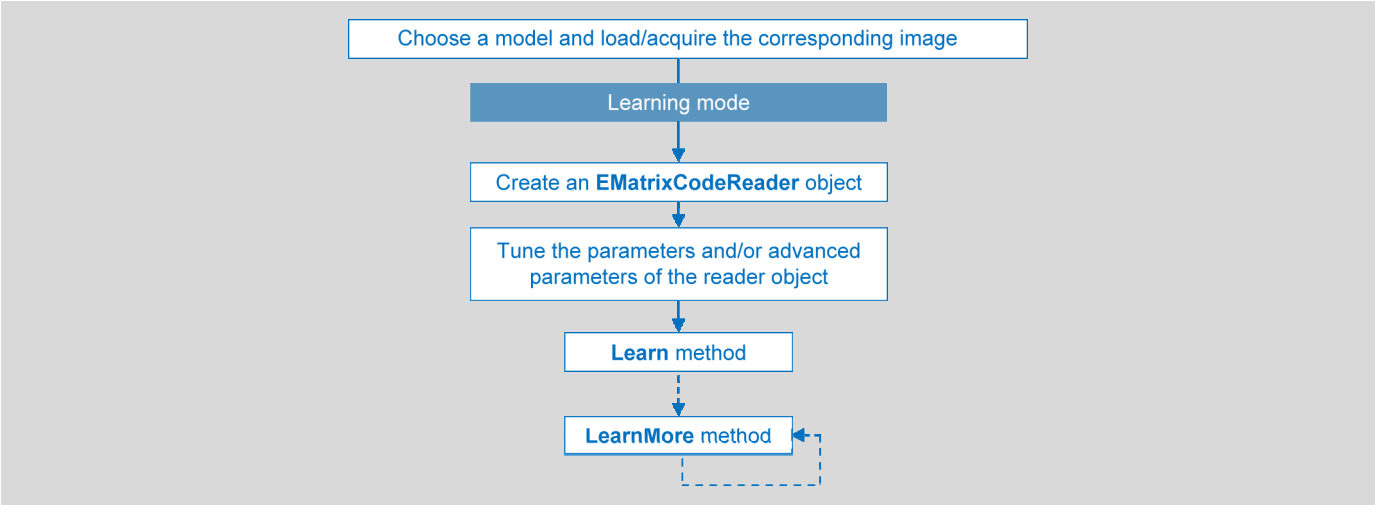
The *Overall Symbol Grade* is the lowest of the parameter grades achieved above. Table N1 summarizes the test parameters and grade levels.

Table N1. Summary of 2D Matrix Bar Code Print Quality *N.3 Overall Symbol Grade*

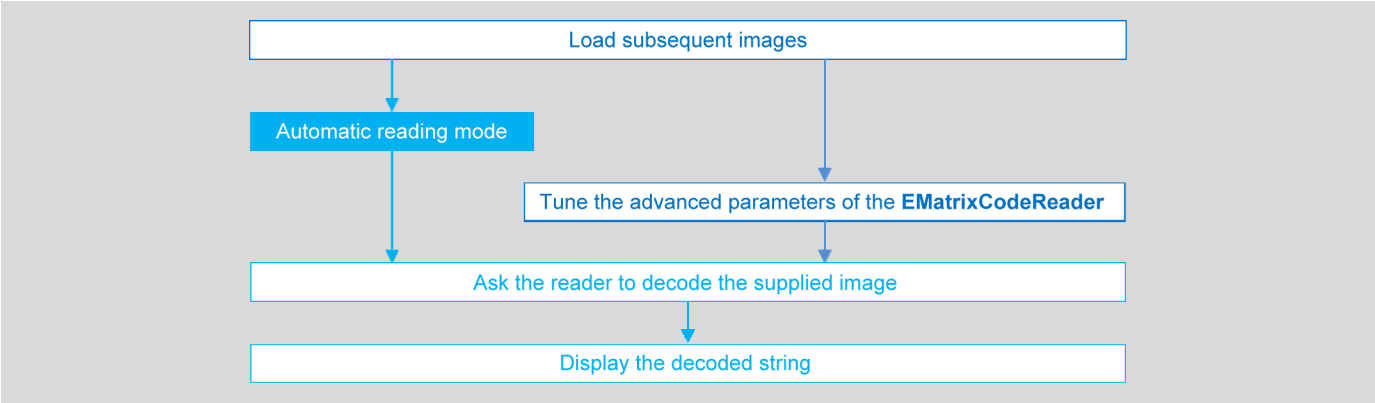
Grade	Reference Decode	Symbol Contrast	"Print" Growth	Axial Non-Uniformity	Unused Error Correction
A (4.0)	Passes	**SC >= 70 %**	**-0.50 <= D' <= 0.50**	**AN <= 0.06**	**UEC >= 0.62**
B (3.0)		**SC >= 55 %**	**-0.70 <= D' <= 0.70**	**AN <= 0.08**	**UEC >= 0.50**
C (2.0)		**SC >= 40 %**	**-0.85 <= D' <= 0.85**	**AN <= 0.10**	**UEC >= 0.37**
D (1.0)		**SC >= 20 %**	**-1.00 <= D' <= 1.00**	**AN <= 0.12**	**UEC >= 0.25**
F (0.0)	Fails	**SC < 20 %**	**D' < -1 or D' > 1**	**AN > 0.12**	**UEC < 0.25**

2. EasyMatrixCode Workflow

Note that the workflow shows the generic steps required to work in EasyMatrixCode.



Learning process (optional)



Reading process

3. Fully Automatic Reading

EasyMatrixCode provides an easy way to read a Data Matrix code. Basically, merely calling the Read method on a supplied image is sufficient. In this way of working, the Data Matrix code and its decoding are computed with any associated information (logical size, contrast type, ...).

Thus, if no special configuration has been performed, the Read method allows to read a symbol of any size, contrast, location and orientation in a single operation.

4. Reading with Prior Learning

It is possible to speed up the localization and decoding process by constraining the EMatrixCodeReader object to look for an object that matches a set of properties (EasyMatrixCode can then work faster by only searching in a subset of all the possible Data Matrix code parameters).

These constraints can be set automatically, by providing the EMatrixCodeReader with a sample matrix code image, whose properties will be stored in the reader and used for subsequent read operations. These constraints can also be set manually, by setting the parameter values that the EMatrixCodeReader will look for when searching for a Data Matrix code in an image. Beware that a significant speed-up is achieved only for some kind of images (bad conditions).

Learning Process

1. Load the learning image

2. Learn the model

The EMatrixCodeReader has parameters that allow customizing the reading process. The following parameters are relevant at reading time:

- Contrast
- Family
- Flipping
- Logical Size

3. Tune the read parameters

This means that only the matrix codes, after the Learn method call, that have the same search parameters (Contrast, Family, Flipping, Logical Size) as the learnt one will be decoded.

The SetLearnMaskElement method configures the EMatrixCodeReader to discard one of the search parameters (For example, disregard the flipping -mirrored orientation-) of the matrix code that is learnt.

This means that, after the Learn method call, only the matrix codes that have the same reader parameter as the learnt one will be decoded except for the parameters that have been removed from the learn mask.

If several matrix codes need to be learnt, then you need to call LearnMore (as the comment shows), passing additional sample images. Note that calling Learn replaces the EMatrixCodeReader processing parameters and, thus, calling Learn several times in a row does not accumulate results, which LearnMore does.

Reading Process

Ask the EMatrixCodeReader to decode the supplied image. Display the decoded string.

5. Advanced Tuning of the Search Parameters

If the fully automatic reading mode or the simple learning process do not suit your needs, it is possible to manually configure the way the EMatrixCodeReader reads the matrix codes. Individual parameters that the matrix code needs to be checked against can be stored in the EMatrixCodeReader.

For example, you can manually constrain the reader to only decode 16x16 cells matrix codes, or only matrix codes that are not flipped (see reference for an enumeration of the matrix code parameters that can be learnt). This can potentially speed up the learning process.

However, keep in mind this is an advanced feature and using the learning mode (that automatically fills these parameters when being supplied with a sample image), is suitable for most uses.

Tuning manually the read parameters

To replace a search parameter with the particular set that we wish to use, first, you will have to remove the default parameter. Then add the ones we are interested in.

6. Serialization

The process of saving to/loading from files is called serialization. Both the EMatrixCode and EMatrixCodeReader objects feature Save and Load methods that accept a file name and that respectively store and restore the object state (the reader learnt parameters, the decoded string, grading values, ...).

Saving an EMatrixCodeReader that has learnt a matrix code

1. Load the image to learn
2. Ask the reader to learn this matrix code
3. **Save** the state of the reader object

Restoring an EMatrixCodeReader from a saved file and reading

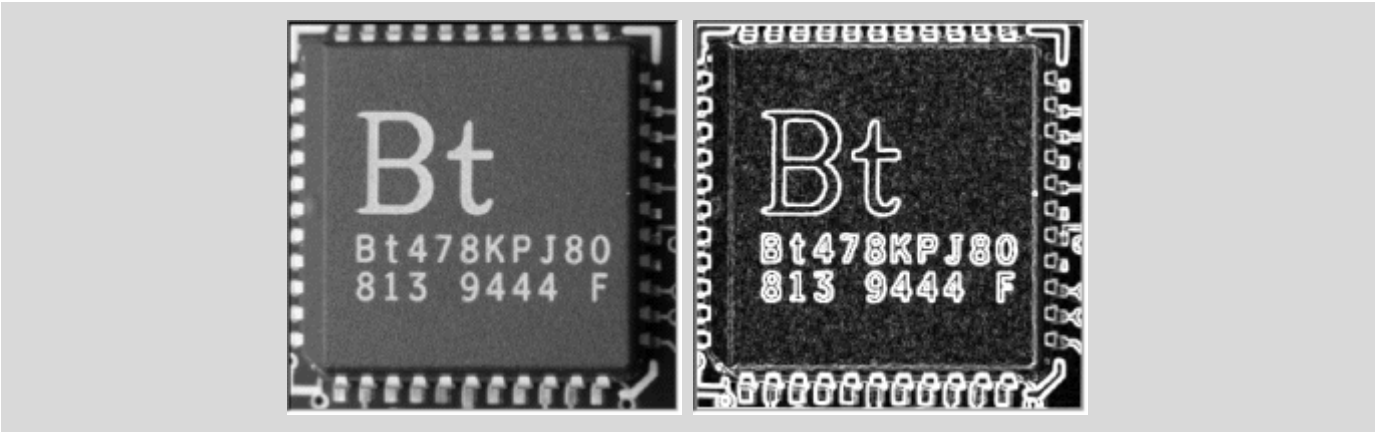
1. Load an image
2. Restore the reader state from the given file **EMatrixCodeReader::Load**
3. Then ask the reader to decode the image
4. And display the decoded string

Developing a Vision Application

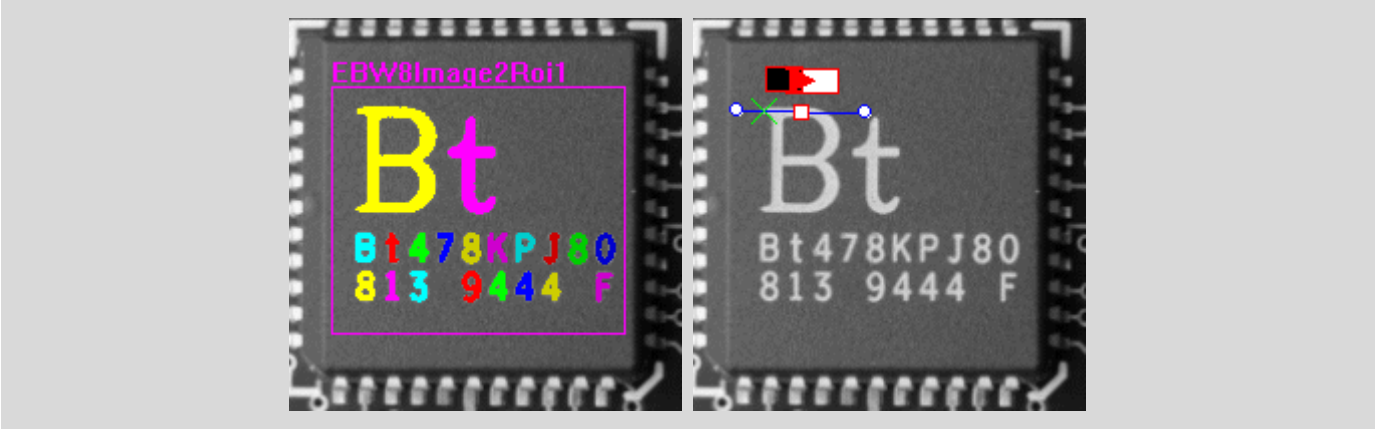
1. Solving a Vision Problem

A typical vision-based application follows a few steps:

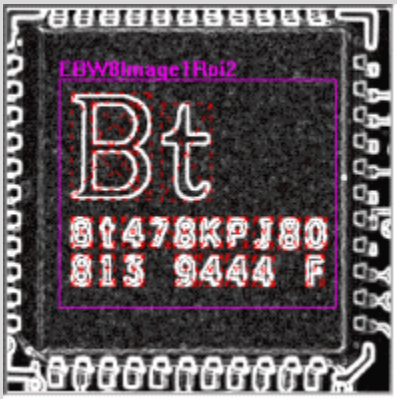
Pre-processing: the image is possibly modified in order to reduce defects that are unavoidably introduced by the image formation and acquisition stages (geometric distortion, de-focusing, noise) or enhance some desired property such as contrast between objects of interest and background.



Location: the objects or parts of interest are roughly or accurately located using various techniques such as segmentation, edge detection or pattern matching. In other cases, the same tools can be used to count the objects if their number is unknown beforehand.



Analysis: when relevant locations are determined, processing concentrates on judicious ROIs and measurements are performed locally: the shape of objects can be quantified by appropriate geometric quantities, defects can be related to abnormal gray-level values...



Area	Limit Center Y	Limit Width
252	29.00	31.00
242	60.00	31.00
535	48.00	34.00
654	44.00	54.00
246	101.00	15.00
140	101.50	11.00
180	101.00	15.00
158	101.00	14.00
235	101.50	16.00
214	101.00	15.00
197	101.00	14.00
157	101.50	13.00
226	101.50	15.00

Diagnosis: the computed quantities can be used to assess the parts quality and detect defects by comparing them to the expected values for good parts. Alternatively, these quantities can be used to recognize and sort objects, as is the case when unknown markings have to be read.

The software provides you all means to try the various tools available. It is highly recommended to start a feasibility study for a given application by acquiring sets of representative images. To achieve this, the actual setup of the targeted inspection equipment should be simulated as accurately as possible, including realistic lighting conditions. The contents of the field of view should be chosen such that the areas of interest occupy the largest part of it to maximize the useful resolution. Anyway, this should take into account the position repeatability, so that the area of interest always wholly fits in the image.

Ideally, two series of images should be considered: the first one will consist of objects without defects (those that will be accepted by the inspection process). It should exhibit the possible sources of variability such as unstable lighting conditions and movement freedom. The second series will picture objects with non-accepted defects (to be rejected by the inspection process).

Once these images are available, a resolution strategy should be designed by combining the available tools. If pre-processing is required, **EasyImage** will probably be used. For location, one of **EasyObject**, **EasyMatch**, **EasyFind** and **EasyGauge** will be useful, depending on what is known beforehand and what accuracy is required. For analysis, **EasyObject** can be used to measure totally unknown shapes; on the other hand, **EasyGauge** can return much more accurate measurements once the object is known. Additional analysis methods such as statistical parameters are provided by **EasyImage**.

Here, the goal is to choose among various features computed on suitable ROIs or on segmented objects, and to study how the values they take on the two series of images. If the set of values they take on the good images can be well separated from the set they take on bad images, then inspection will be successful. Otherwise, the set of features is not discriminatory enough and must be improved.

At each of these steps, **Open eVision Studio** will give you the opportunity to try the required functions, twiddle their working parameters in order to understand their operation and choose their appropriate value. Most of the time, it is handy to write prototype applications to automate part of the process for evaluation purposes. **Open eVision Studio** will help you as well at this stage because it acts as a code generator and will allow you to copy and paste the required lines of code immediately after you tried an operation.

2. Measuring the Execution Times

Accurate measurement of execution times is essential in real-time applications. Timing a particular piece of code is simply achieved by bracketing it with start and stop operations.

- Start the timing: **Easy::StartTiming**
- Stop the timing: **Easy::StopTiming**
- Clock resolution: **Easy::TrueTimingResolution**

3. Reporting Errors

Open eVision functions throw **exceptions** instead of returning error codes.

Each time the execution of an Open eVision function fails, an exception is thrown. To catch a potential exception, the function call is included in a try-catch block.

Exceptions feature an error code and a description.

4. Developing Thread-Safe Applications

Open eVision is multi-thread safe. It means that it is designed to support simultaneous execution by multiple threads on the same CPU. It is particularly suitable if your application includes independent tasks and allows them to be executed simultaneously. But it also requires that your application accepts sharing piece of data to be accessed by only one thread at any given time. Then, you can design such applications so that each part is controlled by a separate thread. As many threads as required can be created.

Rules for Thread-Safe Developments

The following rules allow to develop with Open eVision avoiding:

- data corruption,
- crashes,
- a program that seems to work correctly, but fails after some time, or under high CPU load.

Thread-safe basic types classes

Basic types	Recommendations	Restrictions
Basic pixel structures **EColor, EPeak, EISH, ELAB, ELCH, ELSH, ELUV, EBW1, EBW8, EBW8Path, EBW16, EBW16Path, EBW32, EC15, EC16, EC24, EC24A, EC24Path, EPath, ERGB, ERGBColor, EVSH, EXYZ, EYIQ, EYSH and EYUV**		No
Pixel collection classes **EColorLookup, EPseudoColorLookup, EPeakVector, EBW8Vector, EBWHistogramVector, EBW8PathVector, EBW16PathVector, EBW16Vector, EBW32Vector, EC24Vector, EPathVector, EColorVector, EColorLookup and EC24PathVector**	No restrictions on the read-only access.	A single instance may not be modified by several threads. If a tread is modifying an instance, no other thread can access it either for read or modify.
Images classes **EImageBW1, EImageBW8, EImageBW16, EImageBW32, EImageC15, EImageC16, EImageC24, and EImageC24A**	No restrictions on the read-only access.	A single instance may not be modified by several threads. If a tread is modifying an instance, no other thread can access it either for read or modify.
ROI classes **EROIBW1, EROIBW8, EROIBW16, EROIBW32, EROIC15, EROIC16, EROIC24, and EROIC24A**	No restrictions on the read-only access.	A single instance may not be modified by several threads. If a tread is modifying an instance, no other thread can access it either for read or modify. Different ROI can be added, removed from an image or moved event if their parent image is the same. Consequently, different threads can work on different areas of an image possibly changing in position and size during the process.

Thread-safe libraries classes

Libraries	Recommendations	Restrictions
EasyImage and EasyColor	Static methods from this class (provided the threading rules applying to their arguments are not broken).	No
EasyObject		No
EasyMatch and EasyFind **EMatcher, EMatchPosition, EPatternFinder and EFoundPattern**		A single instance can not be accessed from several threads. Search field -read-only- can be shared by the different objects.
EasyGauge and Shape subclasses **Gauging classes (EPointGauge, ELineGauge, ERectangleGauge, ECircleGauge, EWedgeGauge), EWorldShape and EFrameShape)**		They can be attached, moved or removed from different threads, even in the same hierarchy. A single instance must not be used by different threads.
Basic geometric classes (EFrame, EPoint, ECircle, ELine, ERectangle and EWedge)		Can be accessed from different threads provided that an instance is not used by two different threads simultaneously.
Gauging classes measuring and processing operations	May be executed in different threads with no blocking even if these gauges perform their measuring operations in the same image. Multiple CPU usage will be optimal.	A single instance cannot be read and modified by two threads .
EWorldShape		A single instance cannot be read and modified from different threads.
EasyOCR, EasyOCV and EasyBarCode **EOCR, EOCV, EOCVChar, EOCVText and EBarCode**	Different instances may be created and used from different threads.	A single instance cannot be accessed from several threads.
EChecker	Different instances may be used from different threads.	
EasyMatrixCode **EMatrixCodeReader**	Multiple CPU usage will be optimal.	A single instance can't be used by several threads. A single MatrixCode cannot be used in multiple threads.

5. Using Open eVision Libraries on Externally Supplied Images

The externally supplied image has to be linked to an Open eVision image object, it is mandatory that the external image is stored into a host PC memory buffer and that the address of this buffer is retrievable. More precisely, four parameters are important to link the external image to the Open eVision image:

- The address of the buffer
- The width and height of the image
- The pitch of the image buffer

For more details, consult Images and Image Construction and Memory Allocation.

6. Restricting the Processing to a Part of the Image

The processing speed of an image can be accelerated by focusing on a specific part of the image, avoiding interferences from the remainder of the image. The number of pixels to consider is then reduced.

Focusing the processing on a particular region of the image, hence isolating brings major benefits in image processing applications:

- It avoids interference from the remainder of the image.
- It speeds up the processing by reducing the amount of pixels to consider.

Open eVision provides two powerful means to restrict the processing of an image to a part of it. Each of them has specific benefits that can drive the user in the choice of one or the other method: the rectangular ROIs and the flexible masks.

ROIs

Open eVision supports **nested rectangular** ROIs, which are organized in **a hierarchical way** in each image.

ROIs are applicable to every Open eVision image processing function.

Flexible Masks

Masking a region is a powerful way to restrict the processing to a part of the image.

Applying a mask on an image allows to not consider the masked areas and consider the unmasked areas for the processing.

The Open eVision masks are flexible compared to the rectangular ROIs: they support complex and disconnected shapes.

6.1 Flexible Masks

An Open eVision mask is a BW8 image having the same height and width as the source image.

Applying a flexible mask on an image separates the image in do-care areas (that must be considered for the processing) and don't-care areas (that should not be considered for the processing).

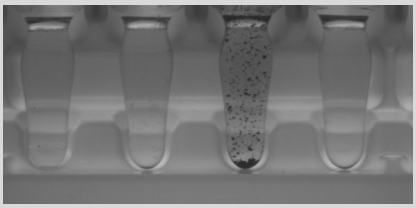
- All pixels of the flexible mask having a value of 0 define the don't-care areas
- All pixels of the flexible mask having any other value than 0 define the do-care areas

A flexible mask can be generated by any application outputting BW8 images or by the image processing functions of Open eVision.

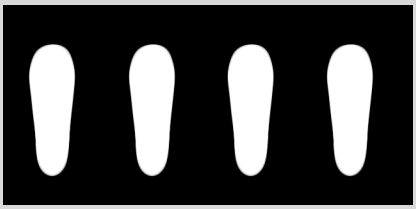
The Open eVision masks are **flexible** compared to the Open eVision rectangular ROIs. It means that the masks can have any distribution of do-care pixels, meaning that the shape of the pattern created by the do-care pixels among the don't-care pixels is freely definable.

The flexible masks are supported for selected functions of the EasyObject and EasyImage libraries.

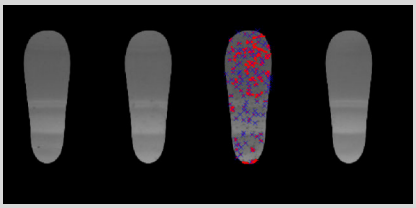
See Flexible Masks in EasyImage and Flexible Masks in EasyObject for detailed information.



A source image



Associated mask



Processing of the masked image

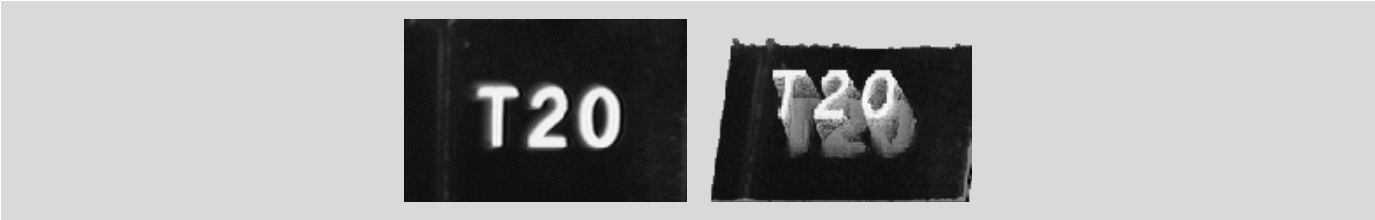
7. 3D Drawing and Rendering

The drawing functions are relative to the Images, or Vector types. Refer to these topics for the appropriate information.

7.1 3D Rendering

Easy::Render3D prepares a 3-dimensional rendering of an image, where the gray-level values are taken for altitudes. The image is viewed by rotating it around the X-axis, then around the Y-axis. Magnification factors in the three directions (X = width, Y = height and Z = depth) can be given.

The rendered image appears as independent dots. The dot size can be adjusted so that the surface appears more or less opaque.



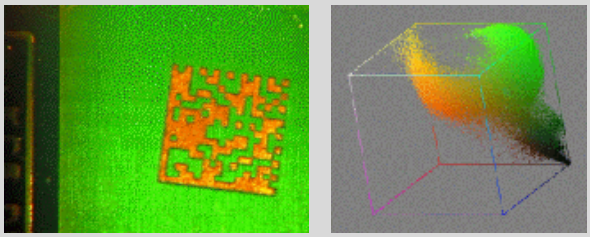
3D rendering

7.2 Color Histogram Rendering

The Easy::RenderColorHistogram function prepares a 3-dimensional rendering of a color image histogram: the pixels are drawn in the RGB space rather than in the XY-plane. This allows to observe the clustering and dispersion of the RGB values.

The image is viewed by rotating it around the X-axis, then around the Y-axis. Magnification factors in the three directions (X = red, Y = green and Z = blue) can be given.

In a more advanced version, the function prepares a 3-dimensional rendering of the pixels in another system than RGB (EasyColor provides conversion means). However, the raw RGB image must still be provided to allow the display of the pixels in their usual colors.



Color histogram rendering

Glossary

Aggregation

A process merging adjacent runs to build a coded element.

Bayer Filter

The Bayer filter is a special color image encoding format that allows to capture **color information from a single sensor**.

RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB
RGB	RGB	RGB	RGB	RGB	RGB

True color pattern

To capture color information with only one sensor, a color filter with a specific layout is placed in front of the sensor, so that part of the pixels receive green light only, while others receive red or blue only.

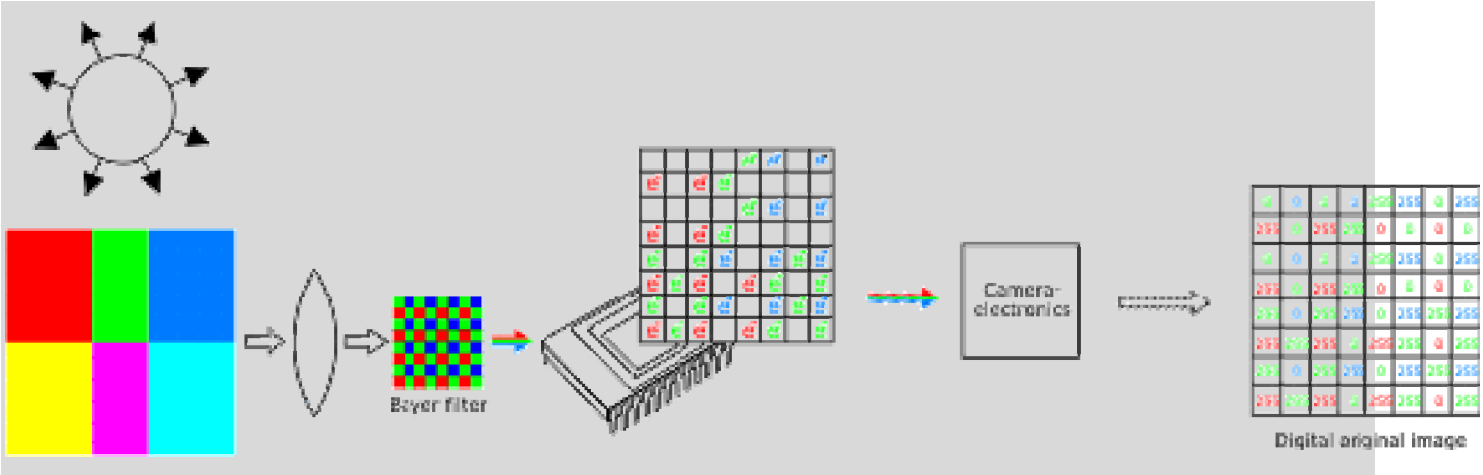
50 % of the total number of pixels are green, 25 % are red, 25 % are blue. (More pixels are dedicated to green than to red and blue, because the human eye is more sensitive to that color.)

G	B	G	B	G	B
R	G	R	G	R	G
G	B	G	B	G	B
R	G	R	G	R	G
G	B	G	B	G	B
R	G	R	G	R	G

Bayer pattern

An image encoded after the Bayer pattern has the same format as a gray-level image, and conveys three times less color data. The true horizontal and vertical resolutions are smaller than those of a true color image.

To supply missing component values and obtain a full-color image, various algorithms can be used. The complete red, green, and blue values for each point can be interpolated by mere duplication (nearest neighbor technique) or by linear interpolation.



Digitizing an image with a Bayer filter

Blob

Synonym of object.

Bilevel Image

In a bilevel image, pixel value **0** is considered as black, and any other value is considered as white.

Binarization

The process of classifying the pixels in an image as belonging to the object of interest (foreground pixels), or not (background pixels). This is a basic technique of image segmentation.

Binary Black-and-White Color System

One bit of information per pixel (**BW1**). **0** stands for black (background). **1** stands for white (foreground).

CCIR YIQ Color System

The CCIR YIQ color system is defined as a standard for the broadcast industry by the "Commission Consultative Internationale des Radiocommunications" (recommendation 601). Similar to the YUV system.

Specifies one luminance and two chrominance components (*Inphase* and *Quadrature*). These parameters are related by a linear transform to the RGB coordinates.

- The luminance carries the black and white information.
- The *I* and *Q* chrominance carry the chromatic information. They are often down-sampled without loss of visual quality. They are dephased by 33° with respect to the YUV *U* and *V* components.

CCIR YUV Color System

The CCIR YUV color system is defined as a standard for the broadcast industry by the "Commission Consultative Internationale des Radiocommunications" (recommendation 601). Similar to the YIQ system.

Specifies one luminance and two chrominance components. These parameters are related by a linear transform to the RGB coordinates.

- The luminance carries the black and white information.
- The *U* and *V* chrominance carry the chromatic information. They are often down-sampled without loss of visual quality (YUV 4:4:4 to YUV 4:2:2). They are proportional to the *Cb* and *Cr* components (B-Y and R-Y respectively).

CIE L*a*b* Color System

The CIE L*a*b* color system is similar to the L*u*v* system..

Specifies a luminance and two chrominance components. These parameters are related by a non-linear transform to the CIE XYZ coordinates, so that they represent the colors in a uniform way: two colors separated by the same distance anywhere in the gamut have the same visual contrast.

CIE L*u*v* Color System

The CIE L*u*v* color system is similar to the L*a*b* system.

Specifies a luminance and two chrominance components. These parameters are related by a non-linear transform to the CIE XYZ coordinates, so that they represent the colors in a uniform way: two colors separated by the same distance anywhere in the gamut have the same visual contrast.

CIE LCH Color System

The CIE LCH color system is derived from the L*a*b* system.

Specifies the colors as a combination of *Luminance* (achromatic perception, the L of L*a*b*), *Chroma* (degree of purity), and *Hue* (position along a colored sequence). The Chroma and Hue correspond to the a*b* coordinates expressed in polar form.

CIE XYZ Color System

The CIE XYZ color system is the standard colorimetric system defined by the "Commission Internationale de l'Eclairage" in 1931.

Specifies an additive mixture of three color primaries called X, Y and Z. The spectral composition of these primaries is tabulated by the standard. Since negative intensities are involved, the primaries cannot be physically implemented. Anyway, the XYZ system is very useful because of its absolute device independence and its ability to cover the whole gamut of the visible spectrum.

Reduced xyz coordinates (X, Y and Z over their sum) are sometimes used.

Coded Element

An object or a hole.

Continuous Tone Black-and-White Color System

Usually, one byte of information per pixel (**BW8**). **0** stands for black (background). **255** stands for white (foreground).

Notice that Open eVision also support images with 16 bits per pixel.

Device Context (Windows API)

Under Windows, when using the standard drawing functions (GDI), a device context is associated to the window. The device context maintains the current drawing attributes, such as pen and brush specifications.

In a Windows API application, the device context is referred to by means of a device context handle (HDC). This value must be passed to the Open eVision drawing functions.

In an MFC application, the device context is encapsulated in the CDC class. Member function **CDC::GetSafeHdc** provides the corresponding device context handle.

In an OWL application, the device context is encapsulated in the TDC class. The typecast member function **TDC::HDC** provides the corresponding device context handle.

In a VCL application, the device context handle can be obtained form the form canvas by using syntax **Form.Canvas.Handle**.

Diagnostics

Tolerances.

Dual Morphological Operator

The dual of an operator has the same effect, provided the white and black colors are exchanged.

For instance, eroding a light object is equivalent to dilating a dark one.

Edge Detection (Image Processing)

The process of finding boundaries of objects (or part of them) in an image.

Event (ActiveX Control)

Occurrence of a particular condition related to an ActiveX control. The application program has the opportunity to provide a handler function capable of reacting appropriately to the event.

Feret's diameter

In microscopy, Feret's diameter is the measured distance between parallel lines that are tangent to an object's profile and perpendicular to the ocular scale.

Generally, Feret's diameter is the greatest distance possible between any two points along the boundary of a region of interest.

The Feret bounding box is defined as the rectangle of minimum width that completely surrounds an object.

By contrast with the limit rectangles that are given for a fixed angle, the Feret box corresponds to the angle that minimizes the width.

The Feret width is thus the smallest object dimension between two parallel lines.

The Feret height is the other dimension of the Feret rectangle, i.e. the dimension measured in the perpendicular direction; it is not the largest dimension of the object.

Except in accidental cases, the limit center, gravity center and centroid do not coincide (even though they are in general very close to each other).

Formula for gravity center **X** (**N** = area):

$$\overline{X} = \frac{1}{N} \sum X_i$$

Formula for centroid **X** ($\mathbf{P}_i$ = pixel value):

$$\hat{X} = \frac{\sum P_i X_i}{\sum P_i}$$

Feature

Geometrical property of a coded element.

GDI

Graphics Device Interface (Windows).

Histogram Stretching

Redistribution of the gray-level values of an image, in order to exploit better the available dynamic range, and improve contrast.

Hole

In EasyObject, a hole is a maximally-sized area of adjacent connected pixels belonging to the layer background that is completely surrounded by an object.

IDE

Integrated Development Environment.

ISH Color System

The ISH color system is often called HSI. Based on the RGB system. Very similar to LSH and VSH.

Specifies the colors as a combination of *Intensity* (achromatic perception, average of the RGB components), *Saturation* (degree of purity) and *Hue* (position along the sequence red/yellow/green/cyan/blue/magenta).

Landmark

Feature point in an image that can be accurately located directly or indirectly (center of a shape, intersection of edges, ...). When an image must be realigned with respect to another one, landmarks can be matched together.

Layer

A binary image constructed by a segmentation process.

Layer Background

For a layer, the pixels that are not set in the underlying binary image.

Layer Foreground

For a layer, the pixels that are set in the underlying binary image.

LSH Color System

The LSH color system is often called HLS. Based on the RGB system. Very similar to ISH and VSH.

Specifies the colors as a combination of *Lightness* (achromatic perception, half sum of the minimum and maximum of the RGB components), *Saturation* (degree of purity) and *Hue* (position along the sequence red/yellow/green/cyan/blue/magenta).

MFC

Microsoft Foundation Classes.

Object

In a general content, the term object should be understood with the meaning of a class instance.

In EasyObject, an object is a maximally-sized area of adjacent connected pixels belonging to the layer foreground.

OWL

Object Windows Library (Borland/CodeGear).

Pattern Matching (Image Processing)

The process of locating a specific *pattern* (small image) in an image in order to measure an object's position.

Preparing the Model

Calibration.

Quantized Color

A quantized color is a triple of integer values representing discrete color components. All component values are defined in the [0..255] interval and stored as unsigned bytes (also see *Color System Variants*). An **EC24Image** image is an array of such quantized colors.

Relative Thresholding

Thresholding in such a way that a known fraction of the image pixels lie above the threshold.

RGB Color System

The RGB true color system specifies an additive mixture of red, green and blue components. Usually, three bytes of information per pixel (**EC24**).

Run

Maximally-sized area of adjacent pixels on the same row belonging to the background or to the foreground of a layer.

Segmentation (Image Processing)

The process of decomposing an image into areas corresponding to objects (or parts of them).

Thresholding

A simple segmentation method based on comparisons of the value of individual pixels with a threshold value.

Unquantized Color

An unquantized color is a triple of 32 bits floating point values representing continuous color components. All component values are defined in the [0..1] interval, with the following exceptions:

- **YUV**: U and V in range [-0.5..+0.5]
- **YIQ**: I and Q in range [-0.5..+0.5]
- **L*a*b***: L* in range [0..100[, a* and b* in range [-128..+128]
- **LCH**: L in range [0..100[, C in range [0..256[
- **L*u*v***: L* in range [0..100[, u* and v* in range [-128..+128]

The hues are computed in revolutions rather than degrees and thus contained in the [0..1] interval as well.

VB

Visual Basic.

VCL

Visual Classes Library (Borland/CodeGear).

VSH Color System

The VSH color system is often called HSV. Based on the RGB system. Very similar to ISH and LSH.

Specifies the colors as a combination of *Value* (achromatic perception, maximum of the RGB components), *Saturation* (degree of purity) and *Hue* (position along the sequence red/yellow/green/cyan/blue/magenta).

YSH Color System

The YSH color system is derived from the YUV system.

Specifies the colors as a combination of *Luminance* (achromatic perception, the *Y* of YUV), *Saturation* (degree of purity), and *Hue* (position along a colored sequence). The Saturation and Hue correspond to the UV coordinates expressed in polar form.

Zooming and Panning (Display)

Zooming allows you to enlarge or shrink an image or drawing (the scale factor being respectively smaller or larger than unity). Two separate positive scale factors named **ZoomX** and **ZoomY** can be specified. When they are unequal, anisotropic zooming results (circles become ellipses). Setting **ZoomY** to **0** always has the same effect as setting **ZoomY = ZoomX**. The origin point **(0,0)** remains fixed.

Panning allows you to translate an image or drawing. Two translation components named **PanX** and **PanY** can be specified. They are counted as positive leftwards and downwards respectively.

The panning factors are applied first, before zooming. The zooming/panning transform corresponds to equations:

```
X' = ZoomX * (X + PanX)
Y' = ZoomY * (Y + PanY)
```

Disclaimer

EURESYS s.a. shall retain all property rights, title and interest of the documentation of the hardware and the software, and of the trademarks of EURESYS s.a.

All the names of companies and products mentioned in the documentation may be the trademarks of their respective owners.

The licensing, use, leasing, loaning, translation, reproduction, copying or modification of the hardware or the software, brands or documentation of EURESYS s.a. contained in this book, is not allowed without prior notice.

EURESYS s.a. may modify the product specification or change the information given in this documentation at any time, at its discretion, and without prior notice.

EURESYS s.a. shall not be liable for any loss of or damage to revenues, profits, goodwill, data, information systems or other special, incidental, indirect, consequential or punitive damages of any kind arising in connection with the use of the hardware or the software of EURESYS s.a. or resulting of omissions or errors in this documentation.

Index

3

3D Drawing and Rendering • 292
3D Rendering • 292

A

Accessing the Model Components • 249
Adjusting Inspection Parameters from Statistics • 253
Advanced Features • 255
Advanced Tuning of the Search Parameters • 280
Aggregation • 294
Arithmetic and Logic • 54
Automatic Thresholding • 51

B

Basic Types and Operations • 16
Bayer Filter • 294
Bayer Patterns to RGB • 98
Bilevel Image • 295
Binarization • 296
Binary Black-and-White Color System • 296
Blob • 295
Building the Threshold Images • 239

C

Calculating a Threshold Value • 52
Calibration and Unwarping • 195
Calibration by Guesswork • 203
Calibration Coefficients • 208
Calibration using EWorldShape • 203
Canny Edge Detector • 82
CCIR YIQ Color System • 296
CCIR YUV Color System • 296
Checksum • 265
Choosing the Pattern • 139
CIE L*a*b* Color System • 296
CIE L*u*v* Color System • 296
CIE LCH Color System • 297
CIE XYZ Color System • 297
Circle Gauge • 190
Coded Element • 297
Color Classification for Segmentation • 97
Color Histogram Rendering • 292
Color Image Components • 96
Color Image Processing • 90
Color Systems • 89
Color Transformations • 92
Colorimetric Systems Conversion • 92
Colors • 38
Command-Line Installation • 11
Computable Features • 124
Computing Inspection Statistics • 253
Computing LUT for Specific Usages • 92
Computing the Coded Element Features • 124
Consistent Edges • 159
Constructing the Blobs • 107
Continuous Tone Black-and-White Color System • 297
Contrasting Regions • 161
Coordinate Transform • 211
Customized Setup • 10

D

Dealing with Strings that Contain NULL Characters • 266
Defining the Inspected ROI • 238
Defining the Pattern and Search ROIs • 237
Degrees of Freedom for Location • 227
Detecting Corners on a Binary Image • 63
Detection of Other Versions and Uninstallation • 9

Developing a Vision Application • 282
Developing Thread-Safe Applications • 287
Device Context (Windows API) • 297
Diagnostics • 297
Disclaimer • 2, 302
Don't Care Area • 153
Dot-Grid-Based Calibration • 206
Drawing the Inspected Items • 252
Dual Morphological Operator • 298

E

EasyBarCode • 257
EasyBarCode Workflow • 260
EasyColor • 87
EasyFind • 143
EasyFind Workflow • 145
EasyGauge • 168
EasyImage • 39
EasyMatch • 135
EasyMatch Workflow • 138
EasyMatch Working Principles • 136
EasyMatrixCode • 267
EasyMatrixCode Workflow • 277
EasyObject • 101
EasyObject Workflow • 104
EasyOCR • 213
EasyOCR Workflow • 215
EasyOCV • 223
ECC 000 • 269
ECC 050 • 270
ECC 080 • 271
ECC 100 • 271
ECC 140 • 272
ECC 200 • 273
EChecker Concept • 235
Edge Detection (Image Processing) • 298
Edge Feature Points • 155
Erosion, Dilation • 58
Event (ActiveX Control) • 298

F

Feature • 298
Feature Model • 163
Feature Point Detectors • 81
Feature Points • 155
Feret's diameter • 298
Finding in a Search Field • 164
Flexible Masks • 290
Flexible Masks in EasyImage • 43
Flexible Masks in EasyObject • 131
Fully Automatic Reading • 278

G

Gain/Offset • 46
Gauge Grouping • 193
GDI • 298
Generating a Flexible Mask from an Encoded Image • 132
Geometric Transforms • 66
Glossary • 294
Graphical Interaction • 29, 194
Gray-Level Normalization • 234

H

Harris Corner Detector • 81
Histogram Equalization • 74
Histogram Stretching • 299
Histogram-Based Threshold • 52
Histograms • 73
Hit-and-Miss Transform • 62
Hole • 299
Holes Construction • 119
Horizontal and Vertical Mirroring • 66
How to Choose a Good Model? • 148

I

IDE • 299
Image Buffer • 23
Image Comparison • 230
Image Comparison with EChecker • 230
Image Construction and Memory Allocation • 26
Image Drawing and Overlay • 27
Image Encoder • 112
Image File Access - Save, Load • 19

Image Files • 18
Image Focus • 75
Image Main Properties • 17
Image Pixel Structures and Image File Types Compatibility • 22
Image Registration - Alignment • 66
Image Segmenters • 107
Image Types • 18
Image Unwarping • 211
Images • 17
Inspecting • 251
Inspecting with EChecker • 241
Inspection Process • 227
Installation Options • 9
Installation under WES 2009 • 8
Installing Open eVision • 6
Intensity Scale Transformation Functions • 46
Interactively Editing a Model • 254
Introduction • 251
ISH Color System • 299

K

Kernel Size Specification • 57
Key to Success of a Thresholding Operation • 53

L

Landmark • 299
Landmark-Based Calibration • 205
Layer • 299
Layer Background • 299
Layer Foreground • 299
Learning a Model • 148
Learning and Matching Processes • 140
Learning Characters and Creating a Font File • 216
Learning Parameters • 150
Learning Passes • 238
Learning Results • 153
Licenses Activation • 14
Line Gauge • 189
Localizing Bar Code Automatically • 262
Localizing Bar Code Manually • 263
Lookup Mapping • 49
Lookup Tables (LUTs) • 92
LSH Color System • 299
LUT for Color Balance • 94
LUT for Color Calibration • 94
LUT for Gain/Offset (Color) • 93
LUT-Based (Un)Warping • 68

M

Manual Threshold Modes • 51
Matrix Codes Sizes and Data Capacity • 269
Measuring the Execution Times • 285
Median Filter • 65
Merging and Extracting Image Components • 96
MFC • 300
Model Fitting • 186
Mono- or Multi-Symbology Modes • 261
Morphological Distance • 64
Morphological Gradient • 64
Morphological Operators • 57

N

Noise Estimation • 78
Noise Reduction and Estimation • 76
Non-Linear Filtering • 57
Normal vs. Continuous Mode • 116
Normalization • 47
Normalized Print Quality Indicators • 274

O

Object • 300
Open eVision Functional Guide • 3
Open eVision Libraries • 4
Opening, Closing • 59
Operation on Interlaced Video Frames • 80
Overlay • 85
OWL • 300

P

Packages • 9
Pattern Matching (Image Processing) • 300
Pattern Types • 159
Peak Selection • 176

Pixel Data Access • 37
Pixel Types • 37
Pixels • 37
Point Location • 173
Point Probe Position • 181
Preparing the Model • 300
Print Quality Control • 274
Profile • 69
Programming with EasyOCV • 251
Projections • 69
Pseudo-Coloring • 96

Q

Quality Indicators • 243
Quantized Color • 300
Quantized vs. Continuous Values • 90

R

Reading Bar Code • 264
Reading with Prior Learning • 279
Realignment • 233
Recognition Parameters • 220
Recognizing Characters • 219
Rectangle Gauge • 190
Region Feature Points • 158
Relative Thresholding • 300
Reported Information • 167
Reporting Errors • 286
Restricting the Processing to a Part of the Image • 290
Retrieving the Diagnostics and Quality Indicators • 252
RGB Color System • 300
ROI Construction and Attachment • 29
ROI Main Properties • 28
ROI Types • 28
ROIs • 28
Run • 300

S

Saving/Loading an EasyFind Model • 166
Scalar Gradient • 86
Segmentation (Image Processing) • 300
Segmentation Parameters • 217
Selecting and Sorting the Blobs • 122
Selecting Items Interactively • 252
Sensor Coordinates (Pixels) • 197
Serialization • 281
Setting Inspection Parameters • 252
Setting the ROI for Inspection • 251
Sliding Window Statistics • 72
Solving a Vision Problem • 283
Spatial Noise Reduction • 76
Special Color Formats • 98
Statistical Training • 232
Statistics • 71, 250
Supported Image File Types • 18
Supported Platforms and Requirements • 7
Supported Symbologies • 258

T

Template Model Definition • 246
Template Model Structure • 225
Temporal Noise Reduction • 77
Thin Structure • 160
Thinning, Thickening • 61
Third-Party Libraries or Components • 15
Thresholding • 51, 301
Top-Hat Filters • 61
Training with EChecker • 235
Transition Feature Points • 157
Translation, Scaling and Rotation with Optional Interpolation • 67
Transverse Filtering • 179
Trying the Reference Images • 238
Tuning the Inspection Parameters • 242
Tuning the Learning Process • 239
Tuning the Matching Parameters • 142
Tuning the Point Measurement Parameters • 181

U

Uniformization • 48
Unquantized Color • 301
Using Flexible Masks • 131

Using Flexible Masks in EasyImage • 44
Using Open eVision Libraries on Externally Supplied Images • 289

V

VB • 301
VCL • 301
Vector Construction and Usage • 36
Vector Drawing • 36
Vector Operations • 69
Vectors • 31
Vectors Types and Main Properties • 31
VSH Color System • 301

W

Wedge Gauge • 191
What Are Data Matrix Codes? • 268
What Is a Bar Code? • 258
What Is Calibration? • 197
What Is Color? • 89
What Is EasyBarCode? • 258
What Is EasyColor? • 88
What Is EasyFind? • 144
What Is EasyGauge? • 169
What Is EasyImage? • 40
What Is EasyMatch? • 136
What Is EasyMatrixCode? • 268
What Is EasyObject? • 102
What Is EasyOCR? • 214
What Is EasyOCV? • 224
What Is EChecker? • 235
What Is Point Location? • 173
World Coordinates (Physical Units) • 198
World-to-Sensor Transform • 199

Y

YSH Color System • 301
YUV 422 Decompression • 98

Z

Zooming and Panning (Display) • 301

Contents

Open eVision Libraries 4

Installing Open eVision 6

1. Supported Platforms and Requirements 7
2. Installation under WES 2009 8
3. Installation Options 9
   - 3.1 Detection of Other Versions and Uninstallation 9
   - 3.2 Packages 9
   - 3.3 Customized Setup 10
4. Command-Line Installation 11
5. Licenses Activation 14
6. Third-Party Libraries or Components 15

Basic Types and Operations 16

1. Images 17
   - 1.1 Image Main Properties 17
   - 1.2 Image Types 18
   - 1.3 Image Files 18
   - 1.4 Image Pixel Structures and Image File Types Compatibility 22
   - 1.5 Image Buffer 23
   - 1.6 Image Construction and Memory Allocation 26
   - 1.7 Image Drawing and Overlay 27
2. ROIs 28
   - 2.1 ROI Main Properties 28
   - 2.2 ROI Types 28
   - 2.3 ROI Construction and Attachment 29
   - 2.4 Graphical Interaction 29
3. Vectors 31
   - 3.1 Vectors Types and Main Properties 31
   - 3.2 Vector Construction and Usage 36
   - 3.3 Vector Drawing 36

4. Pixels 37
4.1 Pixel Types 37
4.2 Pixel Data Access 37
5. Colors 38

EasyImage 39

1. What Is EasyImage? 40
2. Flexible Masks in EasyImage 43
2.1 Using Flexible Masks in EasyImage 44
3. Intensity Scale Transformation Functions 46
3.1 Gain/Offset 46
3.2 Normalization 47
3.3 Uniformization 48
3.4 Lookup Mapping 49
4. Thresholding 51
4.1 Automatic Thresholding 51
4.2 Manual Threshold Modes 51
4.3 Histogram-Based Threshold 52
4.4 Calculating a Threshold Value 52
4.5 Key to Success of a Thresholding Operation 53
5. Arithmetic and Logic 54
6. Non-Linear Filtering 57
6.1 Morphological Operators 57
6.2 Median Filter 65
7. Geometric Transforms 66
7.1 Image Registration - Alignment 66
7.2 Horizontal and Vertical Mirroring 66
7.3 Translation, Scaling and Rotation with Optional Interpolation 67
7.4 LUT-Based (Un)Warping 68
8. Vector Operations 69
8.1 Projections 69
8.2 Profile 69
9. Statistics 71
9.1 Sliding Window Statistics 72
9.2 Histograms 73
9.3 Histogram Equalization 74
9.4 Image Focus 75

10. Noise Reduction and Estimation 76
10.1 Spatial Noise Reduction 76
10.2 Temporal Noise Reduction 77
10.3 Noise Estimation 78
11. Operation on Interlaced Video Frames 80
12. Feature Point Detectors 81
12.1 Harris Corner Detector 81
12.2 Canny Edge Detector 82
13. Overlay 85
14. Scalar Gradient 86

EasyColor 87

1. What Is EasyColor? 88
1.1 What Is Color? 89
1.2 Color Systems 89
1.3 Color Image Processing 90
2. Color Transformations 92
2.1 Lookup Tables (LUTs) 92
2.2 Computing LUT for Specific Usages 92
3. Color Image Components 96
3.1 Merging and Extracting Image Components 96
3.2 Pseudo-Coloring 96
4. Color Classification for Segmentation 97
5. Special Color Formats 98
5.1 YUV 422 Decompression 98
5.2 Bayer Patterns to RGB 98

EasyObject 101

1. What Is EasyObject? 102
2. EasyObject Workflow 104
3. Constructing the Blobs 107
3.1 Image Segmenters 107
3.2 Image Encoder 112
3.3 Normal vs. Continuous Mode 116
3.4 Holes Construction 119
4. Selecting and Sorting the Blobs 122

5. Computing the Coded Element Features 124
5.1 Computable Features 124
6. Flexible Masks in EasyObject 131
6.1 Using Flexible Masks 131
6.2 Generating a Flexible Mask from an Encoded Image 132

EasyMatch 135

1. What Is EasyMatch? 136
1.1 EasyMatch Working Principles 136
2. EasyMatch Workflow 138
3. Choosing the Pattern 139
4. Learning and Matching Processes 140
5. Tuning the Matching Parameters 142

EasyFind 143

1. What Is EasyFind? 144
2. EasyFind Workflow 145
3. Learning a Model 148
3.1 How to Choose a Good Model? 148
3.2 Learning Parameters 150
3.3 Don't Care Area 153
3.4 Learning Results 153
4. Feature Points 155
4.1 Edge Feature Points 155
4.2 Transition Feature Points 157
4.3 Region Feature Points 158
5. Pattern Types 159
5.1 Consistent Edges 159
5.2 Thin Structure 160
5.3 Contrasting Regions 161
6. Feature Model 163
7. Finding in a Search Field 164
8. Saving/Loading an EasyFind Model 166
9. Reported Information 167

EasyGauge 168

1. What Is EasyGauge? 169
2. Point Location 173
   2.1 What Is Point Location? 173
   2.2 Peak Selection 176
   2.3 Transverse Filtering 179
   2.4 Point Probe Position 181
   2.5 Tuning the Point Measurement Parameters 181
3. Model Fitting 186
   3.1 Line Gauge 189
   3.2 Circle Gauge 190
   3.3 Rectangle Gauge 190
   3.4 Wedge Gauge 191
4. Gauge Grouping 193
5. Graphical Interaction 194
6. Calibration and Unwarping 195
   6.1 What Is Calibration? 197
   6.2 Sensor Coordinates (Pixels) 197
   6.3 World Coordinates (Physical Units) 198
   6.4 World-to-Sensor Transform 199
   6.5 Calibration using EWorldShape 203

EasyOCR 213

1. What Is EasyOCR? 214
2. EasyOCR Workflow 215
3. Learning Characters and Creating a Font File 216
   3.1 Segmentation Parameters 217
4. Recognizing Characters 219
   4.1 Recognition Parameters 220

EasyOCV 223

1. What Is EasyOCV? 224
2. Template Model Structure 225
3. Inspection Process 227
   3.1 Degrees of Freedom for Location 227

3.2 Image Comparison with EChecker 230
4. EChecker Concept 235
4.1 What Is EChecker? 235
4.2 Training with EChecker 235
4.3 Defining the Pattern and Search ROIs 237
4.4 Trying the Reference Images 238
4.5 Defining the Inspected ROI 238
4.6 Learning Passes 238
4.7 Building the Threshold Images 239
4.8 Tuning the Learning Process 239
4.9 Inspecting with EChecker 241
4.10 Tuning the Inspection Parameters 242
5. Quality Indicators 243
6. Template Model Definition 246
7. Accessing the Model Components 249
8. Statistics 250
9. Programming with EasyOCV 251
9.1 Introduction 251
9.2 Setting the ROI for Inspection 251
9.3 Inspecting 251
9.4 Drawing the Inspected Items 252
9.5 Retrieving the Diagnostics and Quality Indicators 252
9.6 Setting Inspection Parameters 252
9.7 Selecting Items Interactively 252
9.8 Computing Inspection Statistics 253
9.9 Adjusting Inspection Parameters from Statistics 253
9.10 Interactively Editing a Model 254
9.11 Advanced Features 255

EasyBarCode 257

1. What Is EasyBarCode? 258
1.1 What Is a Bar Code? 258
1.2 Supported Symbologies 258
2. EasyBarCode Workflow 260
3. Mono- or Multi-Symbology Modes 261
4. Localizing Bar Code Automatically 262
5. Localizing Bar Code Manually 263

6. Reading Bar Code 264
7. Checksum 265
8. Dealing with Strings that Contain NULL Characters 266

EasyMatrixCode 267

1. What Is EasyMatrixCode? 268
   1.1 What Are Data Matrix Codes? 268
   1.2 Print Quality Control 274
2. EasyMatrixCode Workflow 277
3. Fully Automatic Reading 278
4. Reading with Prior Learning 279
5. Advanced Tuning of the Search Parameters 280
6. Serialization 281

Developing a Vision Application 282

1. Solving a Vision Problem 283
2. Measuring the Execution Times 285
3. Reporting Errors 286
4. Developing Thread-Safe Applications 287
5. Using Open eVision Libraries on Externally Supplied Images 289
6. Restricting the Processing to a Part of the Image 290
   6.1 Flexible Masks 290
7. 3D Drawing and Rendering 292
   7.1 3D Rendering 292
   7.2 Color Histogram Rendering 292

Glossary 294

Index 303

Contents 309