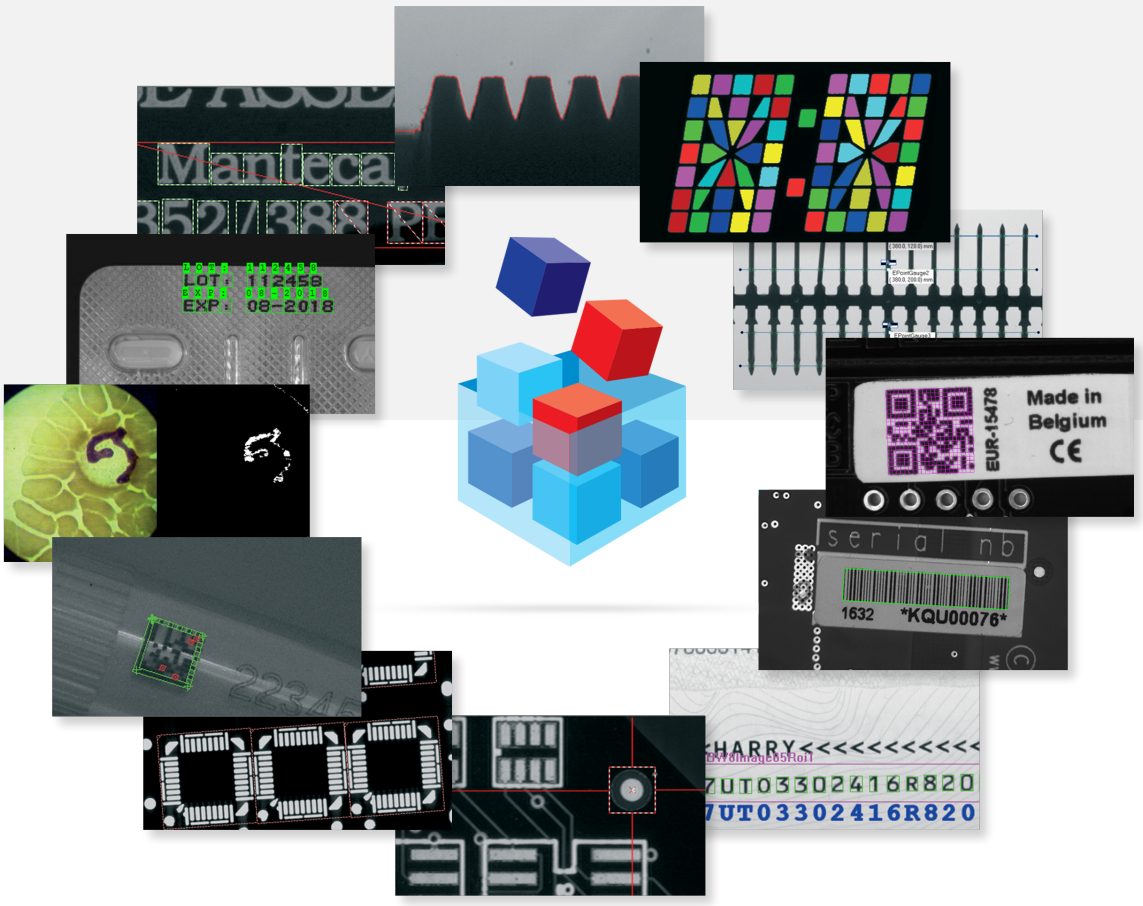
EURESYS™
Excellence in vision

FUNCTIONAL GUIDE

Open eVision

2.1.0



Terms of Use

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.

This documentation is provided with Open eVision 2.1.0 (doc build 2017-06-21).

Contents

Installing Open eVision

INSTALLER PACKAGE

Open eVision comes as a single installer package **"Open_eVision_Installer_2.1.0.msi"**. It contains everything needed to run or develop applications using **Open eVision**

Installation Types

THE OPEN EVISION INSTALLER PROVIDES THE FOLLOWING INSTALLATION TYPES:

- **Complete**: Everything needed for running or developing applications is installed on the system.
- **Typical**: Same as Complete, with the exception of Legacy components and VC++ 6.0 specific components.
- **Runtime**: Installs all binaries needed to run applications using Open eVision on the system.
- **Custom**: Allows to select exactly what components will be installed on the system.

Older Versions

Open eVision will not replace other Open eVision major versions, but install alongside. If the major version is identical, minor versions releases as well as maintenance releases will update automatically

Command-Line Interface

To install Open eVision with the command line, use:

msiexec /i "Open_eVision_Installer_2.1.0.msi" /qn INSTALLTYPE=[install_type]"

Where *[install_type]* can be **Complete**, **Typical** or **Runtime**. By default, installation type is *'Typical'*.

For the command prompt to wait for the end of the installation add *'start /wait'* at the start of the command:

start /wait msiexec /i "Open_eVision_Installer_2.1.0.msi" /qn INSTALLTYPE=[install_type]"

LICENSE ACTIVATION

Open eVision licenses are activated from the Open eVision License MAnager. The License MAnager can be launched at the end of the installation, or from the Windows start menu.

Note: On Windows XP, the license Manager requires .NET 2.0.

SUPPORTED PLATFORMS AND REQUIREMENTS

WES 2009

Windows Embedded Standard 2009 can install drivers and applications after FBA completes.

It is recommend to install Open eVision as follows:

1. **Add mandatory components to the Run-Time Image using Target Designer**
   - **.Net framework 3.0 setup** component: For Open eVision license manager
   - **Windows Installer Service** component: For Open eVision to install C/C++ run-time libraries.
   - Open eVision license manager needs either the internet support components for online activation, or USB Flash for an offline activation:.
   - **Sysprep** (Windows System preparation) **Component** or **System Cloning Tool Component**: To reseal the "**Master Target**" before deploying the image to multiple devices.
     Note: To prevent **System Cloning Tool** from executing **FBRESEAL** automatically when FBA finishes:
     Set System Cloning Tool Settings **Reseal Phase** to `Manual`, or change **System Cloning Tool Advanced Settings** `cmiResealPhase` from **12000** to **0**.
2. **Install Open eVision 2.1 and newer on the "Master Target"**
   - Once the Pre FBA OS Image has been built, boot the "Master Target" and allow FBA to complete.
   - When the "Master Target" 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, include all the run-time libraries needed by your application.
3. **Reseal the master package ready for mass deployment**
   - Run **FBRESEAL** or **Sysprep**. Once the computer shuts down, this image is the master.
   - Each time you deploy the "Master" in a new device, the Open eVision libraries need to be activated as described in the license manager documentation.

OPEN EVISION IN C++

Include the header (Open_evVsion_2_0.h) located in the installation folder > Include subfolder. No linker settings are required.

Microsoft Visual Studio C++ environments automatically adds the Open eVision Include folder at installation time. This must be done manually for Borland/CodeGear C++ environments.

OPEN EVISION IN .NET

Add a reference to the Open_eVision_NetApi_2_1.dll in the development environment. No other DLL must be copied.

USING OPEN EVISION IN ACTIVEX

Add a reference to the Open_eVision_ActiveXApi_2_1.dll component.

VISUAL STUDIO 6.0

- If you are using the regular API (new style API with exceptions and namespaces)
  1. Open your project settings, and add the following preprocessor macro definition:
     `DO_NOT_USE_INLINE_OPEN_EVISION_2_1`

2. Add the **Open_evVsion_2_0_VC6_Release.lib** and **Open_evVsion_2_0_VC6_Debug** in the corresponding configuration linker settings.
   These files are in the Open eVision installation folder.

- If you are using the legacy support API (compatible with eVision 6.7.1):
  1. Open your project settings, and add the following preprocessor macro definition:
     `DO_NOT_USE_INLINE_LEGACY_OPEN_EVISION_2_1`
  2. Add the **Legacy_Open_evVsion_2_0_VC6_Release.lib** and **Legacy_Open_evVsion_2_0_VC6_Debug** in the corresponding configuration linker settings.
     These files are in the Open eVision installation folder.
- If you are using both the regular API and the legacy support API, you must perform all steps (and thus all the relevant libraries to your solution).

In order to use these libraries, your program must use the Multithreaded DLL (/MD) or Multithreaded Debug DLL (/MDd) code generation flags.

Visual Basic 6.0

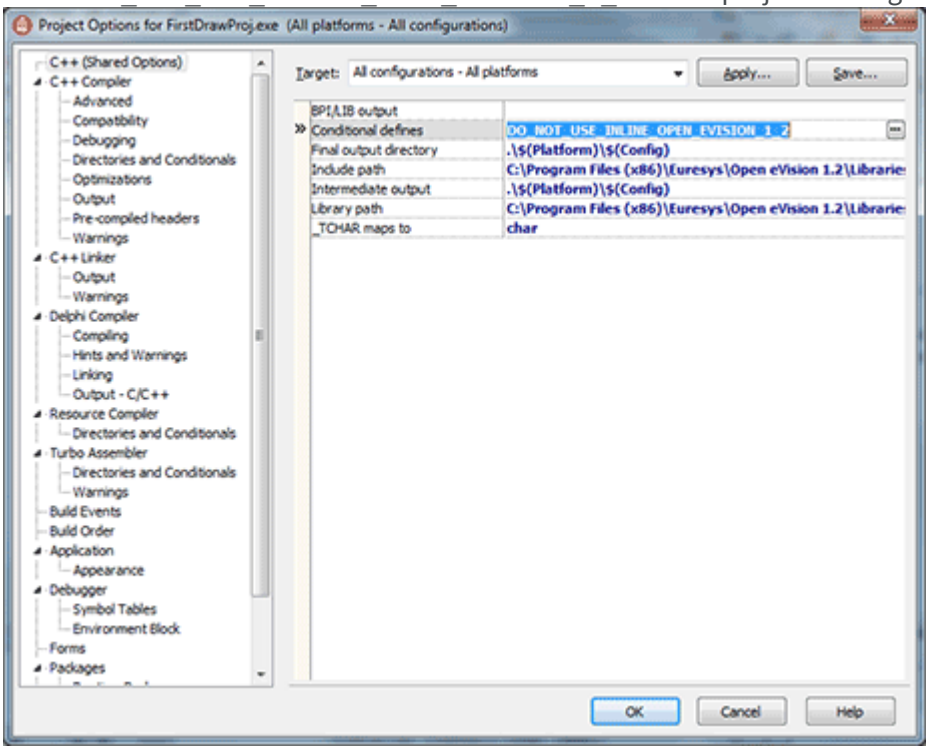
Add a reference to the Open_eVision_ActiveXApi_2_1.dll (menu: "Project > Add Reference"). All objects are then directly usable in Visual Basic.

EMBARCADERO RAD STUDIO XE4/XE5

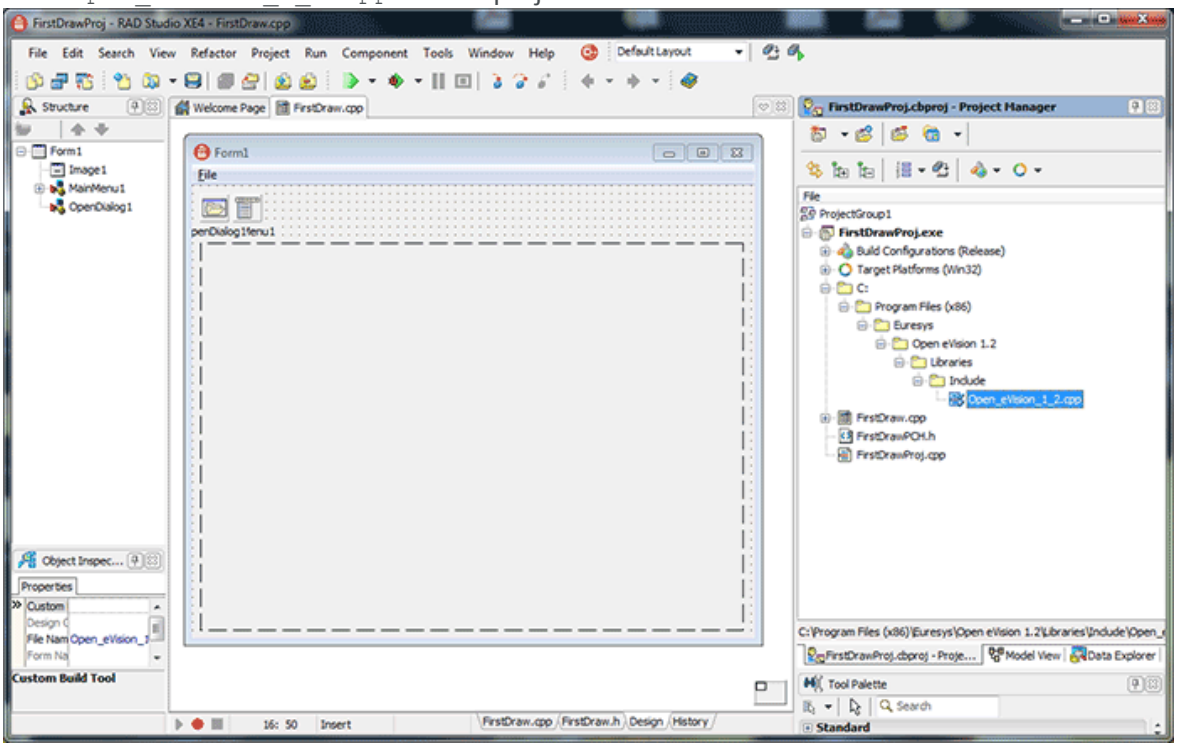
To configure projects in Embarcadero RAD Studio XE4 and XE5 to use Open eVision:

C++

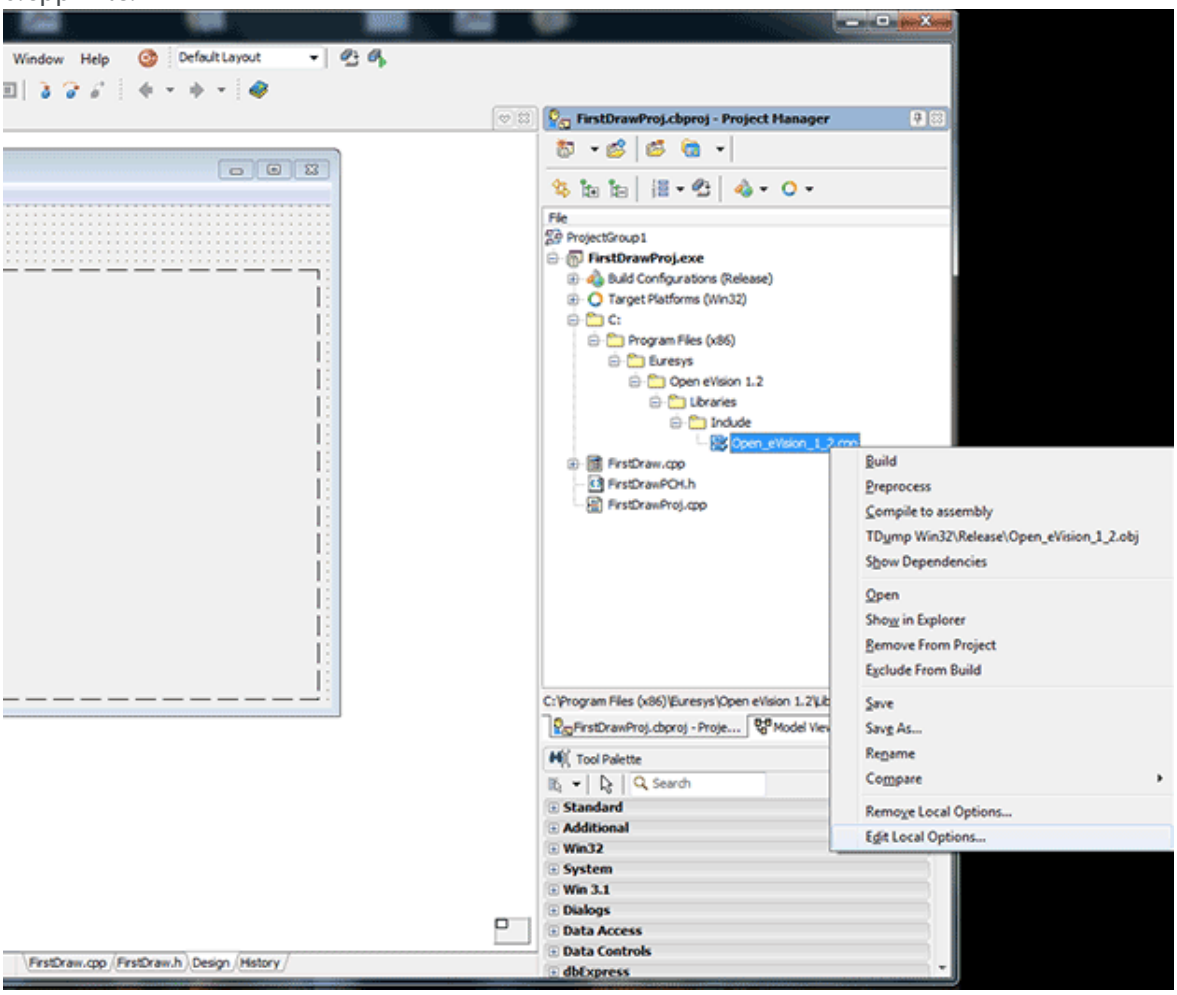
1. Create a new C++ project.
2. Add the Open eVision include path to the project dependencies.
3. Define `DO_NOT_USE_INLINE_OPEN_EVISION_2_1` in the project settings.



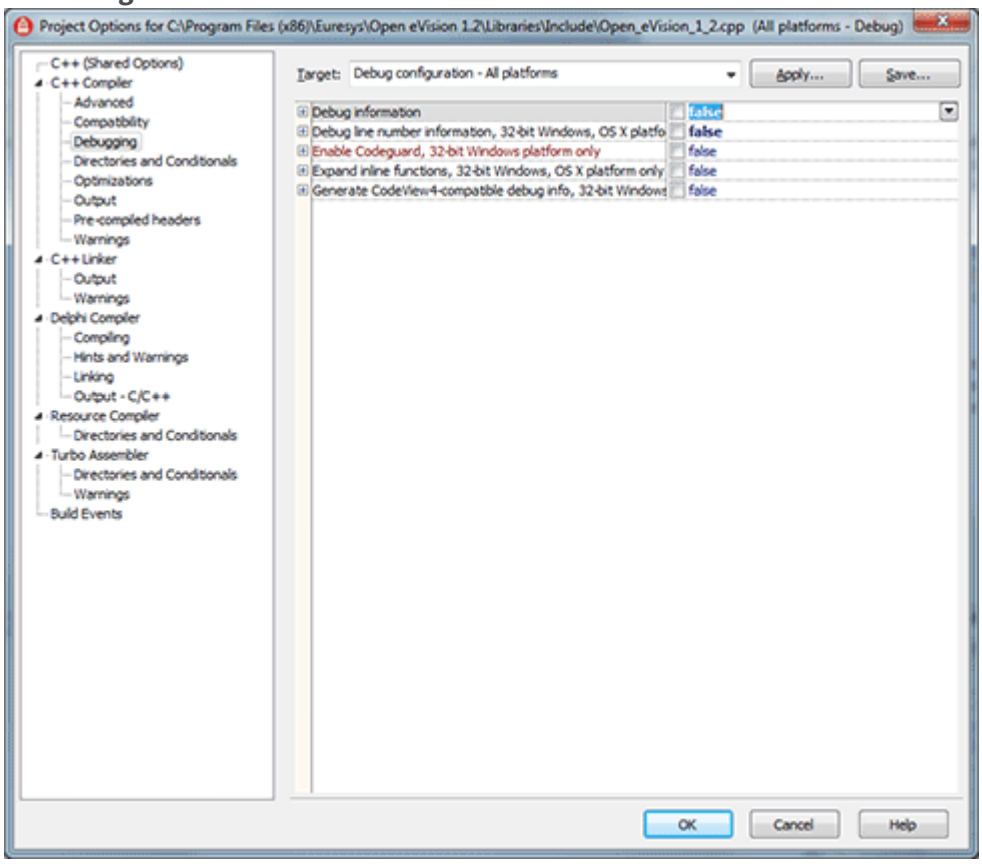
4. Include "`Open_evVsion_2_0.h`" in the .cpp files where you want to use Open eVision.
5. Add "`Open_evVsion_2_0.cpp`" to the project.



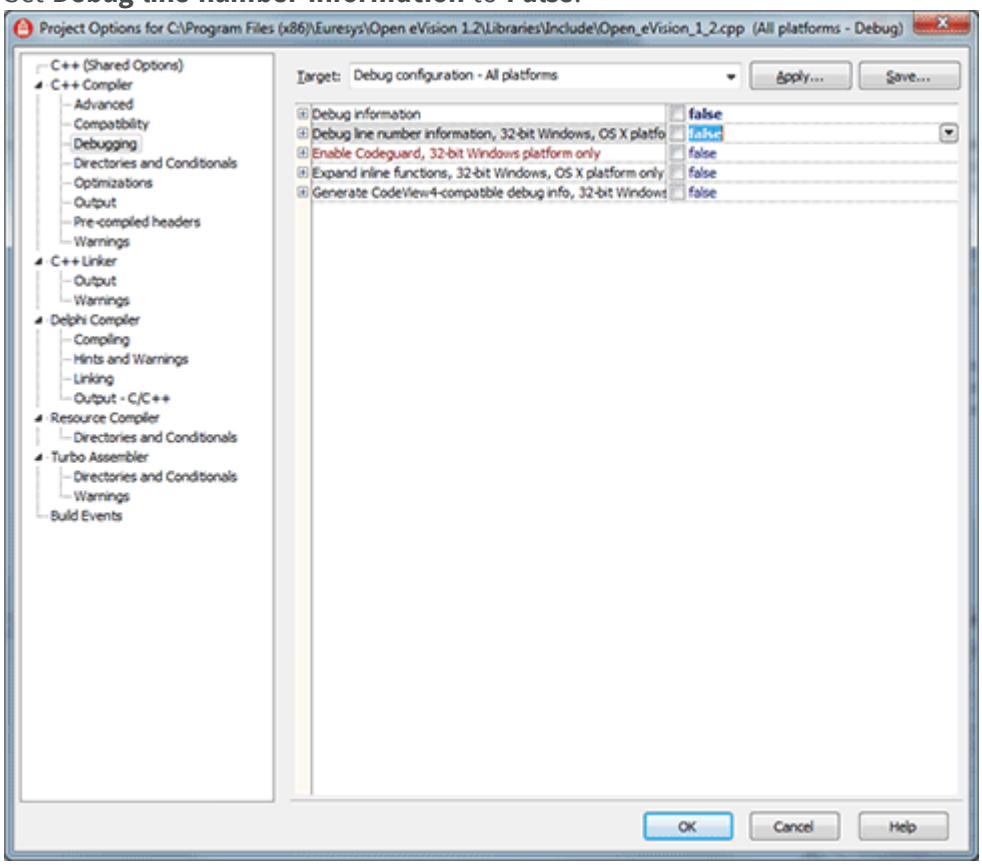
6. In both Debug and Release, modify the local C++ compiler options of the "Open_evVsion_2_0.cpp" file:



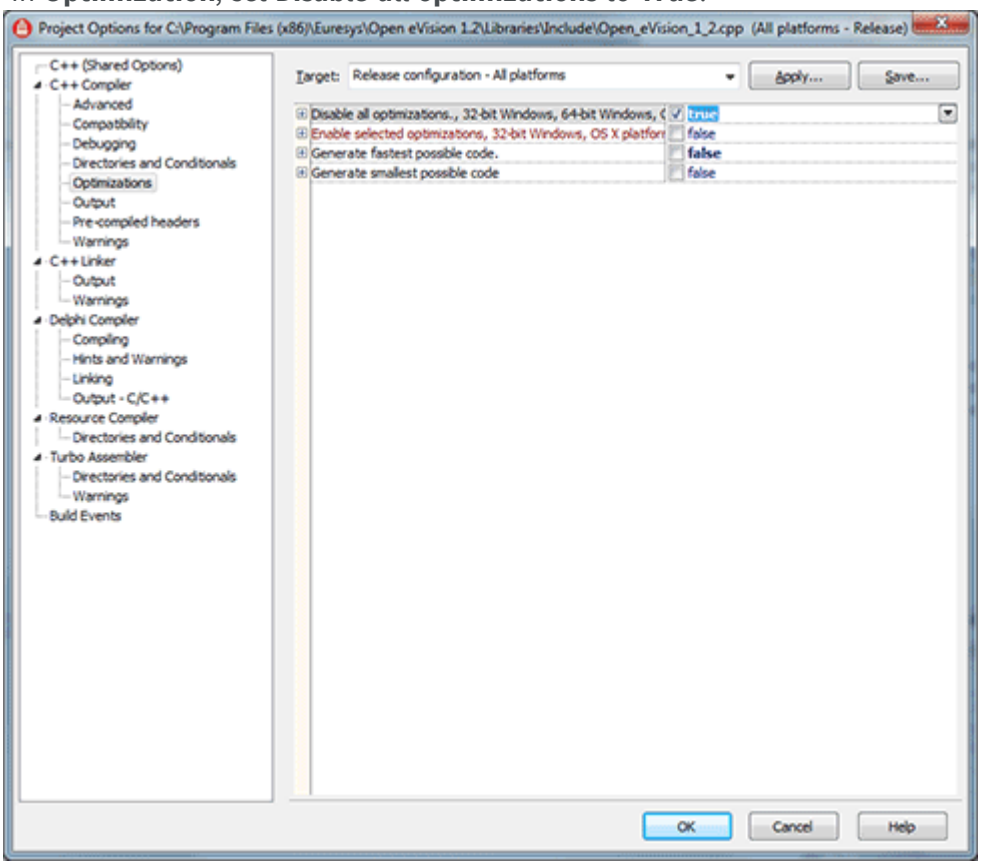
7. In **Debugging**:
   a. Set **Debug information** to **False**.



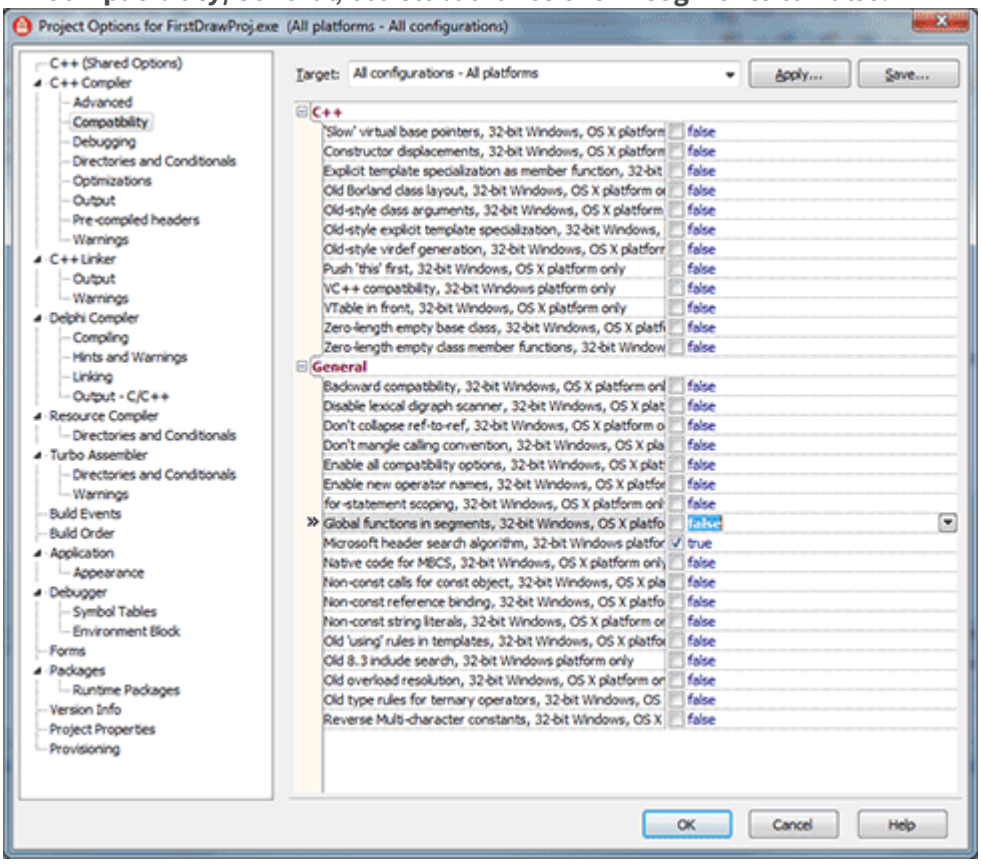
   b. Set **Debug line number information** to **False**.



8. In **Optimization**, set **Disable all optimizations** to **True**.



9. In **Compatibility/General**, set **Global functions in segments** to **False**.

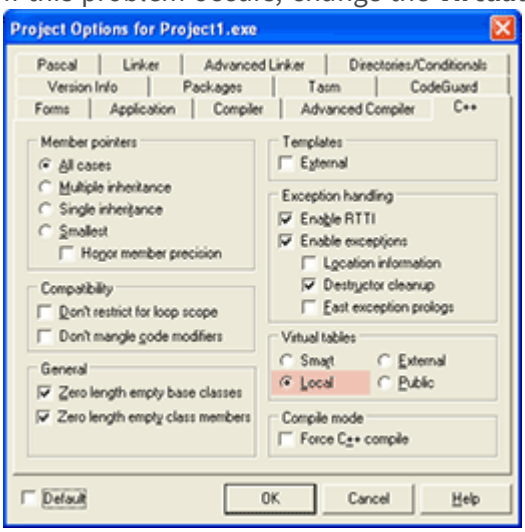


BORLAND C++

An error may occur due to a limitation in the number of functions (and virtual functions) in a single translation unit:

```
BCB6 Error E2491: Maximum VIRDEF count exceeded; check for recursion
```

If this problem occurs, change the **Virtual tables** C++ option to **Local**:

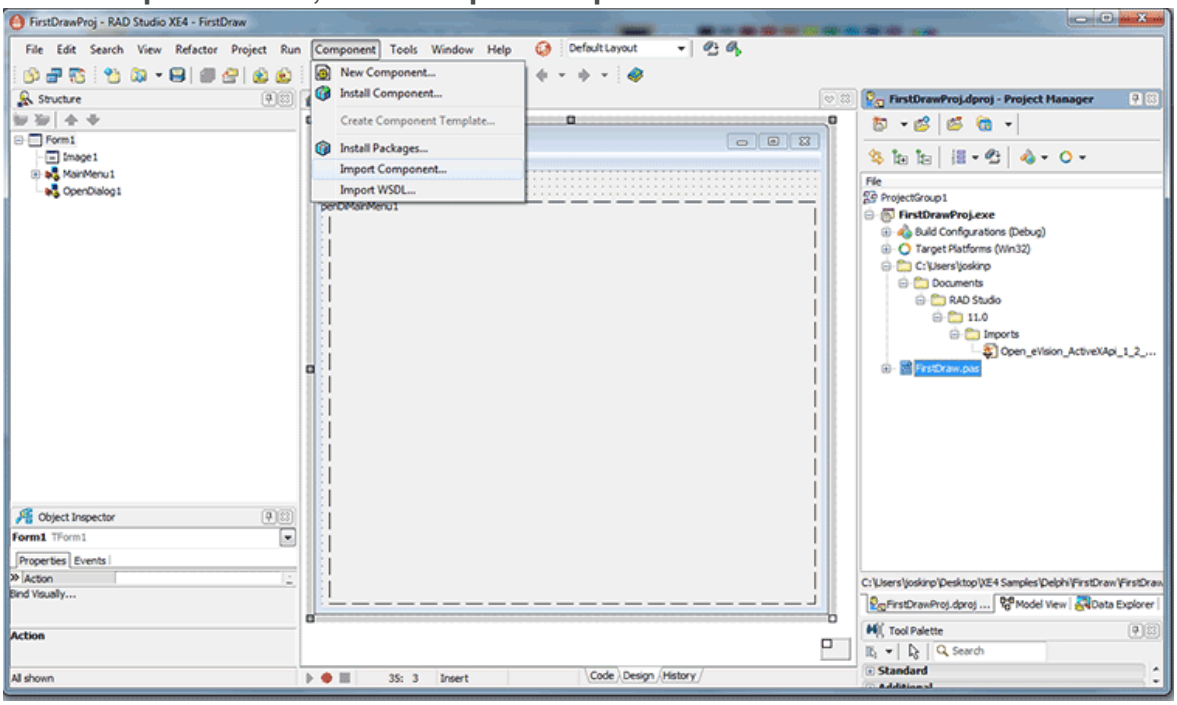


When using Open eVision objects as members of Borland GUI classes, like `TDialog` for instance, an unhandled exception can occur when the application is closed. To avoid this issue, create Open eVision objects dynamically using `new` and delete them in the destructor of the parent class.

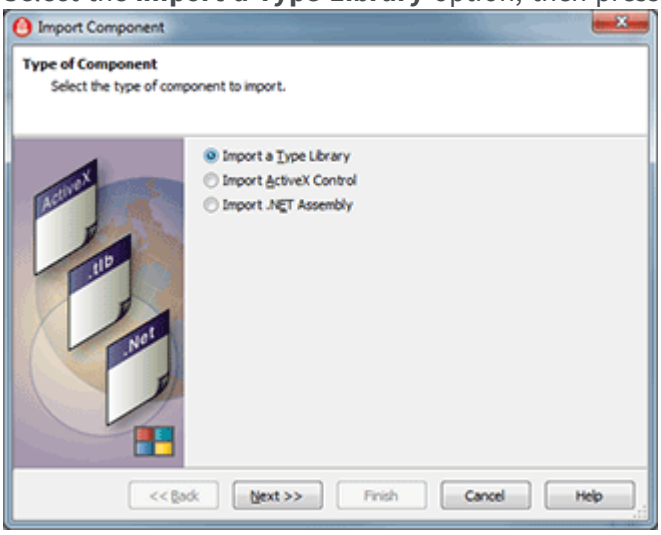
DELPHI

Open eVision must be previously installed with the **Legacy ActoveX** option.
You must use the ActiveX DLL ("Component > Import Component... > Import .NET Assembly").

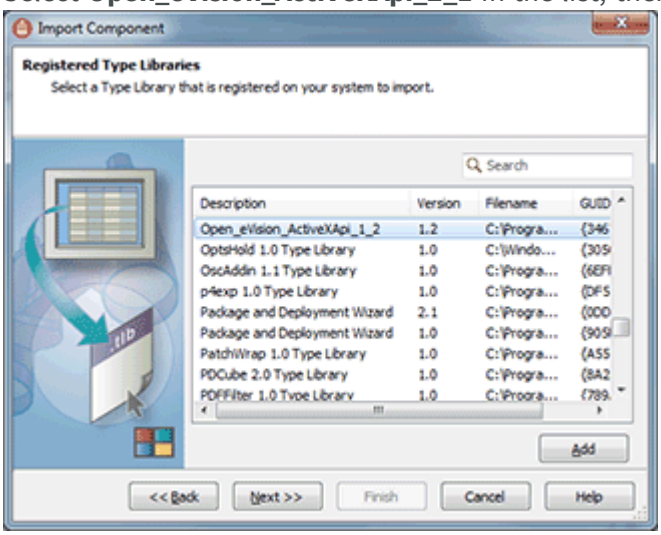
1. Create a new Delphi project.
2. In the **Component** menu, click on **Import Component**.



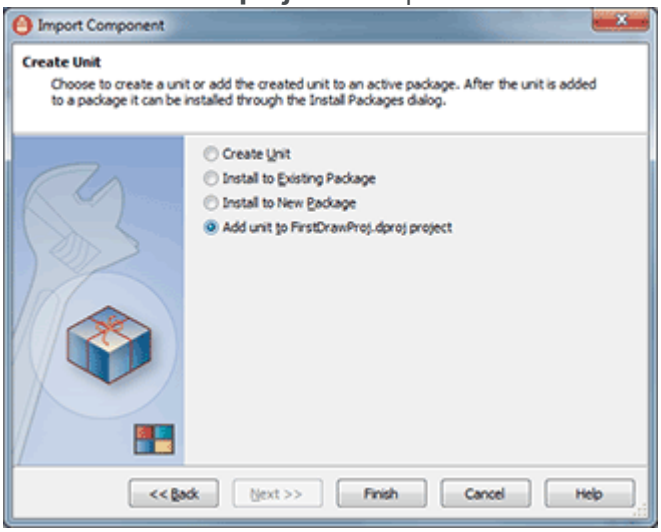
3. Select the **Import a Type Library** option, then press **Next**.



4. Select **Open_eVision_ActiveXApi_2_1** in the list, then press **Next**.



5. Don't change anything on the next form and press **Next**.
6. Select **Add unit to project** then press **Finish**.



Solving a Vision Problem

TYPICAL VISION APPLICATION

THE STEPS OF A TYPICAL VISION APPLICATION ARE:

- Image acquisition
- Pre-processing
- Location
- Analysis
- Diagnosis

Image Acquisition

Open eVision has no built-in image acquisition capabilities. It is, however, compatible with a wide variety of image acquisition solutions thanks to the capability of the EImage object to accept external pixel buffers.

Pre-processing

Pre-processing prepares the image to optimize the next steps of the application. It can reduce defects introduced by the acquisition (geometric distortion, de-focusing, noise) and enhance some desirable properties such as contrast between objects and background.

Most preprocessing features can be found in the EasyImage and EasyColor libraries.

Location

Many libraries, such as EasyOCR, EasyOCV, EasyBarCode, EasyQRCode or EasyMatrixCode are able to locate automatically their associated objects of interest.

Objects of interest of other types can be located using techniques such as segmentation (EasyObject) or pattern matching (EasyMatch, EasyFind).

When relevant, the same techniques can be used to count the number of objects of interest present in the image.

Analysis

Specialized libraries such as EasyOCR, EasyOCV, EasyBarCode, EasyQRCode or EasyMatrixCode will analyze and return metrics pertaining to their specific objects of interest.

Open eVision also provides other libraries to analyze the content of the image. EasyImage can provide pixel statistics, EasyObject can provide metrics on the objects it detected and EasyGauge precise measurements of objects in the image.

Diagnosis

Using the metrics computed during analysis, the application can diagnose the objects of interest by comparing them with good instances of the same objects.

PROTOTYPING

Open eVision Studio is a powerful prototyping tool that ships with Open eVision.

Using images stored on disk, Open eVision Studio allows to exert all the features of the Open eVision Libraries using simple dialog-based graphical user interfaces.

Open eVision Studio can be used to prototype most steps of a vision application, testing out the Open eVision tools and varying their parameters to find out the optimal values for solving an given vision problem.

Moreover, during the tests, Open eVision Studio generates source code that can be directly integrated into the application, reducing even further development time.

DEVELOP THREAD SAFE APPLICATIONS

Open eVision is conditionally thread-safe, meaning that different threads can access different Open eVision objects simultaneously, and access to shared data is protected from race conditions.

Moreover, when using static functions (or class methods), it is required for objects passed as parameters to be kept unmodified until the function ends. If the same Open eVision object is accessed by multiple threads, the code sections must be protected by a mutual exclusion mechanism.

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.

FOUNDATIONS

- Basic Types and Operations contains the definition of fundamental objects , types, classes and functions used in all Open eVision components.
- Easy3D contains a set of tools for solving computer vision problem using 3D acquisition and processing.

PREPROCESSING

- EasyImage contains **gray-level image** processing functions that improve image quality and contrast between background and objects of interest as well as linear and non-linear **filtering**,and **geometric transformations**.
- EasyColor contains **color image** processing functions that efficiently convert images between several color systems.

BLOB INSPECTION

- EasyObject obtains information about distinct **objects** (blob analysis), **identifies** them using connected component labeling, then **sorts** them and **selects** with them respect to their geometric features.
- EasyObject 2 contains advanced blob detection and analysis tools.

PATTERN MATCHING

- EasyMatch **locates patterns** in an image based on a pixel-by-pixel comparison with a reference pattern or template. It can be used for image registration or component placement inspection.
- EasyFind **locates patterns** in an image based on a geometrical model from a reference pattern or template. Compared to EasyMatch, it is computationally fast,robust against noise, occlusion, blurring and illumination variations.

2D MEASUREMENT

- EasyGauge **assesses dimensions** of objects to sub-pixel precision, detects edges, locates points and fits geometric models. EasyGauge can measure in physical units (mm, inch, ...) instead of pixels if the field of view is calibrated.

MARK-INSPECTION LIBRARIES

- EasyOCR performs **optical character recognition**. It can be used for reading serial numbers or printed labels.
- EasyOCR2 contains **advanced** optical character recognition functions.
- EasyOCV **checks the print quality of labels** against a template. EasyOCV can inspect the global image or independent shapes. It can detect issues with low contrast, misalignment, scratches or incorrect markings.
- Echecker **creates a Golden template and inspects images**. It uses EasyOCV library functions.
- EasyBarCode **reads bar codes**.
- EasyMatrixCode **reads Data Matrix codes**.
- EasyQRCode **detects and decodes QR codes**.

IMAGE STATISTICS

- EasyImage statistics contains tools for quantifying image focus, sliding window statistics and histogram analysis.

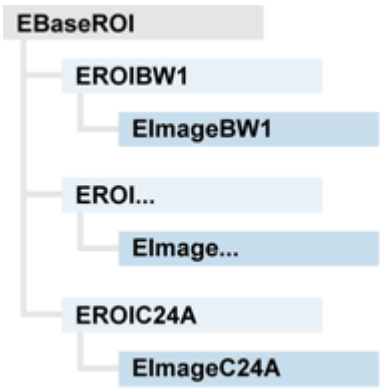
Basic Types and Operations

These Open eVision basic types are used in all Open eVision components:

- Image Definition and Parameters
- "Image Types" on the next page
- "Supported Image File Types" on page 23
- "Image File Access - Save, Load" on page 23
- "Image Pixels " on page 26
- "Image Buffer" on page 27
- "Image Construction Memory Allocation" on page 27
- "Image Drawing and Overlay" on page 30
- "Vector Types and Main Properties" on page 31
- ROIs and masks: Restrict processing to part of an image
- Color types

CLASSES

Image and ROI classes derive from abstract class `EBaseROI` and inherit all its properties.



Images

Open eVision image objects contain image data that represents rectangular images.

Each image object has a data buffer, accessible via a pointer, where pixel values are stored contiguously, row by row.

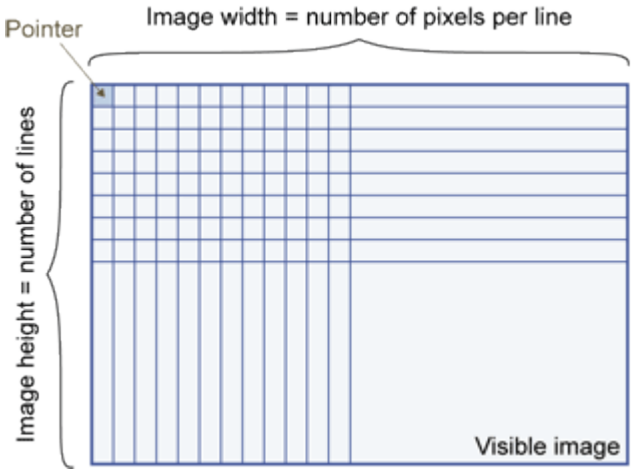


IMAGE MAIN PARAMETERS

An Open eVision image object has a rectangular array of pixels characterized by `EBaseROI` parameters .

- `Width` is the number of columns (pixels) per row of the image.
- `Height` is the number of rows of the image. (Maximum width / 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.)
- `Size` is the width and height.

The `Plane` parameter contains the number of color components. Gray-level images = 1. Color images = 3.

Image Types

Several image types are supported according to their pixel types: black and white, gray levels, color, etc.

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

BW1	1-bit black and white images (8 pixels are stored in 1 byte)	`EImageBW1`
BW8	8-bit grayscale images (each pixel is stored in 1 byte)	`EImageBW8`
BW16	16-bit grayscale images (each pixel is stored in 2 bytes)	`EImageBW16`
BW32	32-bit grayscale images (each pixel is stored in 4 bytes)	`EImageBW32`
C15	15-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images and MultiCam RGB15 format.	`EImageC15`
C16	16-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB16 color images and MultiCam RGB16 format.	`EImageC16`

C24	C24 images store 24-bit color images (each pixel is stored in 3 bytes). Compatible with Microsoft® Windows RGB24 color images and MultiCam RGB24 format.	`EImageC24`
C24A	C24A images store 32-bit color images (each pixel is stored in 4 bytes). Compatible with Microsoft® Windows RGB32 color images and MultiCam RGB32 format.	`EImageC24A`

Supported Image File Types

Type	Description
BMP	Uncompressed image data format (Windows Bitmap Format)
JPEG	Lossy data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 10918-1. Compression irretrievably looses quality.
JFIF	JPEG File Interchange Format
JPEG-2000	Data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 15444-1 and ISO/IEC 15444-2. Open eVision supports only lossy compression format, file format and code stream variants. - **code stream** describes the image samples. - **file format** includes meta-information such as image resolution and color space.
PNG	Lossless data compression method (Portable Network Graphics).
Serialized	Euresys proprietary image file format obtained from the **serialization** of Open eVision image objects.
TIFF	Tag Image File Format is currently controlled by Adobe Systems and uses the LibTIFF third-party library to process images written for 5.0 or 6.0 TIFF specification. File **save** operations are lossless and use CCITT 1D compression for 1-bit binary pixel types and LZW compression for all others. File **load** operations support all TIFF variants listed in the LibTIFF specification.

Image File Access - Save, Load

SAVE

The `Save` method saves the image data of an image object into a file using two arguments:

- Path: path, filename, and file name extension.
- Image File Type. If omitted, the file name extension is used.

Images bigger than 65,536 (either width or height) must be saved in Open eVision proprietary format.

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

IMAGE FILE TYPE ARGUMENTS

Argument	Image File Type
EImageFileType_Auto(*)	Automatically determined by the filename extension. See below.
EImageFileType_Euresys	Open eVision Serialization.
EImageFileType_Bmp	Windows bitmap - BMP
EImageFileType_Jpeg	JPEG File Interchange Format - JFIF
EImageFileType_Jpeg2000	JPEG 2000 File format/Code Stream -JPEG2000
EImageFileType_Png	Portable Network Graphics - PNG
EImageFileType_Tiff	Tagged Image File Format - TIFF

(*) Default value.

ASSIGNED IMAGE FILE TYPE IF 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.

SAVING JPEG AND JPEG2000 LOSSY COMPRESSIONS

`SaveJpeg` and `SaveJpeg2K` specify the compression quality when saving compressed images. They have two arguments:

- Path: a string of characters including the path, filename, and file name extension.
- Compression quality of the image file, an integer value in range [0: 100].
  `SaveJpeg` saves image data using JPEG File Interchange Format – JFIF.
  `SaveJpeg2K` saves image data using JPEG 2000 File format.

JPEG COMPRESSION VALUES

JPEG compression	**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).

REPRESENTATIVE JPEG 2000 COMPRESSION QUALITY VALUES

JPEG 2000 compression	**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).

LOAD

The `Load` method loads image data into an image object.
It has one argument: the **path:** path, filename, and file name extension.
File type is determined by the file format.
The destination image is automatically resized according to the size of the image on disk.

The `Load` method throws an exception when:

- File type identification fails
- File type is incompatible with pixel type of the image object

Note. Serialized image files of Open eVision 1.1 and newer are incompatible with serialized image files of previous Open eVision versions.

Image Pixels

PIXEL ACCESS

The recommended method to access pixels is to use `SetImagePtr` and `GetImagePtr` to embed the image buffer access in your own code. See also Image Construction and Memory Allocation and Retrieving Pixel Values.

Use of the following methods should be limited because of the overhead incurred by each function call:

DIRECT ACCESS

`EROIBW8::.GetPixel` and `SetPixel` methods are implemented in all image and ROI classes to read and write a pixel value at given coordinates. To scan all pixels of an image, you could run a double loop on the X and Y coordinates and use `GetPixel` or `SetPixel` each iteration, but this is not recommended.

C++ QUICK ACCESS TO BW8 PIXELS

In BW8 images, a call to `EBW8PixelAccessor::.GetPixel` or `SetPixel` will be faster than a direct `EROIBW8::.GetPixel` or `SetPixel`.

SUPPORTED STRUCTURES

- `EBW1`, `EBW8`, `EBW32`
- `EC15` (*), `EC16` (*), `EC24` (*)
- `EC24A`

(*) These formats support RGB15 (5-5-5 bit packing), RGB16 (5-6-5 bit packing) and RGB32 (RGB + alpha channel) but they must be converted to/from EC24 using `EasyImage::.Convert` before any processing.

Transition with versions prior to eVision 6.5 should be seamless: image pixel types were defined using typedef of integral types, pixel values were treated as unsigned numbers and implicit conversion to/from previous types is provided.

PIXEL AND FILE TYPE COMPATIBILITY DURING LOAD OR SAVE OPERATIONS

Type	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: Not supported. An exception occurs if you use the combination.

Ok: Image integrity is preserved with no data loss (apart from JPEG and JPEG2000, lossy compression).

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

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

Image Construction Memory Allocation

An image can be constructed with an internal or external memory allocation.

INTERNAL MEMORY ALLOCATION

The image object dynamically allocates and unallocates a buffer. Memory management is transparent.
When the image size changes, re-allocation occurs.
When an image object is destroyed, the buffer is unallocated.
To declare an image with internal memory allocation:

1. Construct an image object, for instance `EImageBW8`, either with width and height arguments, OR using the `SetSize` function.
2. Access a given pixel. There are several functions that do this. `GetImagePtr` returns a pointer to the first byte of the pixel at given coordinates.

EXTERNAL MEMORY ALLOCATION

The user controls buffer allocation, or links a third-party image in the memory buffer to an Open eVision image.
Image size and buffer address must be specified.
When an image object is destroyed, the buffer is unaffected.
To declare an image with external memory allocation:

1. Declare an image object, for instance `EImageBW8`.
2. Create a suitably sized and aligned buffer (see Image Buffer).
3. Set the image size with the `SetSize` function.
4. Access the buffer with `GetImagePtr`. See also Retrieving Pixel Values.

Image Buffer

Image pixels are stored contiguously, from top row to bottom, from left to right, in Windows bitmap format (top-down DIB[1]) into an associated buffer.
The buffer address is a pointer to the start address of the buffer, which contains the top left pixel of the image.

[1]device-independent bitmap

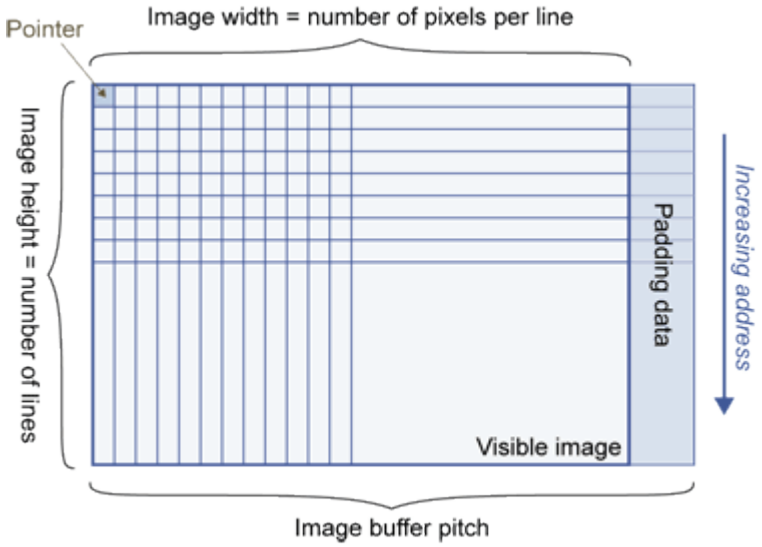
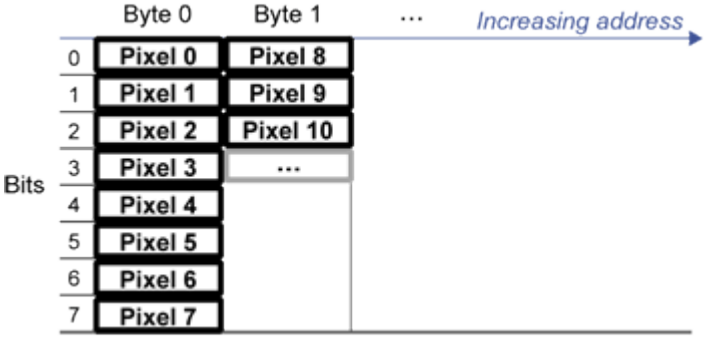


IMAGE BUFFER PITCH

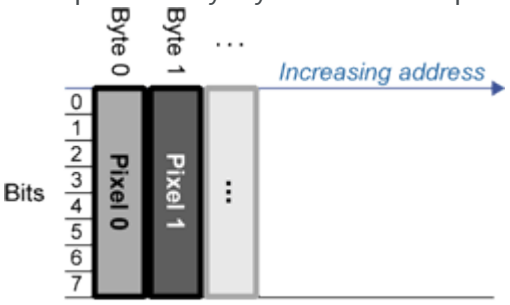
Alignment must be a multiple of 4 bytes.
Open eVision 1.2 onwards default pitch is 32 bytes for performance reasons (Open eVision 1.1.5 was 8 bytes).

MEMORY LAYOUT

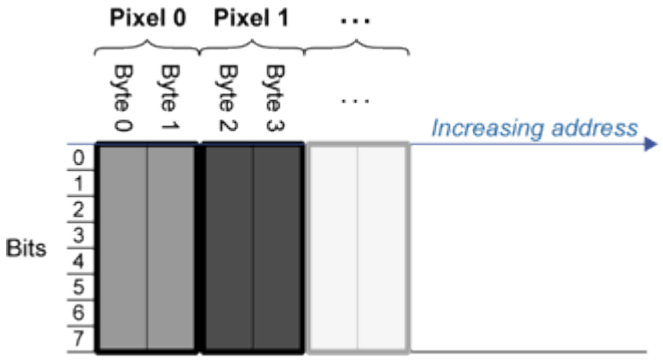
- `EImageBW1` stores 8 pixels in one byte.
  Example memory layout of the first 2 pixels of a BW1 image buffer:



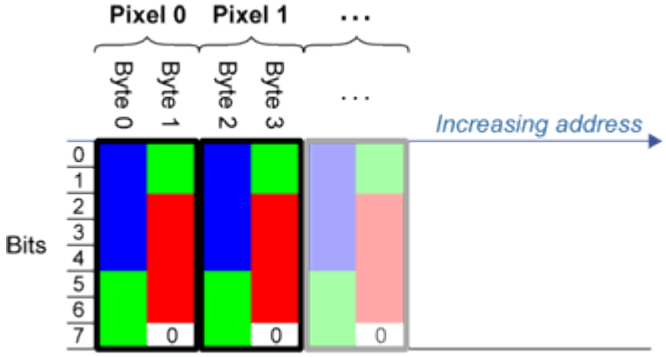
- `EImageBW8`stores each pixel in one byte.
  Example memory layout of the first pixels of a BW8 image buffer:



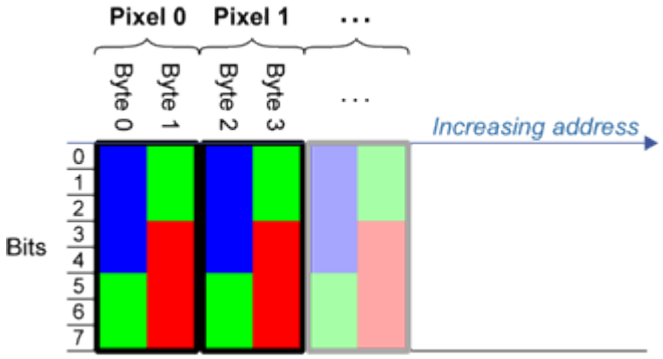
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).
  Example memory layout of the first pixels of a BW16 image buffer:



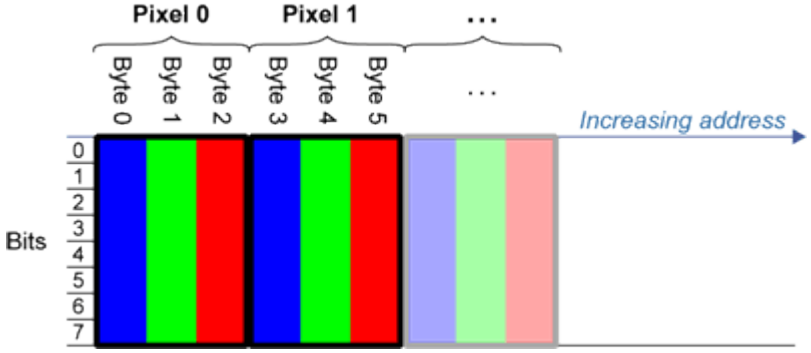
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits.
  The 16th bit is left unused.
  Example memory layout of the first pixels of a C15 image buffer:



- `EImageC16` stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
  The second color component is coded with 6-bits.
  Example memory layout of the first pixels of a C16 image buffer:



- `EImageC24` stores each pixel in 3 bytes. Each color component is coded with 8-bits.
  Example memory layout of the first pixels of a C24 image buffer:



- `EImageC24A` stores each pixel in 4 bytes. Each color component is coded with 8-bits. The alpha channel is also coded with 8-bits.
  Example memory layout of the first pixels of a C24A image buffer:

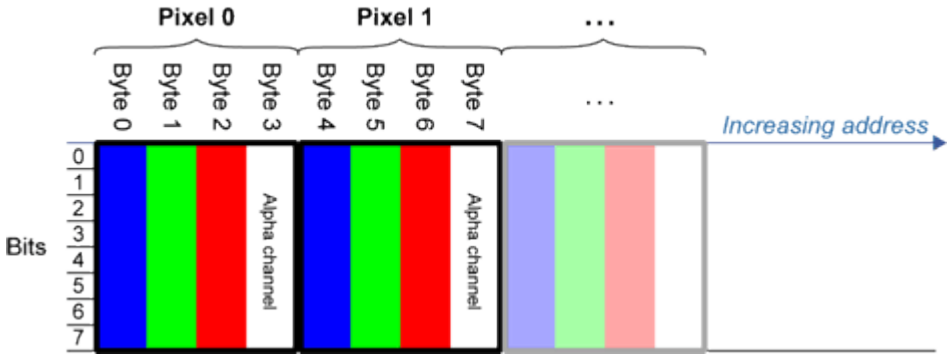


Image Drawing and Overlay

- Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.
- The color palette in 256-color display mode gives optimal rendering. Gray-level images can be improved using LUT[3]s (using histogram stretching techniques or pseudo-coloring).
- The zoom can be different horizontally and vertically.
- `DrawFrameWithCurrentPen` method draws a frame.
- **Non-destructive overlaying** drawing operations do not alter the image contents, such as `MoveTo`/`LineTo`.
- **Destructive overlaying** drawing operations alter the image contents by drawing inside the image such as `Easy::.OpenImageGraphicContext`. Gray-level [color] images can only receive a gray-level [color] overlay.

3D Rendering

These images are viewed by rotating them around the X-axis, then the Y-axis.

GRAY 3D RENDERING

`Easy::.Render3D` prepares a 3-dimensional rendering where gray-level values are altitudes. Magnification factors in the three directions (X = width, Y = height and Z = depth) can be given. The rendered image appears as independent dots whose size can be adjusted to make the

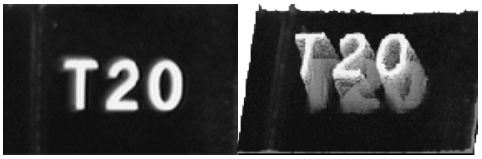
---

[1]Graphics Device Interface

[2]Microsoft Foundation Class

[3]LookUp Tables

surface more or less opaque.

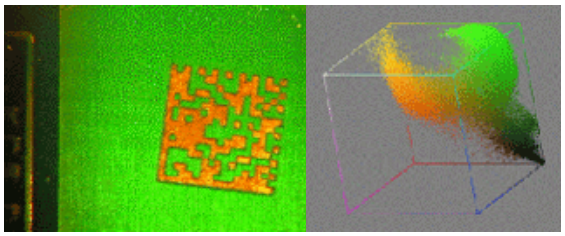


3D rendering

COLOR HISTOGRAM 3D RENDERING

`Easy::.RenderColorHistogram` prepares a 3-dimensional rendering of a color image histogram.
The pixels are drawn in the RGB space (not XY-plane) to show clustering and dispersion of RGB values.
This function can process pixels in other color systems (using EasyColor to convert), but the raw RGB image is required to display the pixels in their usual colors.

Magnification factors in all three directions (X = red, Y = green and Z = blue) can be given.

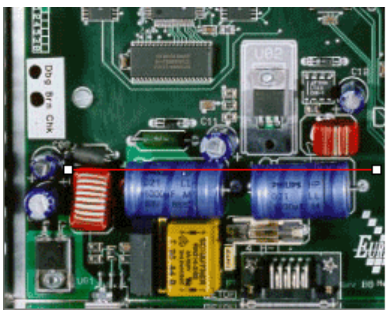


Color histogram rendering

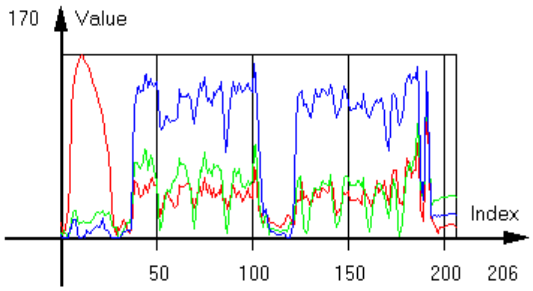
Vector Types and Main Properties

A vector is a one-dimensional array of pixels (taken from an image profile or contour).

`EVector` is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.



Profile in a C24 image



RGB values plot along profile

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

RGB values array (`EC24Vector`)

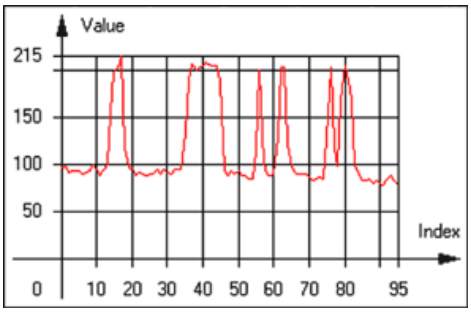
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

1. **Create a vector of the appropriate type**, using its constructor and pre-allocate elements if required.

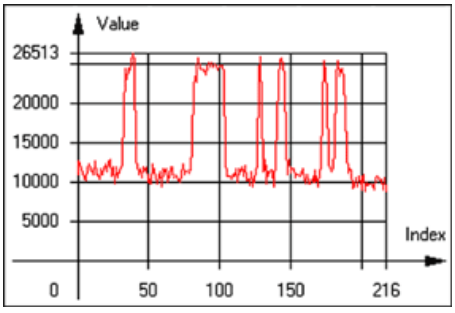
VECTOR TYPES

- `EBW8Vector`: 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`, ...).



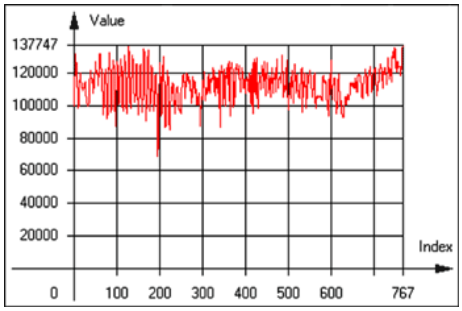
Graphical representation of an `EBW8Vector` (see `Draw` method)

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



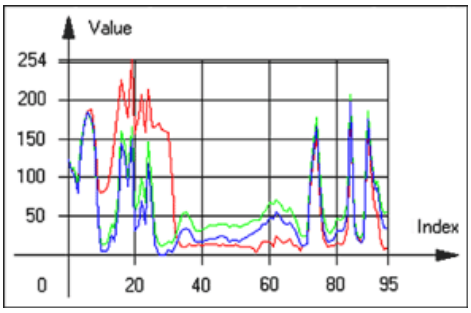
Graphical representation of an `EBW16Vector`

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



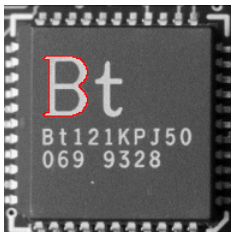
Graphical representation of an `EBW32Vector`

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



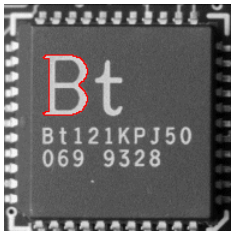
Graphical representation of an `EC24Vector`

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



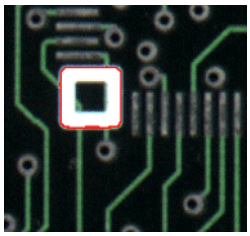
Graphical representation of an `EBW8PathVector` (see `Draw` method)

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



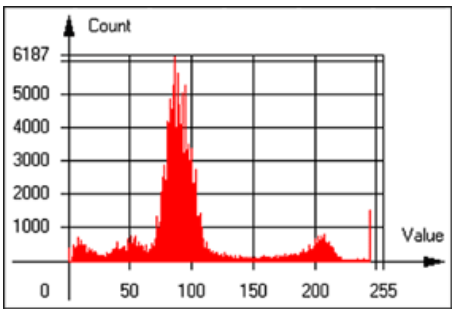
Graphical representation of an `EBW16PathVector` (see `Draw` method)

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



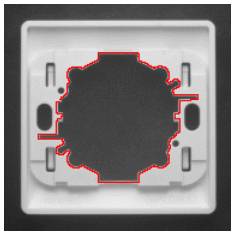
Graphical representation of an `EC24PathVector` (see `Draw` method)

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

2. **Fill a vector with values**. First empty it, using the `EVector::.Empty` member, then add elements one at a time by calling the `EC24Vector::.AddElement` member. You can access any element by means of indexing.
3. **Access a vector element**, either for reading or writing. Use the brackets operator, for instance, `EC24Vector::.operator[]`.
4. **Determine the current number of elements**, use member `EVector::.NumElements`.
5. **Draw the vector**.
   A pixel vector is a plot of the element values as a function of the element index, so its graphical appearance depends on its type. You can draw a vector in a window. For legibility, the drawing should appear on a neutral background.
   Drawing is done in the device context associated to the desired window. By default, curves are drawn in blue, annotations are drawn in black. The following parameters can be defined: graphicContext, width, height, origin, origin, color0, color1, color2.
   The `EC24Vector` has three curves drawn instead of one, each corresponding to a color component. By default, red, blue and green pens are used.

ROI Main Properties

ROIs are defined by a width, a height, and **origin x and y coordinates**.
The origins are specified with respect to the top left corner in the parent image or ROI.
The ROI must be wholly contained in its parent image.
The processing/analysis time of a BW1 ROI is faster if `OrgX` and `Width` are multiples of 8.

SAVE AND LOAD

You can save or load an ROI as a separate image, to be used as if it was a full image. The ROIs perform **no memory allocation** at all and never duplicate parts of their parent image, the parent image provides them with access to its image data.

The image size of the new file must match the size of the ROI being loaded into it. The image around the ROI remains unchanged.

ROI CLASSES

An Open eVision ROI inherits parameters from the abstract class `EBaseROI`.

There are several ROI types, according to their pixel type. They have the same characteristics as the corresponding image types.

- `EROIBW1`
- `EROIBW8`
- `EROIBW16`
- `EROIBW32`
- `EROIC15`
- `EROIC16`
- `EROIC24`
- `EROIC24A`

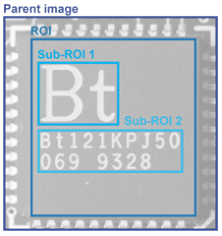
ATTACHMENT

An ROI must be `attached` to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.
A normal image cannot be attached to another image or ROI.

NESTING

Set and Get functions change or query the width, height and position of the origin of an ROI, with respect to its immediate or topmost parent image.

An image may accommodate an arbitrary number of ROIs, which can be nested in a hierarchical way. Moving the ROI also moves the embedded ROIs accordingly. The image/ROI classes provide several methods to traverse the hierarchy of ROIs associated with an image.



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

CROPPING

`CropToImage` crops an ROI which is partially out of its image. The resized ROI never grows. An exception is thrown if a function attempts to use an ROI that has limits that extend outside of the parents.

(In Open eVision 1.0.1 and earlier, an ROI was silently resized or repositioned when placed out of its image and sometimes grew. If ROI limits extended outside parents, they were silently resized to remain within parent limits.)

RESIZING AND MOVING

- ROIs can easily be resized and positioned by two functions and dragging handles:
  - `EBaseROI::.Drag` adjusts the ROI coordinates while the cursor moves.
  - `EBaseROI::.HitTest` informs if the cursor is placed over a dragging handle. Once the handle is known, the cursor shape can be changed by an `OnSetCursor` MFC event handler. HitTest is unpredictable if called while dragging is in progress.
    HitTest can be used in an OnSetCursor MFC event handler to change the cursor shape, or before a dragging operation like OnLButtonDown,
    (or EvSetCursor and EvLButtonDown in Borland/CodeGear's OWL)
    (or FormMouseMove and FormMouseDown in Borland/CodeGear's VCL).
    In VB6, MouseDown, MouseMove,MouseUp events return the current cursor position in twips rather than pixels, so conversion is mandatory.

ROIs and masks

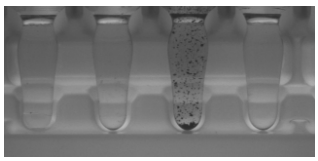
ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 35 apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
- "Flexible Masks " below are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some EasyObject and EasyImage library functions.

Flexible Masks

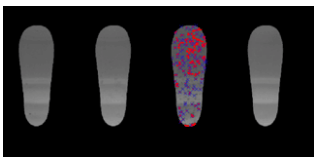
A flexible mask is a BW8 image with the same height and width as the source image. It contains shapes of areas that must be processed and ignored areas (that will not be considered during processing):

- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image

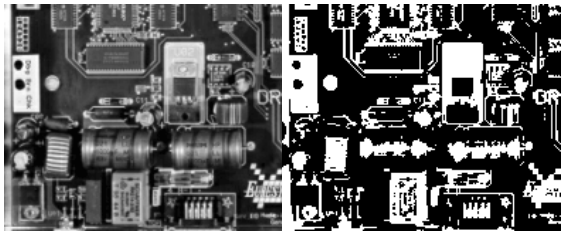
Associated mask



Processed masked image

A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.
See Flexible Masks in EasyImage and Flexible Masks in EasyObject for detailed information.

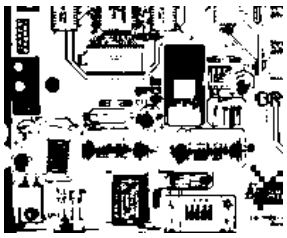
Flexible Masks in EasyImage



Source image (left) and mask variable (right)

SIMPLE STEPS TO USE FLEXIBLE MASKS IN EASYIMAGE

1. **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.
2. **Display the results.**



Resulting image

EASYIMAGE FUNCTIONS THAT SUPPORT FLEXIBLE MASKS

- `EImageEncoder::.Encode` has a flexible mask argument for BW1, BW8, BW16, and C24 source images.
- AutoThreshold.
- Histogram (function `HistogramThreshold` has no overload with mask argument).
- RmsNoise, SignalNoiseRatio.
- Overlay (no overload with mask argument for BW8 source images).
- ProjectOnAColumn, ProjectOnARow (Vector projection).
- ImageToLineSegment, ImageToPath (Vector profile).

Flexible Masks in EasyObject

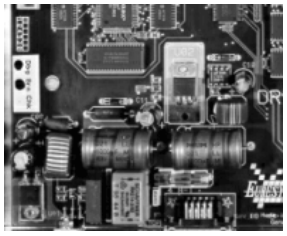
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

If an object of interest has the same gray level as other regions of the image, you can define "keep" and "ignore" areas using flexible masks and `Encode` functions.

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

- A pixel value of 0 in the flexible mask masks the corresponding source image pixel so it doesn't appear in the encoded image.
- Any other pixel value in the flexible mask causes the pixel to be encoded.

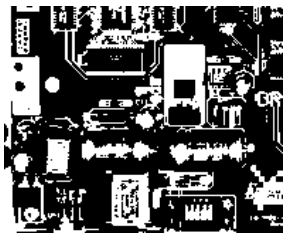
EASYOBJECT FUNCTIONS THAT CREATE FLEXIBLE MASKS



Source image

1) ECODEDIMAGE2::.RENDERMASK: FROM A LAYER OF AN ENCODED IMAGE

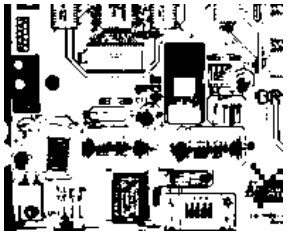
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers using minimum residue thresholding).
4. Make the mask image the desired size using `mask.SetSize(sourceImage.GetWidth(), sourceImage.GetHeight())`.
5. Exploit the flexible mask as an argument to `ECodedImage2::.RenderMask`.



BW8 resulting image that can be used as a flexible mask

2) ECODEDELEMENT::.RENDERMASK: FROM A BLOB OR HOLE

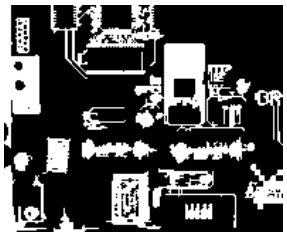
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::.RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

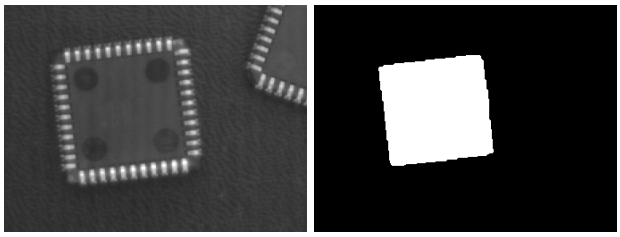
3) EOBJECTSELECTION::.RENDERMASK: FROM A SELECTION OF BLOBS

`EObjectSelection::.RenderMask` can, for example, discard small objects resulting from noise.



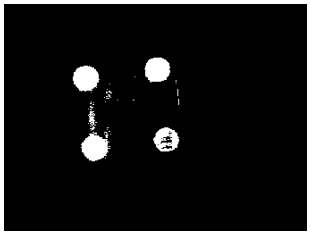
BW8 resulting image that can be used as a flexible mask

EXAMPLE: RESTRICT THE AREAS ENCODED BY EASYOBJECT



Find four circles (left) Flexible mask can isolate the central chip (right)

1. Declare a new `ECodedImage2` object.
2. Setup variables: first declare source image and flexible mask, then load them.
3. Declare an `EImageEncoder` object and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
4. Encode the source image. Encoding a layer with just the area in 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:



5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

Profile

PROFILE SAMPLING

A **profile** is a series of pixel values sampled along a line/path/contour in an image.

- `EasyImage::.ImageToLineSegment` copies the pixel values along a given line segment (arbitrarily oriented and wholly contained within the image) to a vector. The vector length is adjusted automatically. This function supports flexible mask.
- A path is a series of pixel coordinates stored in a vector.
  `EasyImage::.ImageToPath` copies the corresponding pixel values to the vector. This 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 values inside a profile vector.

PROFILE ANALYSIS

The profile can be processed to find peaks or transitions:

- 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 (which transforms transitions (edges) into peaks) and looking for peaks in it.
  `EasyImage::.ProfileDerivative` computes the first derivative of a profile extracted from a gray-level image.
  The `EBW8` data type only handles unsigned values, so the derivative is shifted up by 128. Values under [above] 128 correspond to negative [positive] derivative (decreasing [increasing] slope).
- A peak is the portion of the signal that is above [or below] a given threshold - the maximum or minimum of the signal. This may correspond to the crossing of a white or black line or thin feature. It is defined by its:
  - Amplitude: difference between the threshold value and the max [or min] signal value.
  - Area: surface between the signal curve and the horizontal line at the given threshold.

`EasyImage::.GetProfilePeaks` detects max and min peaks in a gray-level profile. To eliminate false peaks due to noise, two selection criteria are used. The result is stored in a peaks vector.

PROFILE INSERTION INTO AN IMAGE

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

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

Color types

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.

Easy3D

Easy3D is a set of tools for solving computer vision problem using 3D acquisition and processing. Easy3D supports laser line triangulation for fast and precise acquisition of depth maps. Most of the Open eVision 2D image operators are compatible with depth maps.

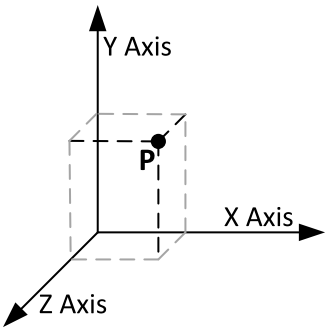
A simple calibration tool is provided to generate corrected, metric, point clouds. 3D operators are provided to work on point clouds and their exportation to the standard PCD file allows integration with other 3D tools.

This document introduces some basic concepts of Easy3D, like depth maps and point clouds, and exposes the typical 3D processing workflow.

Introduction to 3D

Basic Concepts

Open eVision uses a right-handed Cartesian 3D coordinate system. In this system, each 3D point is represented by its 3 coordinates X, Y and Z.



Right-handed means that if you were looking at the plane formed by the X and Y axis, Y up and X right, the Z axis would point in your direction.

Open eVision provides two main ways to represent an object in 3D: The Point Cloud and the Depth Map.

DEPTH MAP

A depth map is way to represent a 3D object using a 2D grayscale image, each pixel in the image representing a 3D point.

The pixel coordinates are the representation of the X and Y coordinates of the point while the grayscale value of the pixel is a representation of the Z coordinate of the point.

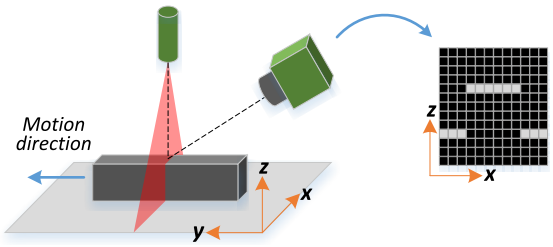
POINT CLOUD

A point cloud is a set of 3D points representing the surface of an object.

3D point clouds are produced by various 3D scanning techniques, such as Laser Triangulation, Time of Flight or Structured Lighting.

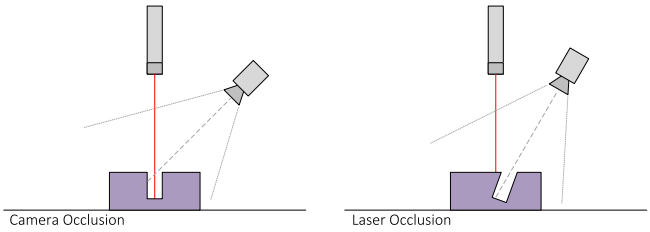
Laser Triangulation

In a laser-line triangulation system, a laser line is projected on a 3D object. A camera, placed in another perspective than the laser, is then used to capture an image of that line, deformed by the shape of the object.



The deformations of the line are a direct representation of the shape of the 3D object in the plane of the laser line. By scanning the object, that is making it move under the laser line and taking multiple images, you can reconstruct its 3D shape.

Using the laser triangulation method, there are parts of the object that will not be scanned. Those are the parts that the laser can't reach or places where the camera cannot see the laser line because of the object geometry. This is called occlusion and can be limited using advanced scanning methods like using two cameras or two lasers.



On the right figure, the laser can't reach the bottom of the hole, inducing laser occlusion. On the left one, the camera cannot see the bottom of the hole, inducing camera occlusion.

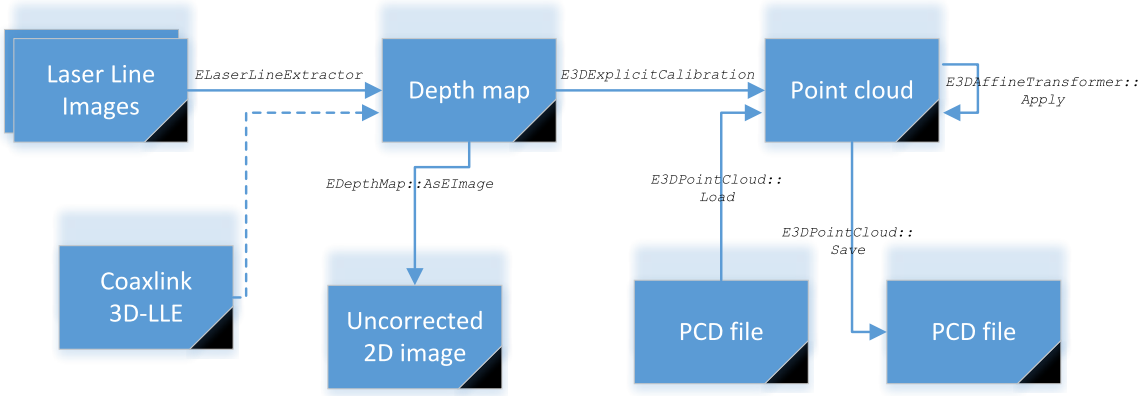
3D Processing Workflow

The illustration below shows the workflow of the 3D pipeline. From native images with visible laser lines, a depth map is constructed. Depth maps are also directly produced by Euresys' Coaxlink Quad 3D-LLE frame grabber.

A depth map can then directly be converted to an uncorrected EImage on which some 2D processing, like code reading or OCR, can be performed.

Alternatively, given a 3D calibration, a corrected point cloud can be calculated. Corrected point clouds use metric data and thus make measurements possible. Point clouds can be saved in and loaded from the standard PCD file format.

Display methods are available in Open eVision for Depth map and point cloud objects.

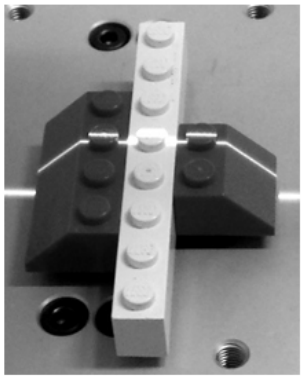


Laser Line extraction

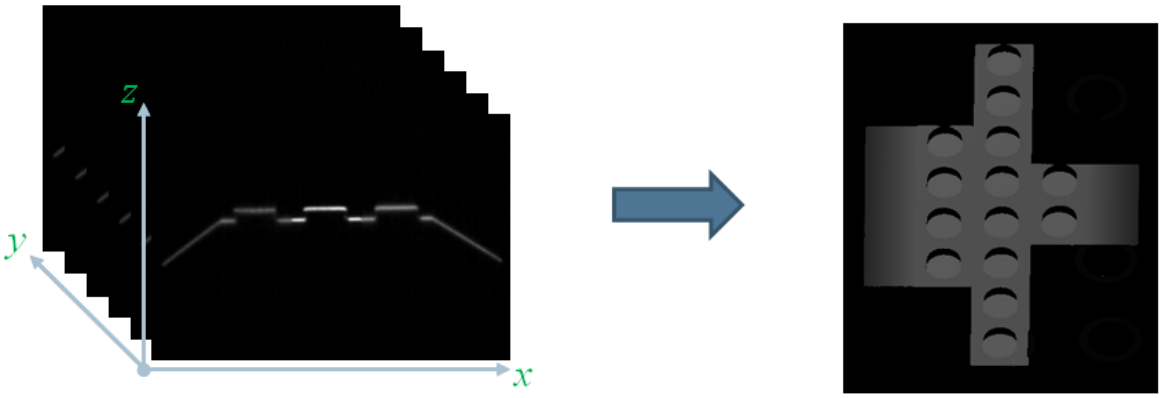
A Laser Line Extraction (LLE) algorithm is required to create a depth map from a sequence of profiles of the object captured by the camera sensor.

The objective of an LLE algorithm is to estimate the position where a laser line horizontally crosses a Region of Interest (ROI). The detection analyzes each column of a frame individually.

An LLE algorithm typically outputs a data array containing the vertical position of a detected laser line in a ROI, i.e., each computed ROI produces a single profile. The figure below illustrates depth map generation.



Measured object



Series of profiles

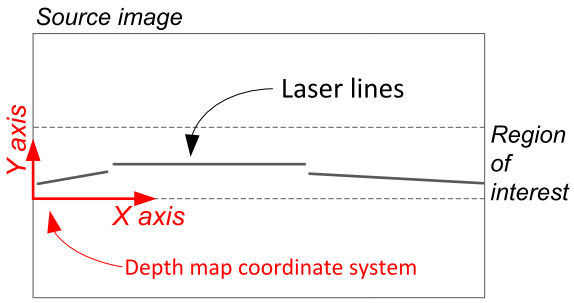
Heightmap

The `ELaserLineExtractor` class provides the laser line extraction functionality in Open eVision. It implements several algorithms to extract the laser line:

- Maximum detection returns the position of maximum intensity. It's the fastest method but don't supports sub-pixel precision.
- Peak detection approach detects local maxima. If several maxima are detected, the one with the highest intensity is returned. That method have a sub-pixel precision.
- Center of gravity algorithm is suitable when the laser line is spread over several pixels. The position is returned with sub-pixel precision.

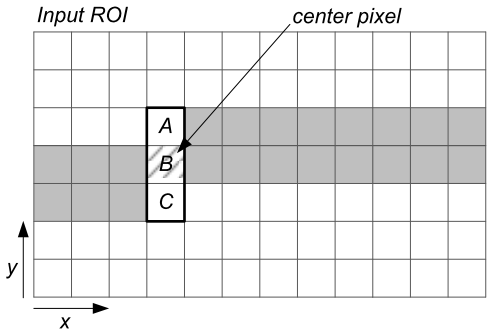
Optionally, a low pass linear filter can be applied before the line extraction in order to reduce noise and high frequencies in the image. A threshold value can exclude pixels with low intensity.

Values returned by the laser line extraction algorithms are relative to the bottom of the region of interest. Thus values in the depth map range from 0 (bottom on the ROI) to the height of the ROI, using the original image pixel scale.



LOW PASS LINEAR FILTER

The low pass filter applies a convolution operator on a 1x3 sliding window. The 3 elements of the convolution kernel (A, B, and C) are configurable, accepting any positive integer. The figure below illustrates the positioning of the convolution kernel elements within a given ROI.



The low pass filter can be activated for any of the laser line extraction methods with the method `ELaserLineExtractor::SetEnableSmoothing(true/false)`. Parameters A, B and C are set with `ELaserLineExtractor::SetSmoothingParameters(A, B, C)`.

Depth Map

A depth map is an image representing a displacement (Z coordinate) on each pixel. Each line of a depth map is composed of values produced by the laser line extraction (either the software implementation or the Coaxlink 3D LLE output). The pixel value of a depth map is thus the laser line vertical offset (relative to the ROI origin).

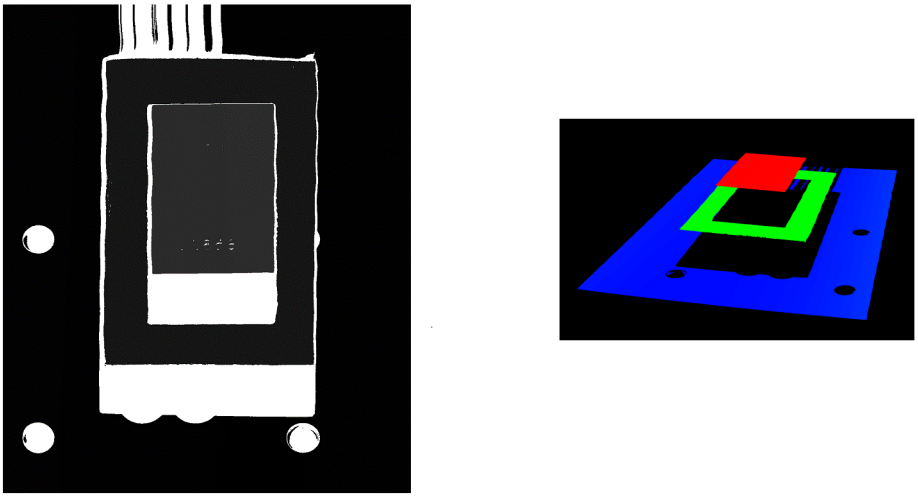
Depth maps are not optically corrected and thus suffer of optical distortions and perspective effect. Without calibration (see further), depth maps cannot be used for measurement or precise model matching.

FORMATS

Easy3D implements several pixel formats for the depth maps:

- `EDepthMap8` is a 8 bits per pixel depth map;
- `EDepthMap16` is a 16 bits per pixel, with variable fixed point encoding;
- `EDepthMap32f` uses single precision floating point encoding, suitable for sub pixel accuracy.

When the laser line position could not be detected by the extraction, a reserved value is used for the corresponding pixel. The `GetUndefinedValue()` method of the `EDepthMap` classes should be used to retrieve that value.



Left, a depth map with undefined values displayed in white. Right, the 3D point cloud showing "holes" in the object surface.

ACCESS

Depth maps values are stored in buffers compatible with the 2D Open eVision images.

The `AsImage()` method return a compatible image object (`EImageBW8`, `EImageBW16` or `EImageBW32f`) that can be used with Open eVision's 2D processing features.

`Accessors GetPixel(x,y)` and `SetPixel(value,x,y)` are also available to read and write individual pixel values.

For performance reasons, these accessors should not be used when a significant number of pixel needs to be processed. When that is the case, retrieving the internal buffer pointer using `GetBufferPtr()` and iterating on the pointer is recommended.

LOAD, SAVE AND DISPLAY

Depth maps support the methods `Save`, `Load` and `Draw` as documented for the Open eVision images.

Point Cloud

A point cloud (https://en.wikipedia.org/wiki/Point_cloud) is an unstructured set of 3D points representing discrete positions on the object surface.

Point clouds are computed from depth maps using a transformation defined by a calibration model. Calibration can be defined as the process of estimating the geometric attributes of the laser triangulation setup (see "Calibration" on the next page).

CONSTRUCTION

Given a depth map, a point cloud can be constructed by simply converting the each pixel (x, y, value) to an `E3DPoint (x, y, z)`. `E3DDepthMapToPointCloud` exposes such a conversion.

Easy3D also provides an explicit geometric calibration (as explained in the following chapter). The class `E3DExplicitGeometricCalibration` converts depth maps to point clouds given some geometric parameters.

ACCESS TO DATA

A point cloud is a vector of `E3DPoint`, accessible using the `AddPoint`/`GetPoint` methods.

When point access performance is an issue, it is also possible to fill the point cloud vector directly from an external buffer with the method `FillPointsBuffer` or to retrieve it using `GetPointBuffer`.

INPUT AND OUTPUT

Loading and saving a point cloud is possible using the `Save` and `Load` methods. These methods use a proprietary file format.

Easy3D also implements the standard PCD file format. Methods `SavePCD` and `LoadPCD` allows the point cloud to be stored in a file compatible with other software such as PCL (Point Cloud Library).

DISPLAY

To display a point cloud, a specific object, `E3DPointCloudViewer`, is provided in Open eVision.

This object allows you to display a point cloud efficiently in a provided window, which can be defined either as stand-alone or as a child of another window.

It also provides you with basic mouse-input capabilities such as rotation and scaling without the need of coding on you part.

Calibration

The calibration of a laser triangulation setup intends to establish the transformation between the laser line position in the camera image and real world coordinates.

Given a calibration, it is possible to reconstruct the surface of the 3D object from a laser triangulation scan in a metric coordinate system.

Calibration is thus mandatory for processing that require measurement or object matching with known models.

The calibration process must compute several parameters:

- The scale of the system.
- The perspective transformation
- Camera and optical distortions
- Relative position of the camera and the laser plane
- Relative motion of the camera and the object (motion direction and speed)

When these calibration parameters have been determined, a depth map (2.5D non metric data) can be converted to a point cloud (3D metric data).

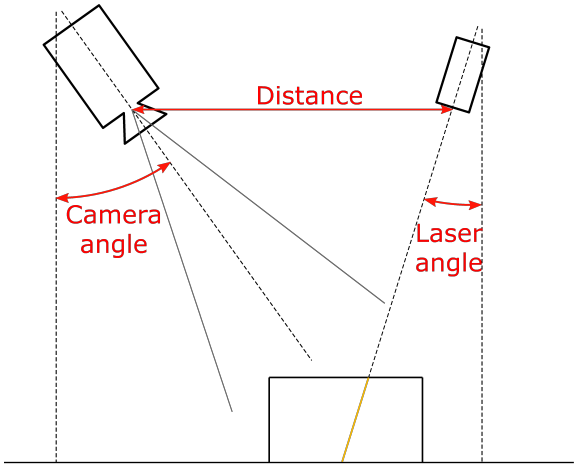
Several approaches can be used to calibrate a laser triangulation setup, but most of them rely on the observation of a reference object of known geometry (caliber). Comparison between the theoretical object geometry and the scanned object allows the computation of the calibration parameters listed above.

Open eVision 2.1 provides explicit geometric calibration only. Field calibration with a reference object will be available in a forthcoming release.

Explicit Geometric Calibration

The explicit calibration makes some strong assumptions on the setup geometry and can only be used when a minimum set of measures are known:

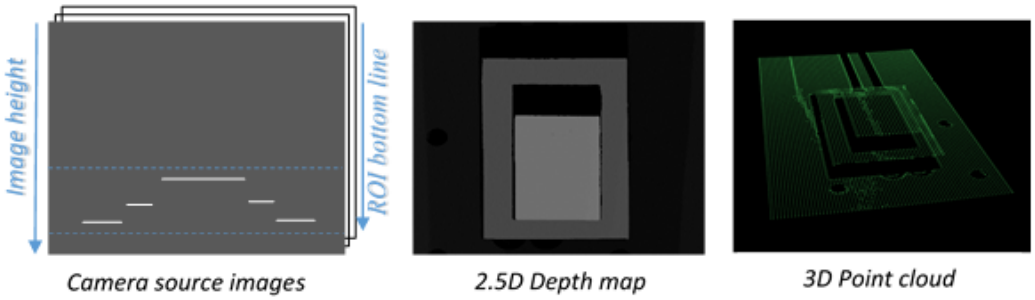
- The angles of the camera and the laser plane
- The distance between the camera and the laser plane
- The field of view of the camera, given as sensor size and optics focal length
- The distance between two line scans of the depth map (depends on acquisition rate and motion speed)



The setup of an explicit geometric calibration uses the constructor of E3DExplicitGeometricCalibration class.

After initialization, the calibration can be applied to a depth map. This process outputs a point cloud.

For correct calculation, the number of lines in the source image (image height) must be provided. If the depth values have been extracted on a region of the source image, the region bottom line must also be given. The method E3DExplicitGeometricCalibration::Apply (srcDepthMap, roiBottomLine, originalImageHeight, dstPointCloud) convert the source depth map to the destination point cloud. ROI bottom line and original image height parameters are requested.



3D Pre-Processing

The 3D processing functions allow you to manipulate point cloud to simplify or speed-up subsequent processing.

Open eVision currently provides two types of pre-processing functions:

- Transforms, that move or modify the shape of the cloud
- Cropping, that gets rid of uninteresting points.

Affine Transforms

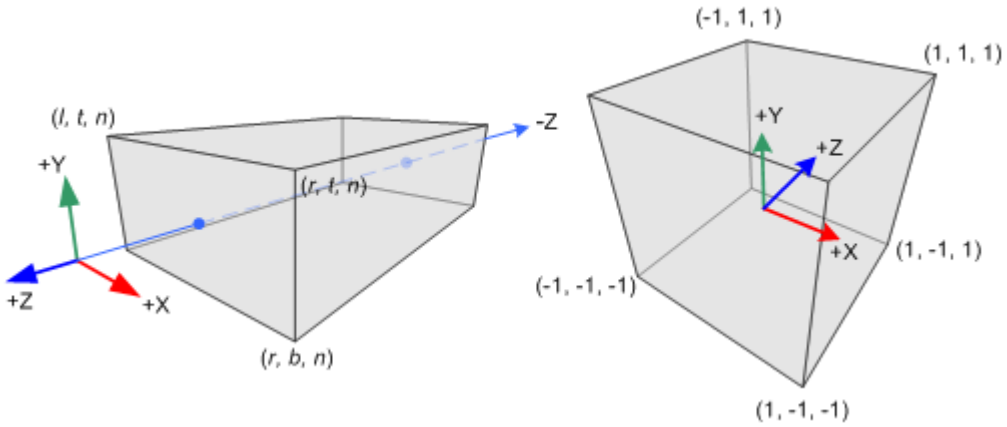
Affine transforms allow you reposition the point cloud inside the 3D space.

Open eVision provides you with the following basic transformations:

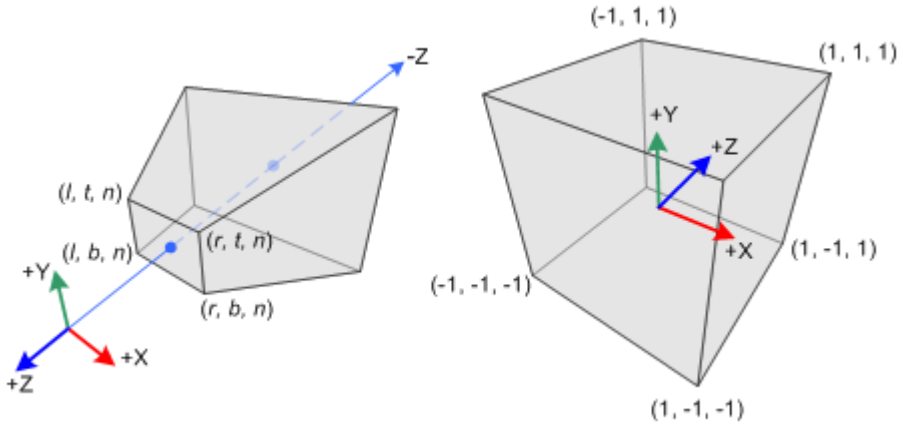
- Rotation around the X, Y or Z axis
- Translation along the X, Y and/or Z axis
- Scaling, around the origin, and either isotropic (the same in all directions) or anisotropic (different along the different axis)

It also provides you with projection transformations, both orthographic and perspective:

- An orthographic projection transforms a volume of space in the shape of a rectangular parallelepiped (and the points it contains) into the canonic view (a cubic space of size 2 and centered on the origin). This projection allows, for instance, you to retrieve side views of a point cloud.



- A perspective projection transforms a volume of space in the shape of a frustum (basically a truncated pyramid) into the canonic view. This projection allow you to simulate the perspective effect given by an eye or a camera.



Cropping

Cropping allows you to exclude points from the point cloud based on geometrical considerations.

Open eVision provides the following cropping functionalities:

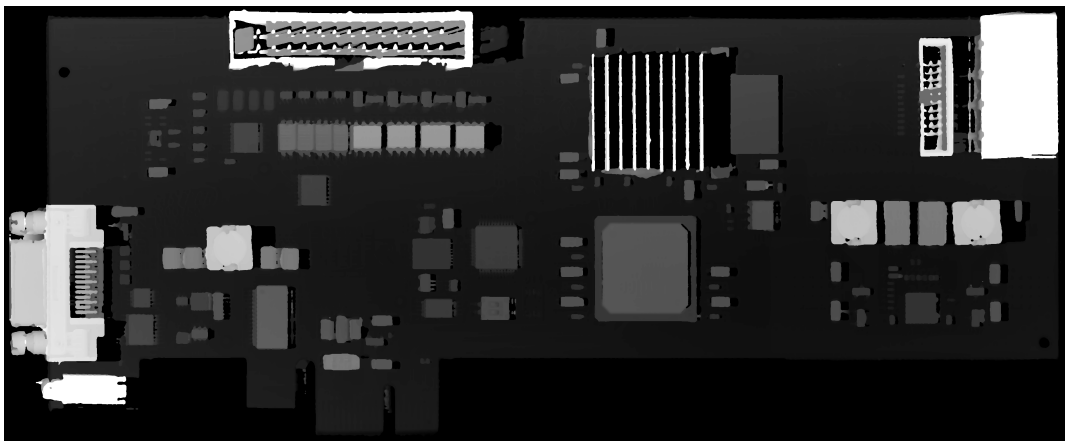
- Simple cropping on the X, Y and/or Z coordinates.
- Cropping the points outside (or inside) a rectangular parallelepiped.
- Cropping the points outside (or inside) a sphere.

Use case

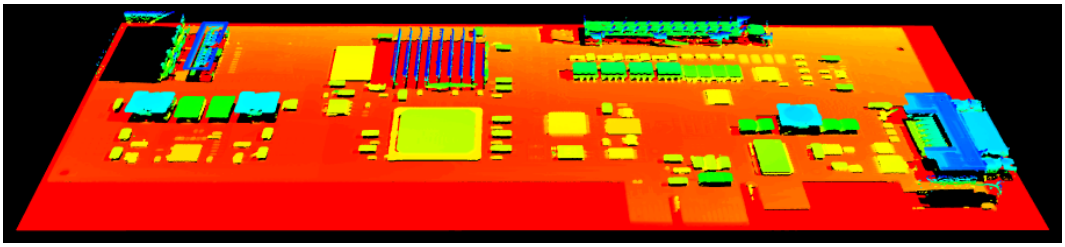
PCB 3D inspection

With Easy3D, it is possible to use Depth Maps for PCB inspection. This section presents a simple detection of missing or misplaced components on a PCB. The processing is done entirely with 2D images but use depth maps as inputs. The workflow is as follow:

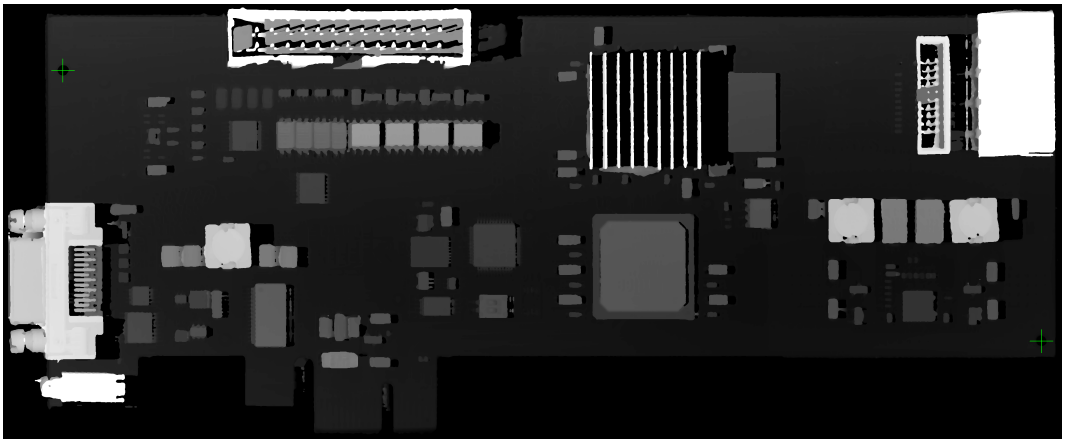
- Perform a 3D acquisition or create the Depth Map with software laser line extraction (ELaserLineExtractor class). Retrieve the grey scale image corresponding to the Depth Map (EDepthMapROI8.AsEImageBW8() method);
- Align the image using fiducial markers (EMatcher class);
- Search for the PCB plane and subtract it from the aligned image, only the components and the connectors remain (EasyImage::Oper(EArithmeticLogicOperation_Subtract…) function);
- Compare the processed image to a golden sample to detect missing or misplaced components (EasyImage::Oper(EArithmeticLogicOperation_Compare…) or EChecker)



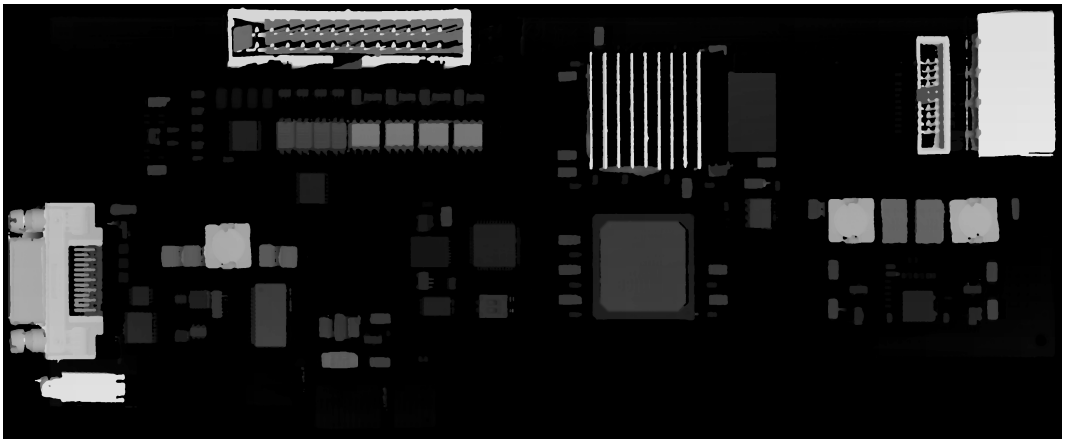
The source Depth Map of a PCB, outputs of CoaxLink Quad 3D-LLE



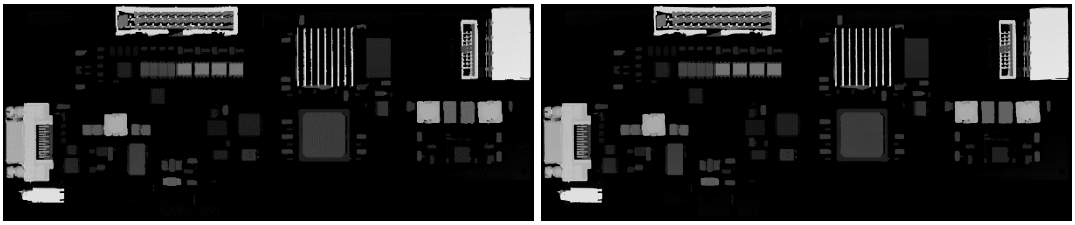
The same Depth Map displayed as a 3D point cloud with false colors

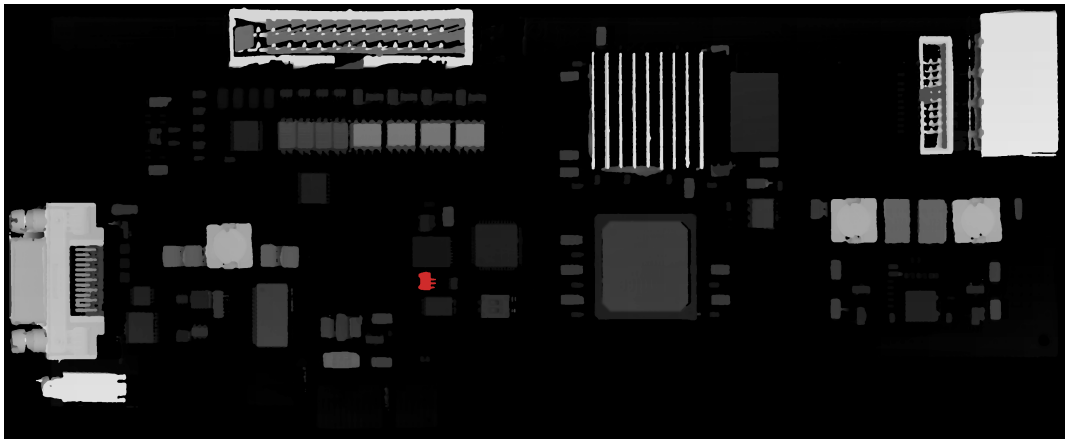


Align the image using fiducial markers (2 holes)



The image with reference plane subtracted, leaving only the components.





The comparison of the image (left) with the golden sample (right, processed with the same workflow) shows the missing component (in red).

EasyImage

EasyImage operations prepare images so that further processing gets better results by:

- isolating defects using thresholding or intensity transformations
- compensating perspective effects (for non-flat surfaces such as a bottle label)
- processing complex or disconnected shapes using flexible masks

The main functions are:

- Intensity Transformations change the gray-level of each pixel to clarify objects (histogram stretching).
- Thresholding transforms a binary image into a bi- or tri-level grayscale image by classifying the pixel values.
- Arithmetic and logic functions manipulate pixels in two images, or one image and a constant.
- Non-Linear Filtering functions use non-linear combinations of neighboring pixels (using a kernel) to highlight a shape, or to remove noise.
- Geometric transforms move selected pixels to realign, resize, rotate and warp.
- Noise Reduction and Estimation functions ensure that noise is not unacceptably enhanced by other operations (thresholding, high-pass filtering).
- Gradient Scalar generates a gradient direction or gradient magnitude map from a gray-level image.
- Vector operations extract 1-dimensional data from an image into a vector, for example to detect scratches or outlines, or to clarify images.
- Harris corner detector returns a vector of points of interest in a BW8 image.
- Canny edge detector returns a BW8 image of the edges found in a BW8 image.
- Overlay overlays an image on top of a color image.
- Operations on Interlaced Video Frames eliminate interlaced image artifacts by rebuilding or re-aligning fields.
- Flexible Masks help process irregular shapes in EasyImage.

Intensity Transformation

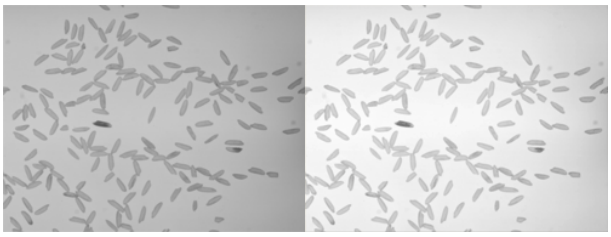
These EasyImage functions change the gray-levels of pixels to increase contrast.

GAIN OFFSET

`Gain Offset` changes each pixel to [old gray value * Gain coefficient + Offset].

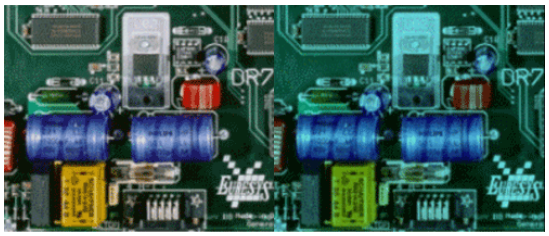
- **gain** adjusts **contrast**. It should remain close to 1.
- **offset** adjusts **intensity** (brightness). It can be positive or negative.
- The resulting values are always saturated to range [0..255].

In this example, the resulting image has better contrast and is brighter than the source image.



Source and result images (with gain = 1.2 and offset = +12)

Color images have three separate gain and offset values, one per color component (red, green, blue).

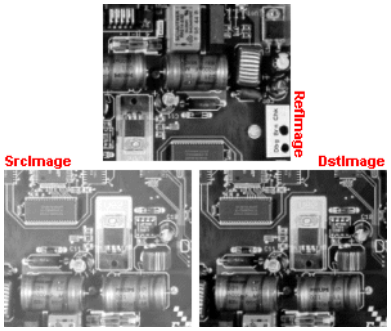


Example of gain/offset applied on a color image

NORMALIZATION

`Normalize` makes images of the same scene comparable, even with different lighting.

It compares the average gray level (brightness) and standard deviation (contrast) of the source image and a reference image. Then, it normalizes the source image with gain and offset coefficients such that the output image has the same brightness and contrast as the reference image. This operation assumes that the camera response is reasonably linear and the image does not saturate.

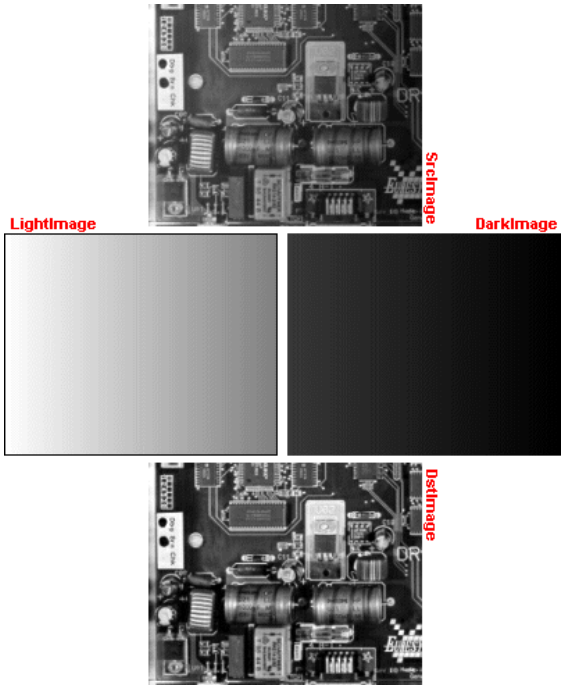


**The reference image (from which the average and standard deviation are computed),
the source image (too bright),
and the normalized image (contrast and brightness are the same as the reference image)**

UNIFORMIZATION

`Uniformize` compensates for non-uniform illumination and/or camera sensitivity based on one or two reference images. The reference image should not contain saturated pixel values and have minimum noise.

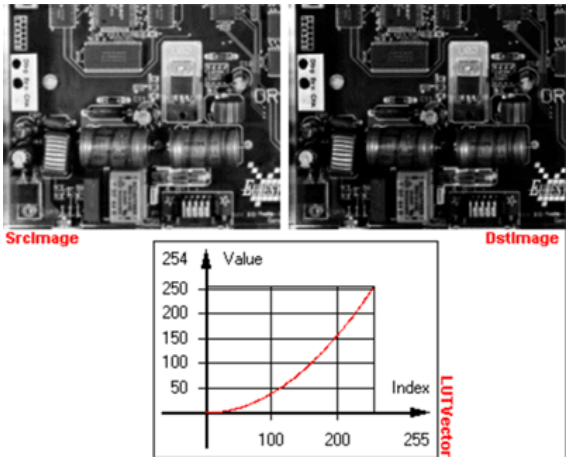
- When one reference image is used, the transformation is similar to an adaptive (space-variant) gain; each pixel in the reference image encodes the gain for the corresponding pixel in the source image.
- When two reference images are used, the transformation is similar to an adaptive gain and offset; each pixel in the reference images encodes either the gain or the offset for the corresponding pixel in the source image.



Example of an image uniformized with two reference images

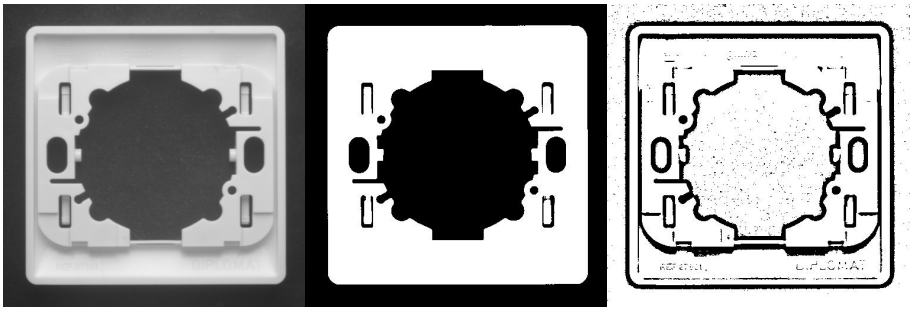
LOOKUP TABLES

`Lut` uses a lookup table of new pixel values to replace the current ones - efficient for BW8 and BW16 images. If the transform function never changes, it is best to use a lookup table.



Example of a transform

Thresholding



Thresholding transforms an image by classifying the pixel values using these methods:

- "Automatic thresholding" on the next page (BW8 and BW16 images only)
- "AutoThreshold" on the next page (BW8 and BW16 images only)
- "Manual thresholding" on the next page using one or two threshold values
- "Histogram based" on the next page (computed before using the thresholding function)

These functions also return the average gray levels of each pixel below and above the threshold.

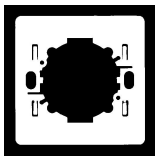
KEYS TO SUCCESSFUL THRESHOLDING

- Object and background areas should be of uniform color and illumination. Image uniformization may be required prior to thresholding.
- The gray level range of the object and background must be sufficiently different (all background pixels should be darker than the darkest object pixel).
- You must decide if the threshold value should be:
  - constant: **absolute** threshold
  - adapted to ambient light intensity: **relative** or **automatic** threshold

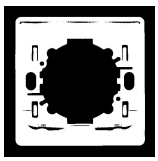
AUTOMATIC THRESHOLDING

The threshold is calculated automatically if you use one of these arguments with the `EasyImage::.Threshold` function.

Min Residue: Minimizes the quadratic difference between the source and resulting image (default if Threshold function is invoked without a argument).



Max Entropy: Maximizes the entropy (i.e. the amount of information) between object and background of the resulting image.



Isodata: Calculates a threshold value that is an average of the gray levels: halfway between the average gray level of pixels below the threshold, and the average gray level of pixels above the threshold.

MANUAL THRESHOLDING

Manual thresholds require that the user supplies one or two threshold values:

- **one** value to the `Threshold` function to classify source image pixels (BW8/BW16/C24) into two classes and create a bi-level image. This can be:
  - `relativeThreshold` is the percentage of pixels below the threshold. The Threshold function then computes the appropriate threshold value, or
  - `absoluteThreshold`. This value must be within the range of pixel values in the source image.
- **two** values to the `DoubleThreshold` function to classify source image pixels (BW8/BW16) into three classes and create a tri-level image.
  - `LowThreshold` is the lower limit of the threshold
  - `HighThreshold` is the upper limit of the threshold

HISTOGRAM BASED

When a histogram of the source image is available, you can speed up the automatic thresholding operation by computing the threshold value from the histogram (using `HistogramThreshold` or `HistogramThresholdBW16`) and using that value in a manual thresholding operation.

These functions also return the average gray levels of each pixel below and above the threshold.

AUTOTHRESHOLD

When no source image histogram is available, `AutoThreshold` can still calculate a threshold value using these threshold modes: EThresholdMode_Relative, _MinResidue, _MaxEntropy and _

Isodata.

This function supports flexible masks.

Arithmetic and Logic

Reasons you may use arithmetic and logic are:

- **to emphasize differences** between images by subtracting the pixels (a conformity check).
- **to compensate for non-uniform lighting** by dividing the target image by the image of the background alone.
- **to remove unwanted areas of an image** by preparing an appropriate mask, and clearing all the pixels that belong to the mask by using logical combinations of pixels.
- **to create a combined image** by combining the pixels of two source images to generate a resulting image.

Arithmetic operations are handled by the `Oper` function, `EArithmeticLogicOperation` enum lists all supported operators.

These operations can be applied to images and constants, they have one or two source arguments (image or integer constants) and one destination argument. If the source operands are a color and a gray-level image, each color component combines with the gray-level component to give a color image. Histogram equalization can improve your results.

ARITHMETIC AND LOGIC COMBINATIONS

ALLOWED COMBINATIONS

	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	

	General	Copy	Invert	Shift	Logical	Overlay	Set
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

Note: 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.

The classification of operations in the above table are:

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

Non-Linear Filtering

These functions use non-linear combinations of neighboring pixels to highlight a shape, or to remove noise.

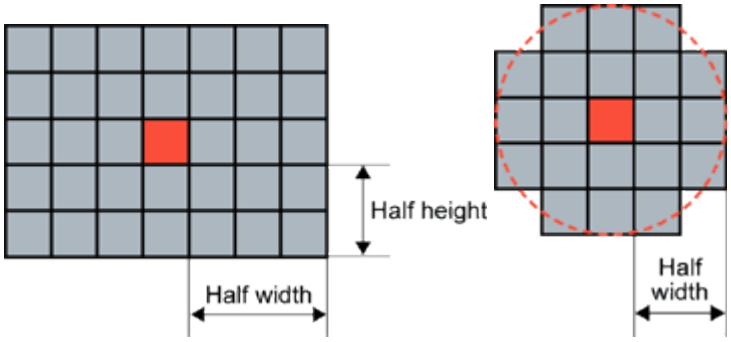
Most can be destructive (except top-hat and median filters) i.e. the source image is overwritten by the destination image. Destructive operations are faster.

All have a gray image and a bilevel equivalent, for example `ErodeBox` and `BiLevelErodeBox`.

1. They define the required shape by a "Kernel" on the next page (usually in a 3x3 matrix).
2. They slide this Kernel over the image to determine the value of the destination pixel when a match is found:
   - Erosion, Dilation: shrinks / grows image regions.
   - Opening, Closing: removes / fills image region boundary pixels.
   - Thinning, Thickening: erodes / dilates using image pattern matching.
   - Top-Hat filters: retains all the tiny image details while removing everything else.
   - Morphological distance: indicates how many erosions are required to make a pixel black.
   - Morphological gradient: indicates the outer and inner edges of the erosion and dilation processes.

- Median filter: removes impulsive noise.
- Hit-and-Miss transform: detects patterns of foreground /background pixels, can create skeletons.

KERNEL



Rectangular kernel of half width = 3 and half height = 2 (left) Circular kernel of half width = 2 (right)

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: which considers the N values in the given neighborhood, sorts them increasingly and selects the K-th largest. *Three special cases are most often used erosion, dilation and median filter where : K can be 1 (minimum of the set), N (maximum) or N/2 (median).*

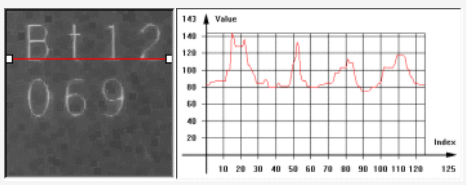
Erosion, Dilation, Opening, Closing, Top-Hat and Morphological Gradient operations all use rectangular or circular kernels of odd size. Kernel size has an important impact on the result.

examples

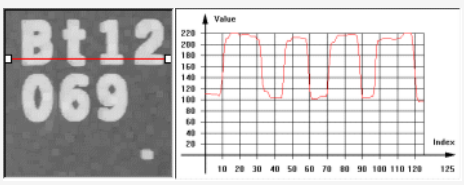
`HalfWidth/HalfHeight`	Actual `width/height`
0	1
1	3
2	5
3	7

EROSION, DILATION

Erosion reduces white objects and enlarges black objects, Dilation does the opposite.



Erosion



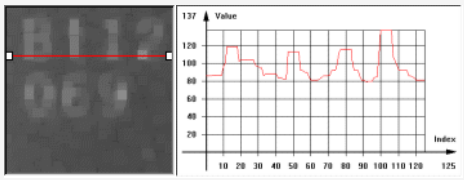
Dilation

Erosion thins white objects by removing a layer of pixels along the objects edges: `ErodeBox`, `ErodeDisk`. As the kernel size increases, white objects disappear and black ones get fatter.

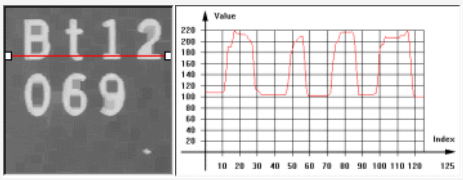
Dilation thickens white objects by adding a layer of pixels along the objects edges: `DilateBox`, `DilateDisk`. As the kernel size increases, white objects get fatter and black ones disappear.

OPENING, CLOSING

Opening removes tiny white objects / dust. Closing removes tiny black holes / dust.



Opening



Closing

An **Opening** is an erosion followed by a dilation using `OpenBox`, `OpenDisk`.
The global effect is to preserve the overall shape of objects, while removing white details that are smaller than the kernel size.

A **Closing** is a dilation followed by an erosion using `CloseBox`, `CloseDisk`.
The global effect is to preserve the overall shape of objects, while removing the black details that are smaller than the kernel size.

THINNING, THICKENING

These functions use a 3x3 kernel to grow (`Thick`) or remove (`Thin`) pixels:

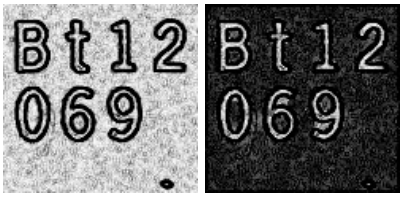
- Thinning: can help edge detectors by reducing lines to single pixel thickness.
- Thickening: can help determine approximate shape, or skeleton.

When a match is found between the kernel coefficients and the neighborhood of a pixel, the pixel value is set to 255 if thickening, or 0 if thinning. The kernel coefficients are:

- 0: matching black pixel, value 0
- 1: matching non black pixel, value > 0
- -1: don't care

TOP-HAT FILTERS

Top-hat filters are excellent for improving non-uniform illumination.



White top-hat filter: source and destination images

They take the difference between an image and its opening (or closure). Thus, they keep the features that an opening (or closing) would erase. The result is a perfectly flat background where only black or white features smaller than the kernel size appear.

- **White top-hat filter** enhances thin white features: `WhiteTopHatBox` ,`WhiteTopHatDisk`.
- **Black top-hat filter** enhances thin black features:`BlackTopHatBoxBlackTopHatDisk`.

MORPHOLOGICAL DISTANCE

`Distance` computes the morphological distance (number of erosion passes to set a pixel to black) of a binary image (0 for black, non 0 for white) and creates a destination image, where each pixel contains the morphological distance of the corresponding pixel in the source image.

MORPHOLOGICAL GRADIENT

The morphological gradient performs edge detection - it removes everything in the image but the edges.

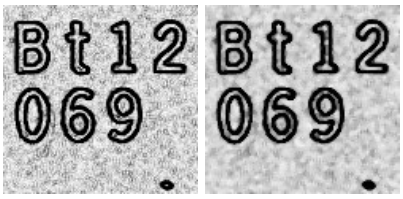
The morphological gradient is the difference between the dilation and the erosion of the image, using the same structuring element.

`MorphoGradientBox`, `MorphoGradientDisk`.

Dilation – Erosion = Gradient

MEDIAN

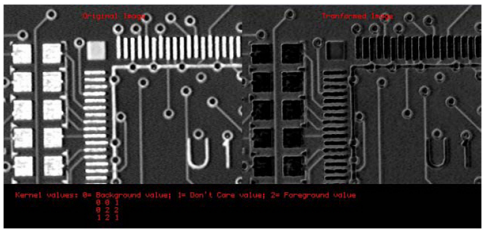
The Median filter removes impulse noise, whilst preserving edges and image sharpness.
It replaces every pixel by the median (central value) of its neighbors in a 3x3 square kernel, thus, outer pixels are discarded.



Median filter: source and destination images

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.



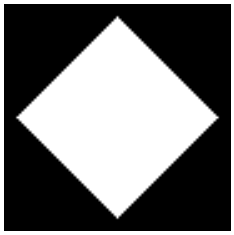
Hit-and-miss transform

The `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. The sizes of the source and destination images must be identical.
- A kernel of type `EHitAndMissKernel` Two constructors are available for the kernel object:
  - `EHitAndMissKernel(int startX, int startY, int endX, int endY)` where:
    `startX`, `startY` are coordinates of the top left of the kernel, must be less than or equal to zero.
    `endX`, `endY` are coordinates of the bottom right of the kernel, must be greater than or equal to zero.
    The constructed kernel has no explicit restrictions on its size, and the following characteristics:
    `kernel width = (endX – startX + 1), kernel height = (endY – startY + 1)`
  - `EHitAndMissKernel(unsigned int halfSizeX, unsigned int halfSizeY)` where:
    `halfSizeX` is half of the kernel width – 1, must be greater than zero.
    `halfSizeY` is half of the kernel height – 1, 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`

EXAMPLE: DETECTING CORNERS IN A BINARY IMAGE.

The hit-and-miss transform can be used to locate corners.

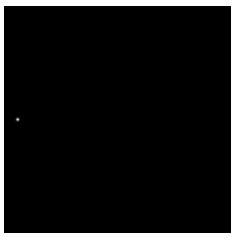


Binary source image

1. Define the kernel by detecting the left corner. The left corner pixel has black pixels on its immediate left, top and bottom; and it has white pixels on its right. The following hit-and-miss kernel will detect the left corner:

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

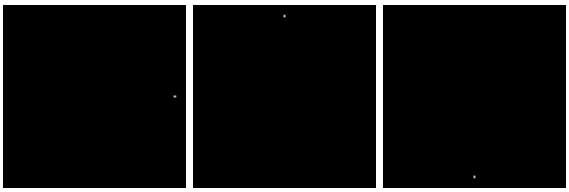
2. Apply the filter on the source image. Note that the resulting image should be properly sized.



Resulting image, highlighted pixel is located on left corner of rhombus

3. Locate the three remaining corners in the same way: Declare three kernels that are the rotation of the filter above and apply them.

4. Detect the right, top and bottom corners.



Geometric Transforms

Geometric transformation moves selected pixels in an image, which is useful if a shape in an image is too large / small / distorted, to make point-to-point comparisons possible.

The selected area may be any shape, but the resulting image is always rectangular. Pixels in the destination image that have corresponding pixels that are outside of the selected area are considered not relevant and are left black.

When the source coordinates for a destination pixel are not integer, an interpolation technique is required.
The nearest neighborhood method is the quickest - it uses the closest source pixel.
The bi-linear interpolation method is more accurate but slower - it uses a weighted average of the four neighboring source pixels.

POSSIBLE GEOMETRIC TRANSFORMATIONS ARE:

REALIGNMENT

The simplest way to realign two misaligned images is to accurately locate features in both images (landmarks or pivots, using pattern matching, point measurement or other) and realign one of the images so that these features are superimposed.

You can register an image by realigning one, two or three pivot points to reference positions. For best accuracy, the pivot points should be as far apart as possible.

- A **single pivot point** transform is a simple translation. If interpolation bits are used, sub-pixel translation is achieved.
- **Two pivot points** use a combination of translation, rotation and optionally scaling. If scaling is not allowed, the second pivot point may not be matched exactly. Scaling should not normally be used unless it corresponds to a change of lens magnification or viewing distance.
- **Three pivot points** use a combination of translation, rotation, shear correction and optionally scaling. A shear effect can arise when acquiring images with a misaligned line-scan camera.

MIRRORING

This destructive feature modifies the source image to create a mirror image:

- horizontally (the columns are swapped) or
- vertically (the rows are swapped).

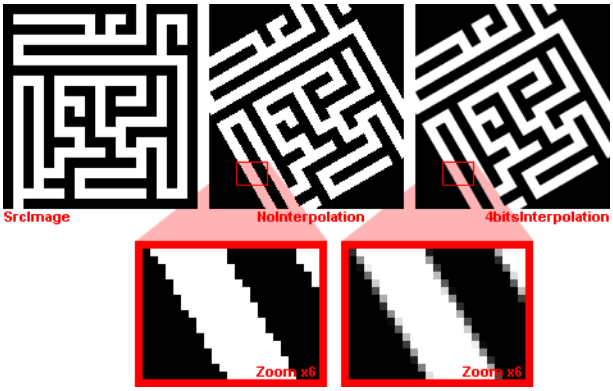
TRANSLATION, SCALING AND ROTATION

If the position or size of an object of interest changes, you can measure the change in position or size and generate a corrected image using the ScaleRotate and Shrink functions.

`EasyImage::.ScaleRotate` performs:

- Image translation: you provide the position coordinates of a pivot-point in the source image and a corresponding pivot point in the destination image.
- Image scaling: you provide scaling factor values for X- and Y-axis.
- Image rotation: you provide 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.



Scale and rotate example

SHRINK

`EasyImage::.Shrink`: resizes an image to be smaller, applying pre-filtering to avoid aliasing.

LUT-BASED UNWARPING

If the feature of interest is distorted due to its shape (anamorphosized), you can unwarp a circular ring-wedge shape (such as text on CD labels)into a straight rectangle. A ring-wedge is delimited by two concentric circles and two straight lines passing through the center.

`EasyImage::.SetCircleWarp` prepares warp images for use with function `EasyImage::.Warp` which moves each pixel to locations specified in the "warp" images which are used as lookup tables.

Noise Reduction and Estimation

Noise can degrade the visual quality of images, and certain processing operations (thresholding, high-pass filtering) will enhance noise in a non-acceptable way. 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. One distinguishes several forms of noise:

- **additive**: noise amplitude is not related to image contents
- **multiplicative**: noise amplitude is proportional to local intensity
- **uniform**: noise amplitude follows a smooth distribution centered around zero
- **impulse**: noise amplitude is infinite.

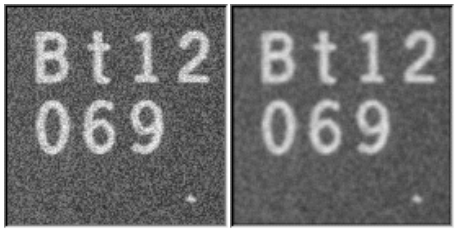
Impulse noise produces a "salt and pepper" effect, while uniform noise blends.

SPATIAL NOISE REDUCTION (IF YOU ONLY HAVE 1 IMAGE):

Reduces uniform and impulse noise but blurs edges.
Cannot distinguish noise from actual signal changes, so always spoils part of the signal.
Uses the correlation between neighboring pixel values to perform convolution or median

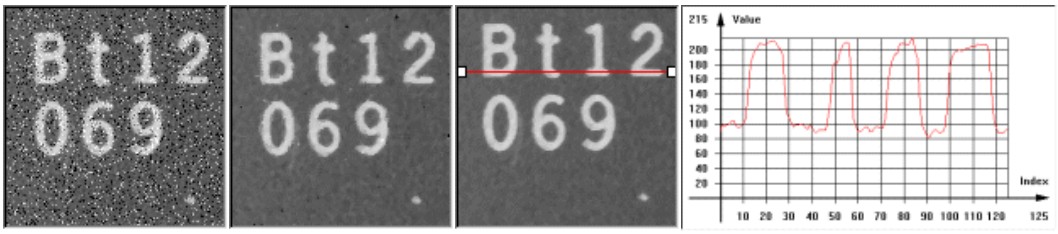
filtering:

- **Convolution** replaces the value at each pixel by a combination of its neighbors, leading to a localized averaging. Linear filtering is recommended to reduce uniform noise. Beware that it tends to blur edges.



Uniform noise reduction by low-pass filtering

- **Median filtering** replaces each pixel by the median value in the pixel neighborhood (5-th largest value in a 3x3 neighborhood). This reduces impulse noise and keeps sharpness.



Impulse noise reduction by median filtering

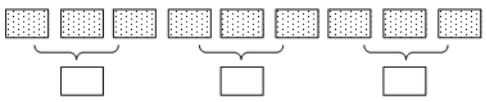
  - `EasyImage::.Median`
  - `EasyImage::.BiLevelMedian`

TEMPORAL NOISE REDUCTION (FOR SEVERAL IMAGES, E.G. MOVING OBJECTS):

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

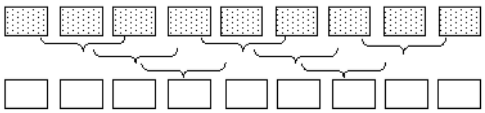
EasyImage provides three ways to minimize noise by means of several images:

- **Temporal average**: just accumulates N images and average them; using standard arithmetic operations, as illustrated below. Creates de-noised image after N acquisitions using average values. Noise varies from frame to frame while the signal remains unchanged, so if several images of the same (still) scene are available, noise can be separated from the signal.
  The disadvantage of producing one de-noised image after N acquisitions only, is that fast display refresh is not possible.



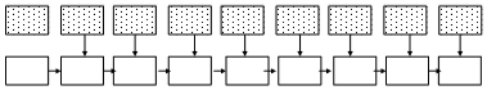
Simple average

- **Temporal moving average**: accumulates the last N images and updates the de-noised image each time a new one is acquired, in such a way that the computation time does not depend on N. The whole process is handled by `EMovingAverage`. The disadvantage of this method is that it combines noisy images together.



Moving average

- **Temporal recursive average**: combines a noisy image with the previously de-noised image using `EasyImage::.RecursiveAverage`.



Recursive average

RECURSIVE AVERAGING

This 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 the following:

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

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. 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 value variations between successive images are usually noise, while large variations correspond to changes in the image.

`EasyImage::.RecursiveAverage` uses this observation and applies stronger noise reduction to small variations and conversely. This reduces noise in still areas and trailing in moving areas.

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

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

NOISE ESTIMATION (OF IMAGE COMPARED TO REFERENCE IMAGE):

To estimate the amount of noise, 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.) Calculates the root-mean-square amplitude and signal-to-noise ratio.

- `EasyImage::.RmsNoise` computes the root-mean-square amplitude of noise, by comparing a given image to a reference image. This function supports flexible mask and an input mask argument. 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.
- `EasyImage::.SignalNoiseRatio` computes the signal to noise ratio, in dB, by comparing a given image to a reference image. This function supports flexible mask and an input mask argument. 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.

Signal amplitude is the sum of the squared pixel gray-level values.

Noise amplitude is the sum of the squared difference between the pixel gray-level values of the given image and the reference.

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.

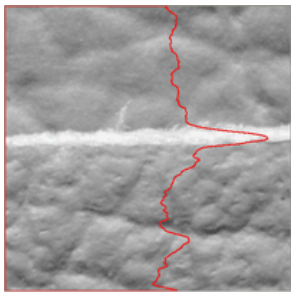
The gradient of a grayscale image corresponds to a vector, the components of which are the partial derivatives of the gray-level signal in the horizontal and vertical direction. A vector can be characterized by a direction and a length, corresponding to the gradient orientation, and the gradient magnitude.

This function 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 `EasyImage::.ArgumentImage` or `EasyImage::.ModulusImage` once before calling `GradientScalar`.

Vector Operations

Extracting 1-dimensional data from an image generates linear sets of data that are handled as vectors. Subsequent operations are fast because of the reduced amount of data. The methods are either:

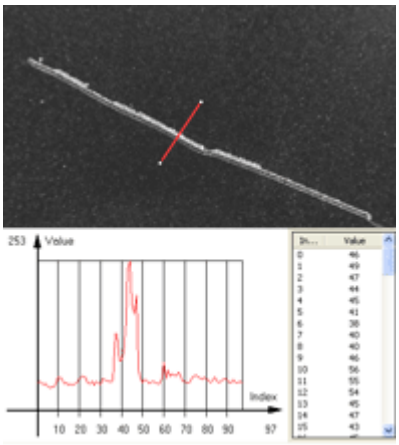
PROJECTION



Projects the sum or average of all gray color-level values in a given direction, into various vector types (levels are added when projecting into an `EBW32Vector` and averaged when projecting into an `EBW8Vector`, `EBW16Vector` or `EC24Vector`). These functions support **flexible mask**.

- `EasyImage::.ProjectOnAColumn` projects an image horizontally onto a column.
- `EasyImage::.ProjectOnARow` projects an image vertically onto a row.

PROFILE



Samples a series of pixel values along a given segment, path or contour, then analyze and modify their Peaks and Transitions to make images clearer:

1. ### OBTAIN THE PROFILE OF A LINE SEGMENT / PATH / CONTOUR.

   `EasyImage::.ImageToLineSegment` copies the pixel values along a given line segment (arbitrarily oriented) to a vector. The line segment must be entirely contained within the image. The vector length is adjusted automatically. This function supports flexible mask.

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

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

2. ANALYSE THE PROFILE TO FIND PEAKS OR TRANSITIONS.

   `EasyImage::.GetProfilePeaks` detects max and min peaks in a gray-level profile. To eliminate false peaks due to noise, two selection criteria are used. The result is stored in a peaks vector.
   A peak is a maximum or minimum of the signal which may correspond to the crossing of a white or black line or thin feature. It is defined by its:

   - Amplitude: difference between the threshold value and the max [or min] signal value.
   - Area: surface between the signal curve and the horizontal line at the given threshold.

   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.
   `EasyImage::.ProfileDerivative` computes the first derivative of a profile extracted from a gray-level image. This derivative transforms transitions (edges) into peaks.
   `EBW8` data type only handles unsigned values, so the derivative is shifted up by 128. Values under 128 correspond to negative derivative (decreasing slope), values above 128 correspond to positive derivative (increasing slope).

3. INSERT THE PROFILE INTO AN IMAGE.

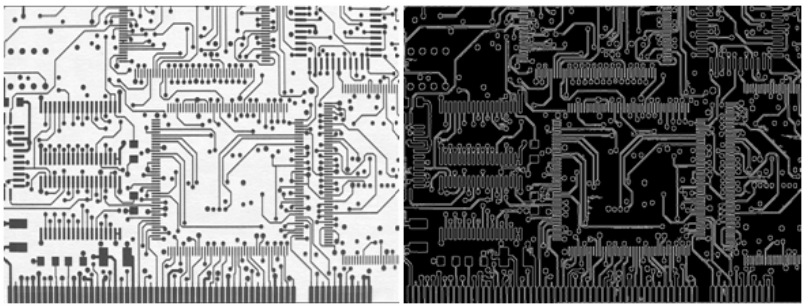
   `EasyImage::.LineSegmentToImage` copies the pixel values from a vector or a constant to the pixels of a given line segment (arbitrarily oriented). The line segment must be wholly contained within the image.

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

Canny Edge Detector

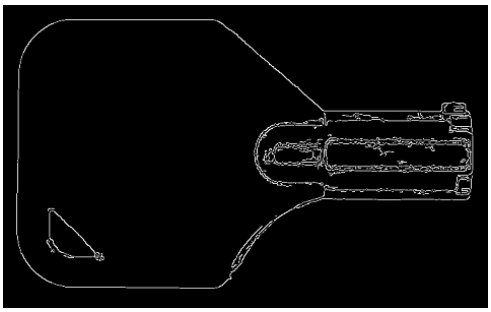
The Canny edge detector facilitates:

- Good detection: finds all edges
- Good localization: the found edges are as close as possible to the "real" edges in the image
- Minimal response: one edge response is accepted for each position, i.e. avoiding multiple close or intersecting edge responses



Source image and the result after a Canny edge detection

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 **255**; all others are set to **0**. It can adjust the scale analysis, it doesn't allow sub-pixel interpolation and it delivers a binary image after thresholding.



Canny edge detector example

The Canny edge detector requires only two parameters:

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

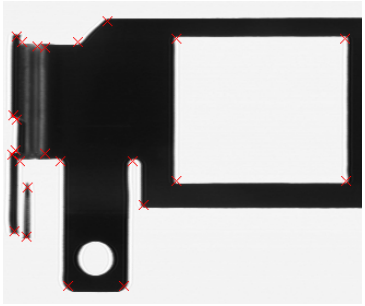
The API of the Canny edge detector is a single class, `ECannyEdgeDetector`, with the following methods:

- `Apply`: applies the Canny edge detector on an image/ROI.
- `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.

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

Harris Corner Detector

The Harris corner detector is invariant to rotation, illumination variation and image noise. It operates on a grayscale BW8 image and delivers a vector of points of interest.



Harris corner detector example

The EasyImage Harris corner detector requires three parameters:

- The integration scale σi: the standard deviation of the Gaussian Filter used for scale analysis. σd = 0,7 x σi, where σd is the differentiation scale: the standard deviation of the Gaussian Filter used for noise reduction during computation of the gradient.
- A corner threshold: a fraction ranging from 0 to 1 of the maximum value of the cornerness of the source image.
- A Boolean that toggles sub-pixel detection.

THE FOLLOWING CHARACTERISTICS ARE AVAILABLE FOR EVERY POINT OF INTEREST:

- Corner position (pixel coordinates with sub-pixel accuracy if enabled).
- Cornerness measurement.
- Gradient magnitude with regards to the differentiation scale σd.
- Gradient value along the X-axis with regards to the differentiation scale σd.
- Gradient value along the Y-axis with regards to the differentiation scale σd

THE API OF THE HARRIS CORNER DETECTOR IS A SINGLE CLASS NAMED `EHarrisCornerDetector` AND THESE METHODS:

- `Apply`: applies 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 Harris detector applications, in order to reduce the setup time.

1. **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.
2. Apply **the detector** with two arguments to the new image : the input image and the interest points in the input image EHarrisInterestPoints.
3. Access the individual elements of the output vector.

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, except the ones that equal the reference color. 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 multicolored overlays.

If a BW8 image is provided as the source image, all the overlay image pixels are copied to the destination image, apart from those that are the reference color which are replaced by the source images.

This function supports flexible mask and an input mask argument. C24, C15 and C16 source images are supported.

Operations 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).

`EasyImage::.RealignFrame` cures this problem if the movement is uniform and horizontal (objects on a conveyor belt), by shifting one of the frames horizontally. 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.

`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.

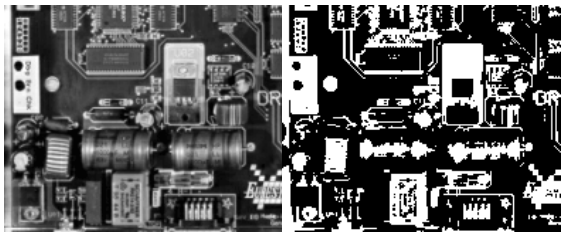
`EasyImage::.SwapFrames`: interchanges the even and odd rows of an image. This is helpful when acquisition of an interlaced image has confused even and odd frames.

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

The size of the destination image is determined as follows:

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

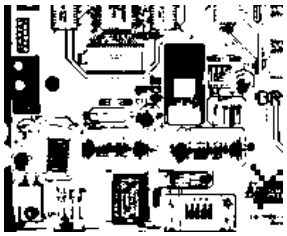
Flexible Masks in EasyImage



Source image (left) and mask variable (right)

SIMPLE STEPS TO USE FLEXIBLE MASKS IN EASYIMAGE

1. **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.
2. **Display the results.**



Resulting image

EASYIMAGE FUNCTIONS THAT SUPPORT FLEXIBLE MASKS

- `EImageEncoder::.Encode` has a flexible mask argument for BW1, BW8, BW16, and C24 source images.
- AutoThreshold.
- Histogram (function `HistogramThreshold` has no overload with mask argument).
- RmsNoise, SignalNoiseRatio.
- Overlay (no overload with mask argument for BW8 source images).
- ProjectOnAColumn, ProjectOnARow (Vector projection).
- ImageToLineSegment, ImageToPath (Vector profile).

EasyColor

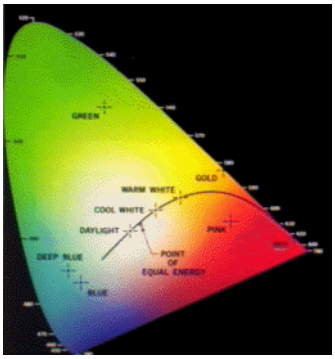
EasyColor makes color image processing as efficient as possible by detecting, classifying and analyzing objects. Several conversion functions mean that any color system can be processed.

COLOR DEFINITION AND SUPPORTED SYSTEMS

WHAT IS COLOR?

The human eye is sensitive to light:

- **Intensity**, or **achromatic** sensation, captured by **grayscale** images.
- **Wavelength**, or **chromatic** sensation, described in red, green and blue primary **colors**. True color digital images (24 bits per pixel; 8 bits per RGB channel) represent as many colors as the eye can distinguish.



Visible color gamut in the XYZ color space

There are three color systems:

- **Mixture** systems (RGB/XYZ) give the proportions of the three primaries to be combined.
- **YUV Luma/chroma** systems (XYZ/YUV) separate the achromatic (Y) and chromatic sensations (U & V). Used when a black and white image is required as well (television).
- **Intensity/saturation/hue** systems (RGB/XYZ/YUV) separate achromatic (black and white Intensity) from enhanced chromatic (color Saturation and Hue) sensations. Used to eliminate lighting effects, or to convert RGB images to another color system. More saturated colors are more vivid, less saturated ones are grayer.

In general:

- RGB is used by monitors, cameras and other display devices.
- YUV is used for efficient transmission of color images by compressing the chrominance information.
- XYZ is used for device-independent color representation.

All image processing operations can use **quantized** coordinates: discrete values in the [0..255] interval, which use a byte representation to store images in a frame buffer.

Color system conversion operations can also use simpler **unquantized** coordinates: continuous values, often normalized to the [0..1] interval.

COLOR IMAGE PROCESSING

A color image is a vector field with three components per pixel. 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 3 ways to process a color image:

- **Component extraction**: you can extract the most relevant feature from the triple color information, to reduce the amount of data. For instance, objects may be distinguished by their hue, a pre-processing step could transform the image to a gray-level image containing only hue values.
- **De-coupled transformation**: you can perform operations separately on each color component. For instance, adding two images together adds the red, green and blue components and stores the result, component by component, in a resulting color image.
- **Coupled transformation**: you can combine all three color components to produce three derived components. For example, converting YIQ to RGB.

SUPPORTED COLOR SYSTEMS

Easycolor supports color systems RGB, XYZ, L*a*b*, L*u*v*, YUV, YIQ, LCH, ISH/LSH, VSH and YSH.

RGB is the preferred internal representation as it is compatible with 24-bit Windows Bitmaps.

	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

TRANSFORM USING LUTS (LOOKUP TABLES)

EasyColors Lookup tables provide an array of values that define what output corresponds to a given input, so an image can be changed by a user-defined transformation.

A color pixel can take 16,777,216 (2^{24}) values, a full color LUT with these entries would occupy 50 MB of memory and transforms would be prohibitively time-consuming. Pre-computed LUTs make color transforms feasible.

To transform a color image, you initialize a color LUT using one of the following functions:

"LUT for Gain/Offset (Color) " on page 86: `EasyImage::.GainOffset`,

"LUT for Color Calibration" on page 87: `Calibrate`,

"LUT for Color Balance" on page 87: `WhiteBalance`,

`ConvertFromRGB`, `ConvertToRGB`.

This color LUT is then used in a transform operation such as `EasyColor::.Transform` or you can create a custom transform using `EColorLookup` which takes unquantized values (continuous, normalized to [0..1] intervals), and specifies the source and destination color systems. Some operations use the LUT on-the-fly thus avoid storing the transformed image, for example to alter the U (of YUV) component while the image is in RGB format.

The optimum combination of **accuracy and speed** is determined by the choice of `IndexBits` and `Interpolation` - the accuracy of the transformed values roughly corresponds to the number of index bits.

- Fewer table entries mean smaller storage requirements, but less accuracy.
- No interpolation gives quicker running time, but less accuracy. Interpolation can recover 8 bits of accuracy per component. When the involved transform is linear (such as YUV to RGB), interpolation always gives exact results, regardless of the number of table entries.

Index Bits	**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

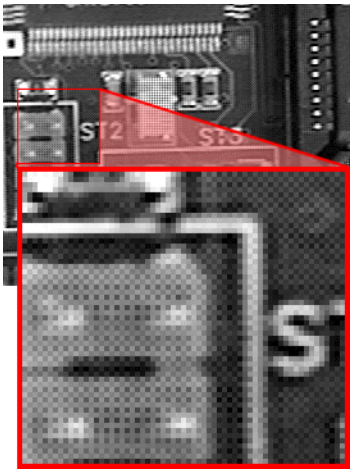
DISCRETE QUANTIZED VS. CONTINUOUS UNQUANTIZED

Color coordinates in the classical systems are normally **continuous** values, often normalized to the **[0..1]** interval. Computations on such values, termed **unquantized**, are simpler.

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

All image processing operations apply to quantized values, but conversion operations can also be specified using unquantized coordinates.

BAYER TRANSFORM



Bayer pattern encoded image

A Bayer encoded image is not compatible with a true color image (EC24), but white balance and gamma correction can be applied to it using `EColorLookup` parameter in `EasyColor::.BayerToC24`.
A Bayer image is three times smaller, so processes much faster.

Easyobject can use the Bayer pattern to create a color image.

TRANSFORM YUV444 / YUV422

YUV images can be minimized without degrading visual quality using function `Format444To422` to convert from 4:4:4 to 4:2:2 format (or you can convert `Format 422 To 444`).

- 4:4:4 uses 3 bytes of information per pixel.
- 4:2:2 uses 2 bytes of information per pixel.
  It stores the **even** pixels of U and V chroma with the **even and odd** pixels of Y luma as follows:

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

MERGE, EXTRACT AND COLOR

A color image contains three color planes of continuous tone images.
A gray-level image can be a component of a color system.

MERGE AND EXTRACT COMPONENTS

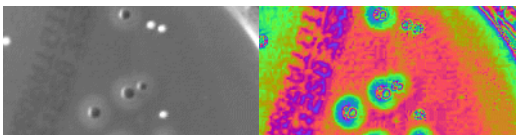
EasyColor can change or extract one plane at a time, or all three together. See `Compose`, `Decompose`, `GetComponent`, `SetComponent`.

These operations can use a color LUT to transform on the fly, they could build an RGB image from lightness, saturation and hue planes.

EasyColor functions perform the necessary interleaving / un-interleaving operations to support Windows bitmap format of interleaved color planes (blue, green and red pixels follow each other).

PSEUDO-COLOR TO TRANSFORM GRAY-LEVEL IMAGES TO COLOR

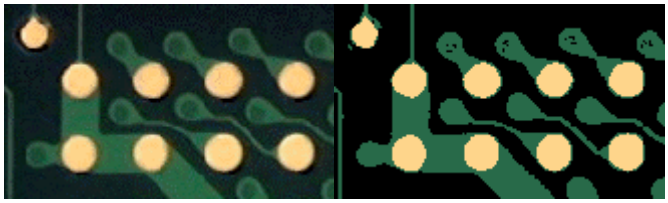
The trick is to define a regular gamut of 256 colors and each color will be assigned to pixels with a corresponding gray-level value.
To define pseudo-color shades, you specify a trajectory in the color space of an arbitrary system. You can then pseudo-color using the drawing functions color palette (see Image and Vector Drawing) then save and/or transform it like any other color image.



Gray-level and pseudo-colored image

SEPARATE COLOR OBJECTS

This EasyColor process takes a set of distinct colors and associates each pixel with the closest color, using a layer index that can then be used in EasyObject with the labeled image segmenter to improve blob creation.



Raw image and segmented image (3 colors)

Bayer Transform

The Bayer pattern is a color image encoding format for capturing color information from a single sensor.
A color filter with a specific layout is placed in front of the sensor so that some of the pixels receive red light only, while others receive green or blue only.
An image encoded by 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.

Bayer vs. true color format

The Bayer pattern normally starts with a GB/RG block in the upper left corner. If the image is cropped, this parity rule can be lost, but 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 gray-level image, into a true color image. There are three ways to reconstruct the missing pixels. The more complex the interpolation, the slower the conversion. However, it is highly recommended to use interpolation.

- **Non-interpolated mode**: duplicates the nearest pixel to above and/or to the left of the current pixel.
- **Standard interpolated mode**: averages relevant neighboring pixels.
- **Improved interpolated mode (recommended)**: interpolates the unknown component values. This mode reduces visible artifacts along object edges.

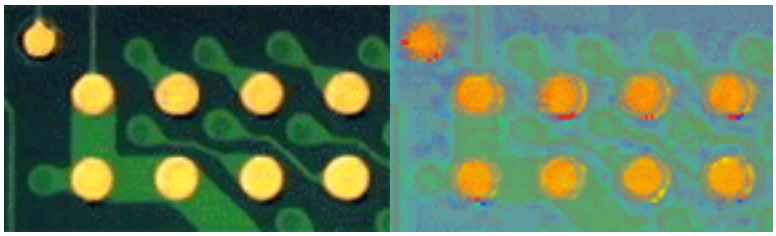
Converted images with no (top), standard (left) and improved interpolation method (right)

LUT for Gain/Offset (Color)

Separate gains and offsets can be applied to each of the three components of an image (contrast enhancement transform). The RGB image must be transformed to the targeted color space, gains and offsets applied, then transformed 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 of the chromatic components should vary in a similar way.
- When applied to intensity/saturation/hue representation, it makes no sense to apply gain and offset to the hue component.



Enhanced saturation / Uniform lightness

The contrast enhancement function can be used to uniformize a given component: setting the gain to 0 for some component has the same result as setting all pixels to the value of the offset for this component.

LUT for Color Calibration

Color distortions introduced by the image acquisition chain can be corrected by comparing sample colors from the image with their true values. A calibrated color chart, such as the IT8, is required.

- Sample colors are the average color in a suitable ROI using `PixelAverage`.
- True color values are specified in the XYZ color system. 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 one, three or four reference colors. In the first case, calibration is a gain adjustment for the three color components. In the second and third case, a linear or affine transform is used.

LUT for Color Balance

A color image can be improved by changing gamma correction and white balance.

These effects can be corrected efficiently by setting up a lookup table using `WhiteBalance` and applying it on a series of images by means of `Transform`.The LUT need only prepared once (it implements a de-coupled color transformation).

GAMMA PRE-COMPENSATION

Many color cameras use a gamma pre-compensation process that deals with the non-linear response of the display device (such as a TV monitor).

Gamma pre-compensation should be used after processing because using it before would change the result because of the non-linearity introduced.

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`

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

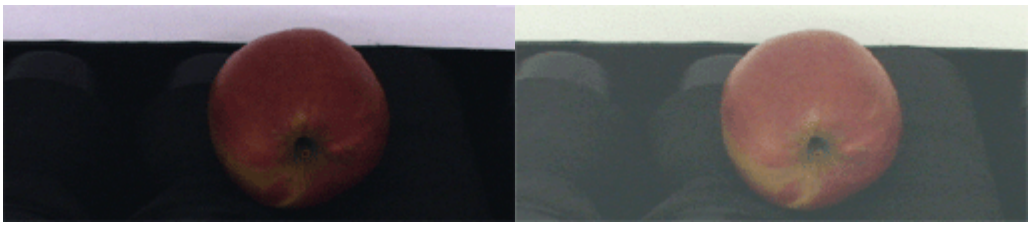
GAMMA PRE-COMPENSATION CANCELLATION

Many color cameras have a built-in gamma pre-compensation feature that can be turned off. If this feature cannot be turned off and is not desired, its effect can be canceled by applying the direct gamma transform. 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`

WHITE BALANCE

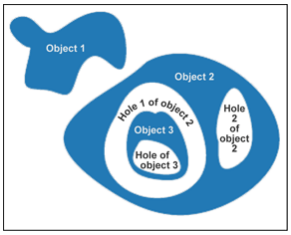
A camera may exhibit color imbalance, i.e. the three color channels having mismatched gains, or the illuminant (the light sources) not being perfectly white. When this occurs, the white areas appear as an unsaturated color. The white balance correction automatically adjusts three independent gains so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



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

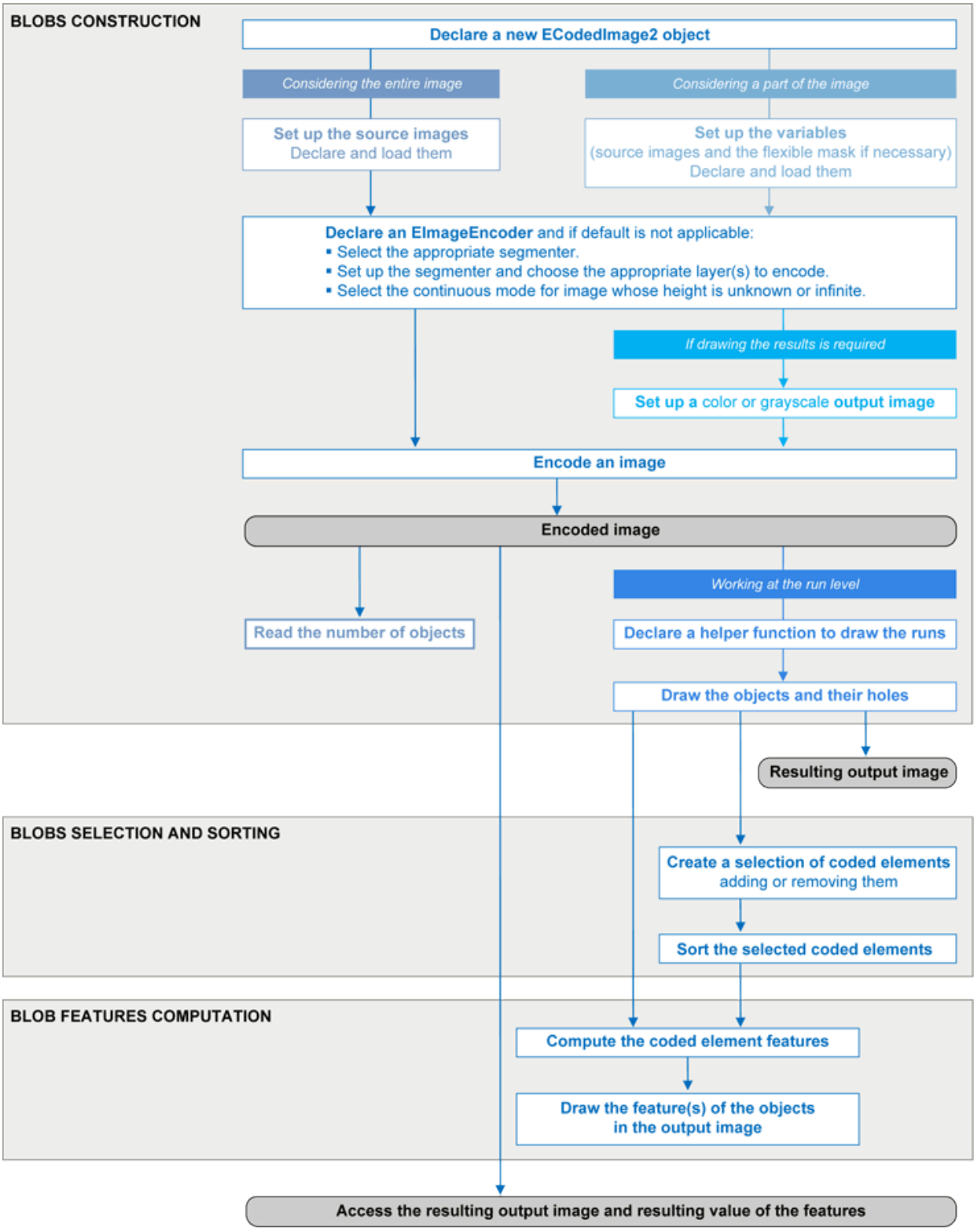
EasyObject

The EasyObject library picks out features in an image by creating and processing blobs (objects or holes that have the same gray level range).



This library can be used for BW1, BW8, BW16 and C24 source images and is accessible from the `ECodedImage2 class` which has improved execution time, especially for large images with many objects.

WORKFLOW



BLOB DEFINITION

A blob is a grouping of neighboring pixels of the same gray level range.
Blobs may be objects or holes in objects. EasyObject functions analyze both objects and holes.
When blobs are built, the inclusion relationship between holes and objects is computed.

Even though holes may be the actual objects of interest, it is easier to find an object of interest, then detect its holes (with EasyObject) and measure their characteristics (with EasyGauge or EasyObject).

Blobs are handled as independent entities:

- They can be selected by means of the layer they belong to, their position, a rectangular ROI or their computed features. The selection criteria can be combined (select the small objects; among these, select those close to the right edge...).
- They can be listed and sorted by their geometric characteristics: such as area, width, or ellipse of inertia.

Blob analysis can be restricted to rectangular and nested ROIs, and to complex or disconnected-shape regions using flexible masks.

BUILD BLOBS

EasyObject chooses objects of interest and constructs blobs/holes in two steps:

1. Segment: classifies the source image pixels, creates layers, and constructs the runs (a run is a sequence of adjacent pixels in a row, that share the same property).
2. Encode: assembles runs, to build blobs for each layer.
   You select which objects or holes are kept.
   `EImageEncoder::.Encode` analyzes the blobs and stores the result into a coded image which has a set of superimposed, mutually exclusive image layers, where the pixels of each layer have properties in common, such as being above a threshold.
   Flexible masks can restrict encoding to an arbitrary shaped area.

There is no need to build holes, they are constructed on-the-fly when required.

FUNCTIONS

- Segmentation `GetSegmentationMethod` and `SetSegmentationMethod`
- Grayscale single threshold `EGrayscaleSingleThresholdSegmenter`
- Grayscale double threshold `EGrayscaleDoubleThresholdSegmenter`
- Color single threshold `EColorSingleThresholdSegmenter`
- Color range threshold `EColorRangeThresholdSegmenter`
- Reference image `EReferenceImageSegmenter`
- Image range `EImageRangeSegmenter`
- Labeled image `ELabeledImageSegmenter`
- Binary images `EBinaryImageSegmenter`

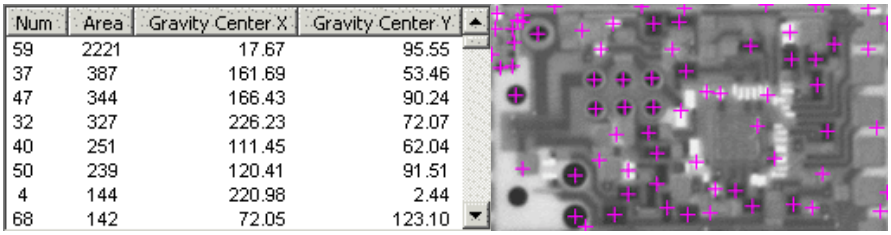
Pixel aggregation (encoder)

- Layer selection
- Object construction: run aggregation into objects
- Hole construction: run aggregation into holes

EXTRACT OBJECTS (USING GEOMETRIC PARAMETERS)

Once an image has been encoded, the coded elements (objects or holes) are accessible through

the abstract class `ECodedElement` which provides a large set of methods applicable to a particular coded element:



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, holes and their features can be efficiently accessed randomly (in an index-based fashion).

Image Segmenters

There are several ways to segment pixels. The method is chosen with `GetSegmentationMethod` and `SetSegmentationMethod`.

1. GRAYSCALE SINGLE THRESHOLD (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 with a gray value 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:

- **Absolute** (integer value): represents the first gray value of the white layer. Set with `SetAbsoluteThreshold` method and got with `GetAbsoluteThreshold` method.
- **Relative** (%): represents the fraction of image pixels that belong to the Black layer, it is a user-defined float value in range 0 to 1. Set with `SetRelativeThreshold` method and got with `GetRelativeThreshold` method.
- **Minimum Residue** (default): The threshold is an automatically computed value such that the quadratic difference between the source and thresholded image is minimized.
- **Maximum Entropy**: automatically computed value such that the entropy (i.e. the amount of information) of the resulting thresholded image is maximized.
- **IsoData**: automatically computed value that lies halfway between the average dark gray value (gray levels below the threshold) and average light gray values (gray levels above the threshold).

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

2. 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 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, set with `SetLowThreshold` and `SetHighThreshold` methods, and got with `GetLowThreshold` and `GetHighThreshold` methods.

3. 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, set with `SetThreshold` method and got with `GetThreshold` method.

4. 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, set with `SetLowThreshold` and `SetHighThreshold` methods and got with `GetLowThreshold` and `GetHighThreshold` methods.

5. IMAGE RANGE

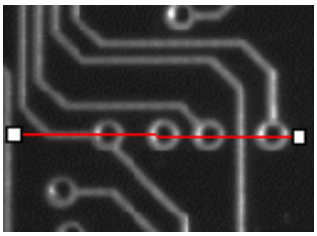
The following cases need a segmentation using **pixel-by-pixel thresholding** which gives an allowed range of values for each pixel:

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

The allowed range for each pixel is specified using two images: a low reference image with the minimum values allowed for each pixel, a high reference image with the maximum values. The reference images are thus the source image minus (or plus) a fixed value all over the image (assuming noise distribution is uniform and additive).
The difficulty is preparing suitable high and low reference images.

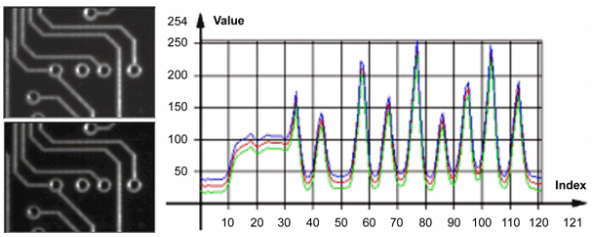
PREPARING HIGH AND LOW REFERENCE IMAGES

You can start from an image of the scene without defects and add security margins before comparison.



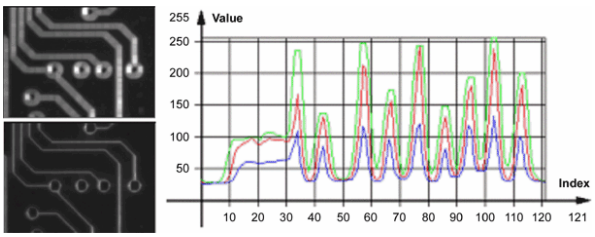
Source image

Gray-level tolerance must be provided for noise and illumination variations.



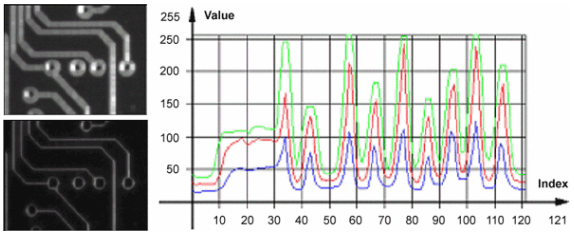
Gray-level tolerance margins

The image may have a slight shift in some direction which can be corrected by enlarging the light and dark areas using dilate and erode morphological operations. This geometric tolerance margin is roughly as large as the morphological filter size.



Geometric tolerance margins

Combining both kinds of tolerance margins gives the best results.



Combined margins

IMAGE SEGMENTER

`EImageRangeSegmenter` and EReferenceImageSegmenter are applicable to BW8, BW16, and

C24 images; and produce 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. The Reference Image defines the reference threshold of each pixel is individually.

- For **grayscale** 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, High or Reference 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, High or Reference Image.
- The **black layer** (usually Layer 0) contains the remaining unmasked pixels.

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

- BW8: `SetLowImageBW8` `GetLowImageBW8`
- BW16: `SetLowImageBW16GetLowImageBW16`
- C24: `SetLowImageC24GetLowImageC24`

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

- BW8: `SetHighImageBW8GetHighImageBW8`
- BW16 `SetHighImageBW16GetHighImageBW16`
- C24 `SetHighImageC24GetHighImageC24`

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

- BW8: `SetReferenceImageBW8`, `GetReferenceImageBW8`
- BW16: `SetReferenceImageBW16`, `GetReferenceImageBW16`
- C24: `SetReferenceImageC24` , `GetReferenceImageC24`

6. 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 which return the lowest and the highest values of the index range respectively.

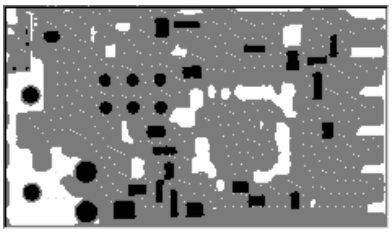
7. BINARY IMAGE

`EBinaryImageSegmenter` is applicable to BW1 binary images; it produces coded images with two layers:

- Black layer (usually index 0) contains unmasked pixels with a binary value equal to zero.
- White layer (usually index 1) contains the remaining unmasked pixels, i.e. unmasked pixels with a binary value equal to one.

Image Encoder

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



Object building

SELECTING THE LAYERS TO ENCODE

The segmentation methods (see Image Segmenters) determine which layer(s) to encode by default, and do not encode pixels from the other layers.

Function `GetMaxLayerIndex` returns the highest Layer Index value . It is available for all segmenters.

Enabling/disabling layer encoding for each layer individually

The following tables list, for each layer, the Set/Get function and the default enable/disable value.

Two-layer segmenters

Layer	Set LayerEncoded function	Get LayerEncoded function	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 tables list, for each layer, the Set/Get function and the default value.

Two-layer segmenters

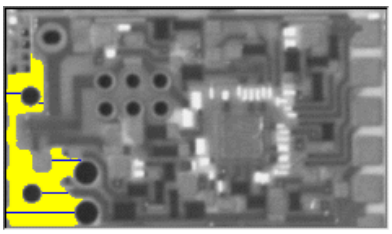
Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	`SetBlackLayerIndex`	`IsBlackLayerIndex`	**0**
White layer	`SetWhiteLayerIndex`	`IsWhiteLayerIndex`	**1**

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**

RUNS

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

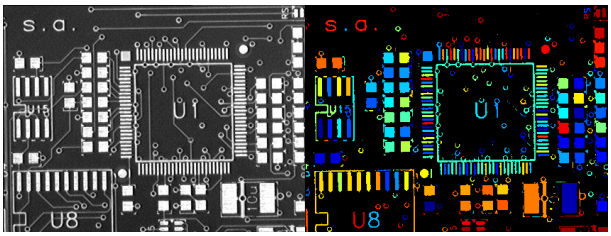


A single object with five enhanced runs

EasyObject can work at object level, and at run level which allows faster processing in critical cases. This is useful to compute custom features on objects then list all runs belonging to a given object as shown in this example of working at run level, with colored runs in the output image.

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

5. **Color the runs in the output image**. Iterate over the objects of a specific layer by constructing a loop and then a `RunsIterator` object. This iterator allows to browse runs of the considered object. Once the iterator has finished a run of the considered object, the inner loop processes the pixels spanned by this run in the output image.
6. **Select a specific layer**.



Source image (left) with the white layer rendered (right)

CONNEXITY

Pixels can touch each other along an edge or by a corner. In Four Connexity only pixels touching by an edge are considered neighbors. In Eight Connexity (the default) pixels touching by a corner are also considered neighbors.

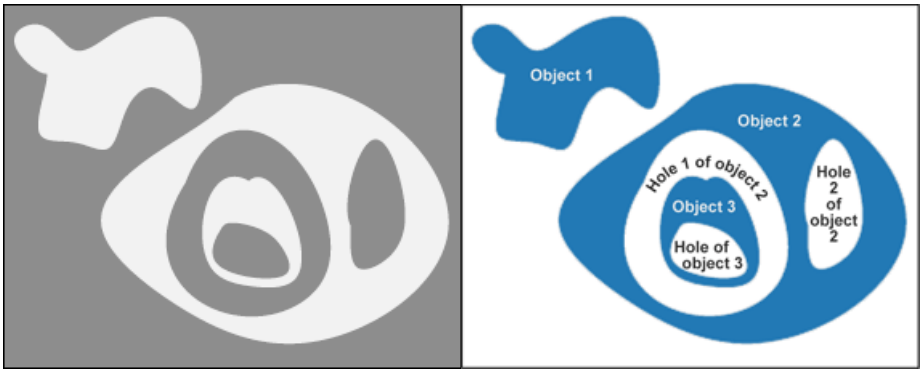
Multiple images can be encoded in continuous mode.

Holes Construction

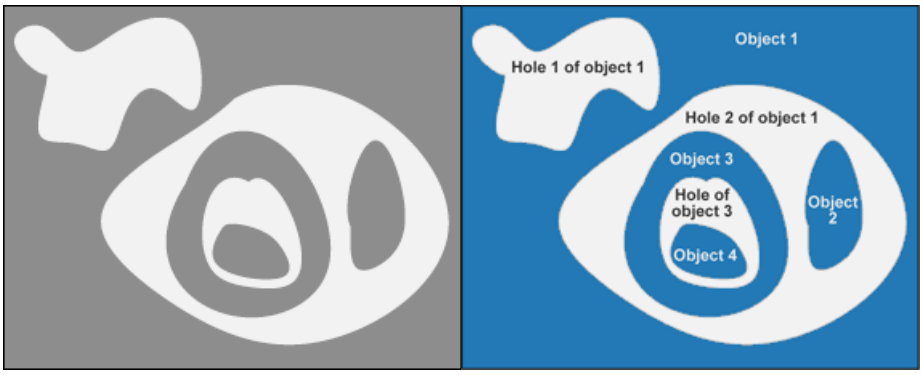
A hole is a set of connected pixels that are entirely surrounded by a parent object (4 or 8 pixels depending on the connexity mode).

A hole has no child. Objects inside a hole are considered as separate objects.

`EObject` and `EHole` classes both derive from `ECodedElement`, so objects and holes are managed in the same way and share the same functions.



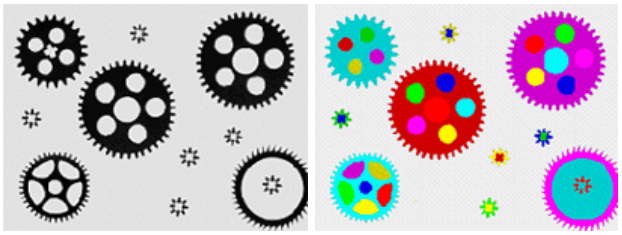
Encoding the white layer (3 objects and 3 holes)



Encoding the black layer (4 objects and 3 holes)

HOW TO COLOR HOLES

1. **Declare a new** `ECodedImage2` object.
2. **Declare an** `EImageEncoder` and, if applicable, select and setup the appropriate segmenter, and choose the appropriate layer(s) to encode.
3. **Setup an output image**.
4. **Encode the image**.
5. **Declare a helper function to draw the runs.** A helper function (see also section Object Construction/Working at the Run Level) draws the runs in an output image, using, for example, a given color. This function can be shared for objects and holes.
6. **Draw the objects and their holes in the output image.** It is necessary to iterate over the objects of the chosen layer.
   a. The helper function draws the runs of each object (`DrawRuns`) using a specific color.
   b. The holes are iterated over the current object, and their runs are drawn.
   c. Each hole of an object is drawn with a different color computed in the global function (`GetFadedColor`) which returns a color that depends upon the hole index, for example a gradation of red to green colors.



Raw image (left) Building of objects and all holes (right)

Normal vs. Continuous Mode

NORMAL MODE (DEFAULT)

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

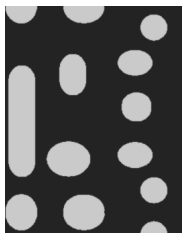
CONTINUOUS MODE

In continuous mode EasyObject can process an image whose height is unknown or infinite (e.g. coming from a line-scan camera). The image is split into a several chunks that are fed into an image encoder. Objects that straddle several successive image chunks can be detected.

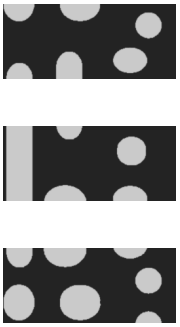
The image encoder encodes only the objects that contain no run touching the last row of the source image. Objects that touch the inferior border of the image are not written in the coded image because they are expected to continue in subsequent image chunks, but they are kept in memory and are processed when subsequent images are analyzed.

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

IN THIS EXAMPLE, WE GENERATE A SEQUENCE OF COLOR IMAGES THAT EXHIBIT OBJECTS ENCODED OVER SUCCESSIVE CHUNKS

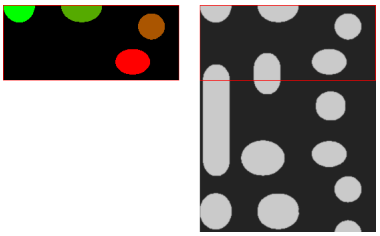


Original image

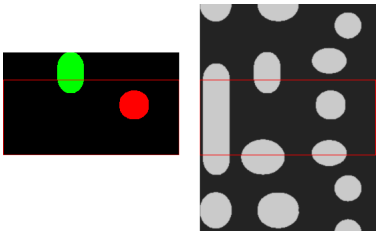


Three chunks of the image

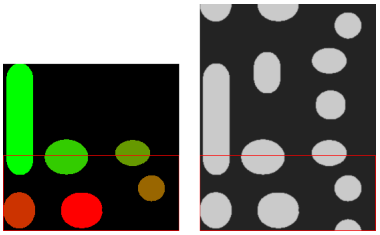
1. **Draw the objects encoded in a layer of a coded image.** This code is essentially the same as in "Browsing Runs" code snippet. The only difference is that an offset can be applied along the Y-axis.
2. **Define a function to draw the objects of a layer.** If a coded image contains objects that were started in a previous image: the runs of this object from the previous image are assigned with a negative Y-coordinate.
   The zero Y-coordinate is the first row of the most recently encoded image. The convention is to assign the lowest Y-coordinate to the oldest run in the encoded objects.
   The method `EImageEncoder::.GetStartY` obtains 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(computed by `GetFadedColor`). This function closely follows the function `DrawRuns`, but is adapted to continuous mode by taking **GetStartY** into account.
3. **Enable continuous mode** in property `EImageEncoder::.SetContinuousModeEnabled`. Additional variables can be declared, for example to store the successive encoded image, or to hold the output images.
4. **Analyze the successive chunks.** To encode successive chunks use `Encode(chunk[count], codedImage)` and then `DrawLayer`. **Note:** The variable count spans integers 0, 1 and 2. When an object from a chunk is not complete 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.
Note that two objects in the lower-left of the image chunk are not encoded,
because they touch the border of the chunk.**



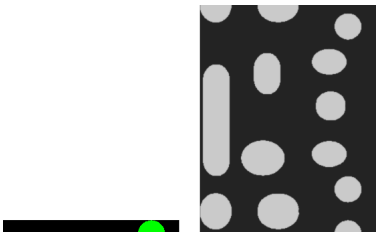
**When count reaches 1, one of these two objects becomes completed,
which leads to the encoding of the following image.
Two other objects are not encoded yet at this time.
Here is the result of the encoding of the last chunk (count = 2).**



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

Flushing Continuous Mode

After encoding the three image chunks, there remains one object to be completed (in the bottom-right corner of the large image). However, as there are no more chunks, it is necessary to explicitly close this object and encode the remaining object using the flushing of the image encoder. The internal memory of the image encoder is then empty.



Result of the flush

Selecting and Sorting Blobs

The object segmentation process considers any blob as an object, including noise pixels which appear as tiny objects. You can select which blobs to keep using the `EObjectSelection` class.

CREATE / MODIFY SELECTION

You can use the EObjectSelection `Add` and `Remove` methods to:

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

These actions can be cascaded and combined at will in a single selection.

CLEAR SELECTION

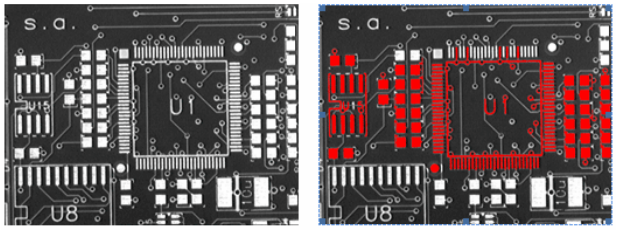
You can clear a previous selection using `EObjectSelection::.Clear`.

SORT SELECTION

You can sort the elements of a selection according to any of their features.

EXAMPLE

In this example, we select objects in the middle band of an image, with a surface >100 pixels.



Source image, and selection of objects

1. **Declare a new** `ECodedImage2` object.
2. **Declare an** `EImageEncoder` object and, if applicable, select and setup the appropriate segmenter and choose the appropriate layer(s) to encode.
3. **Encode the source image**.
4. **Create a selection of objects.** Create an instance of the `EObjectSelection` class and add objects to this selection, for instance through `EObjectSelection::.AddObjects`.
5. **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.
6. **Remove the objects based on their position** using `EObjectSelection::.RemoveUsingFloatFeature`. For details, see also "Working at the Run Level".
7. **Sort the selected objects** using `EObjectSelection::.Sort`.
8. **Access the selected objects.**

Advanced Features

Computable Features

Methods prefixed with **Get** indicate a lazy evaluation: the result is computed on the first invocation of the method and cached.

Methods prefixed with **Compute** indicate that the function is reevaluated at every invocation and the result is never cached.

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.

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
Min. enclosing rectangle (angle, center X and Y, heath, width)	ECodedElement::.MinimumEnclosingRectangle ECodedElement::.MinimumEnclosingRectangleAngle ECodedElement::.MinimumEnclosingRectangleCenter ECodedElement::.MinimumEnclosingRectangleCenterX ECodedElement::.MinimumEnclosingRectangleCenterY ECodedElement::.MinimumEnclosingRectangleHeight ECodedElement::.MinimumEnclosingRectangleWidth

FERET BOX

A feret box is a rectangle with the minimum surface rotated at a specified angle that contains all the pixels center points of an object.

- **Bounding box** is the Feret box at 0°.
- **Minimum enclosing rectangle** is the Feret box with the minimum surface across all the possible angles.
- **Width** of a **FeretBox rectangle** is 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 Feret box rectangle is 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

The object perimeter can be measured indirectly by tracing the object contour with contouring methods and counting the pixels.

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

Note. Formulas (N = area):

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

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

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

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

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

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

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

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

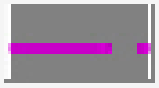
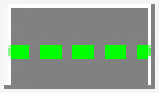
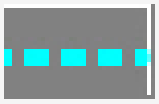
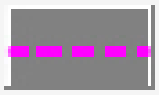
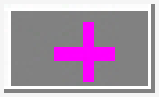
CONVEX HULL

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`.

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	Graphic
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, removing the need to access the feature through an **enum**.

Draw Coded Elements

Once an image has been encoded, the coded elements (object or hole) are accessible through the abstract class `ECodedElement` and a large set of methods:

TO DRAW CODED ELEMENTS

1. **Declare a new** `ECodedImage2` object.
2. **Declare an** `EImageEncoder` object and, if applicable, select and setup the appropriate segmenter and choose the appropriate layer(s) to encode.
3. **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.
4. **Encode the source image**.
5. **Draw the features for each object in a layer**.
6. Read the result, which can be rounded down. A specific drawing can be created to mark the feature (for example, draw a target for a gravity center).

To render flexible masks use `ECodedElement::.RenderMask`.

The objects, holes and their features can be efficiently accessed randomly (in an index-based fashion).

Flexible Masks in EasyObject

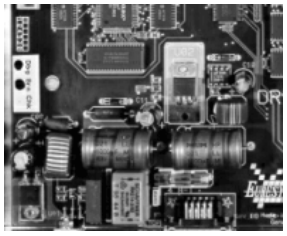
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

If an object of interest has the same gray level as other regions of the image, you can define "keep" and "ignore" areas using flexible masks and `Encode` functions.

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

- A pixel value of 0 in the flexible mask masks the corresponding source image pixel so it doesn't appear in the encoded image.
- Any other pixel value in the flexible mask causes the pixel to be encoded.

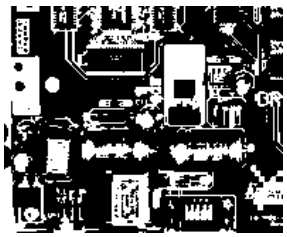
EASYOBJECT FUNCTIONS THAT CREATE FLEXIBLE MASKS



Source image

1) ECODEDIMAGE2::.RENDERMASK: FROM A LAYER OF AN ENCODED IMAGE

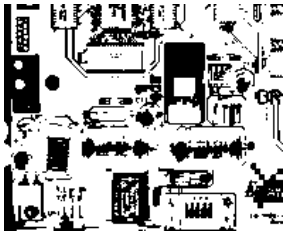
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers using minimum residue thresholding).
4. Make the mask image the desired size using `mask.SetSize(sourceImage.GetWidth(), sourceImage.GetHeight())`.
5. Exploit the flexible mask as an argument to `ECodedImage2::.RenderMask`.



BW8 resulting image that can be used as a flexible mask

2) ECODEDELEMENT::.RENDERMASK: FROM A BLOB OR HOLE

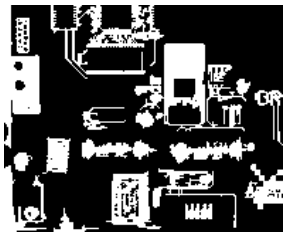
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::.RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

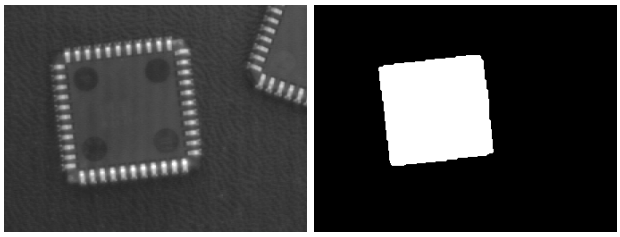
3) EOBJECTSELECTION::.RENDERMASK: FROM A SELECTION OF BLOBS

`EObjectSelection::.RenderMask` can, for example, discard small objects resulting from noise.



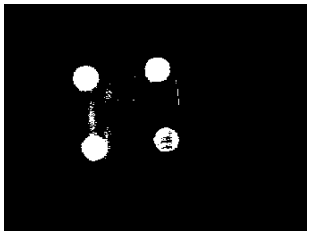
BW8 resulting image that can be used as a flexible mask

EXAMPLE: RESTRICT THE AREAS ENCODED BY EASYOBJECT



Find four circles (left) Flexible mask can isolate the central chip (right)

1. Declare a new `ECodedImage2` object.
2. Setup variables: first declare source image and flexible mask, then load them.
3. Declare an `EImageEncoder` object and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
4. Encode the source image. Encoding a layer with just the area in 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:

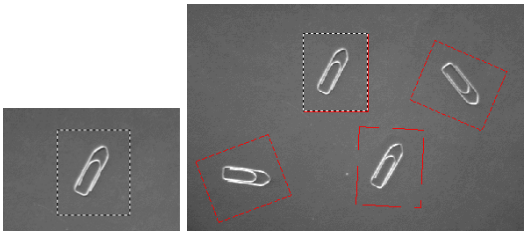


5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

EasyMatch

EasyMatch learns a pattern and finds exact matches:

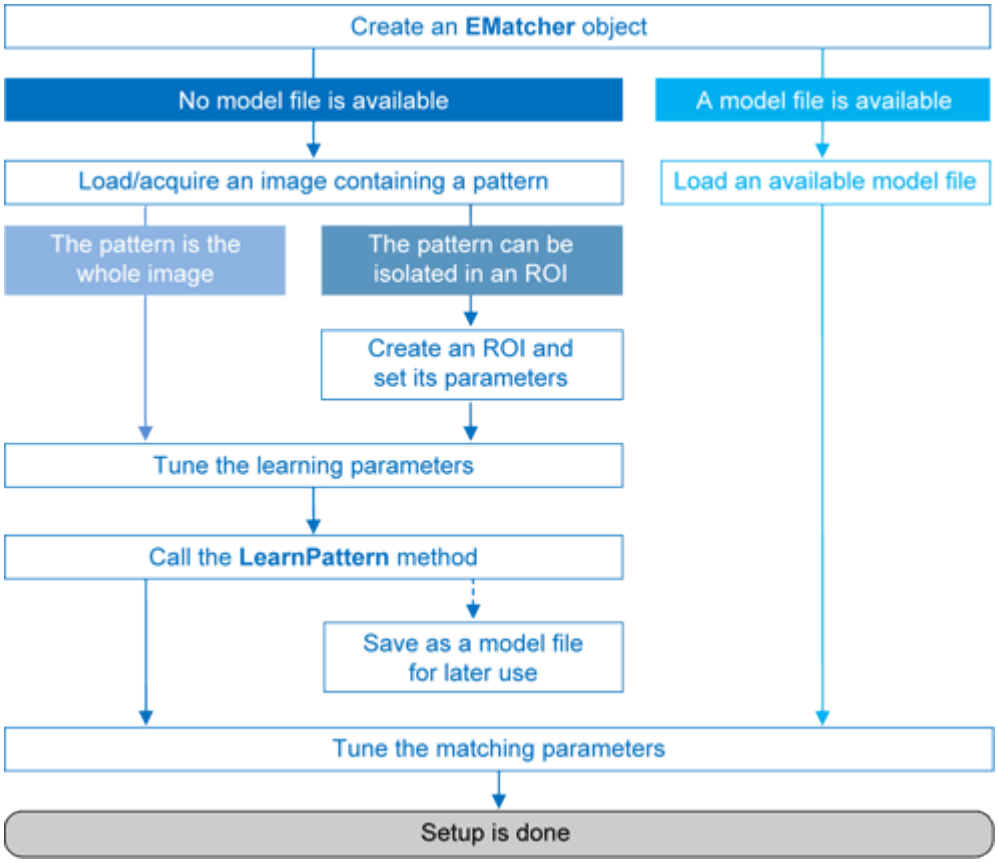
1. The pattern is learned by defining an ROI that contains the object to be matched.
   This ROI is created after iteratively learning from several images which contain the object.
2. The parameters are tuned to ensure the pattern is found reliably.
3. Images can now be searched for one or more occurrences of the pattern, which may be translated, rotated or scaled.



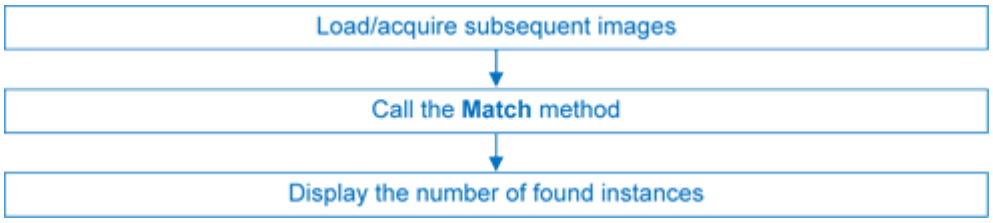
Learning and Matching a pattern

WORKFLOW

Learning workflow



Matching workflow



LEARNING PROCESS

Select an image containing the pattern/ROI to be searched for and call LearnPattern.

The resulting pattern can be saved as a model for later use. You can repeat this process to search for and save multiple patterns.

BEST PATTERN CHARACTERISTICS:

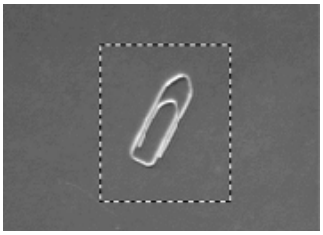
- **repeatable**, you need to know if it can translate or rotate or scale.
- **represent the object to be located.** It should:
  - keep the same appearance whatever the lighting conditions.
  - remain at a fixed location with respect to the part.
  - be rigid and not change shape.

- **exhibit good contrast in small and large scale.** It should be distinctly visible from a distance, and on a reduced image.
- **not be invariant under the degrees of freedom to be measured.** For instance, a pattern of black and white horizontal stripes cannot detect horizontal translation; a cog wheel cannot help measure large rotations.
- **have a neutral background.** If objects around the pattern in the ROI may change, this area should be neutralized by means of "don't care" pixels or a mask.
- **have contrasted margin around the objects** so that foreground and background intensities are seen.

CUSTOMIZE PARAMETERS:

Parameters can be tuned to minimize processing time, but it still takes longer than EasyFind as the entire selected area is matched.

- `DontCareThreshold`: If don't care areas are required, the corresponding pixels must hold a value below the `DontCareThreshold`.
  If all the background can be ignored, merely adjusting the `DontCareThreshold` to the right thresholding value can do.
  Otherwise, when the don't care area is unrelated 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 sub-samples the pattern. This parameter stipulates the minimum number of pixels of the pattern image to be kept. The smaller the value, the faster the matching process, but it may give unreliable results. The default value (64) is usually a good compromise.
- `FilteringMode`: If the image has sharp gray-level transitions, it is better to choose a low-pass kernel instead of the usual uniform kernel.



Learning a pattern

MATCHING PROCESS

For each new image, one or more occurrences of the pattern is searched for, allowing it to translate, rotate or scale, using a single function call:

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

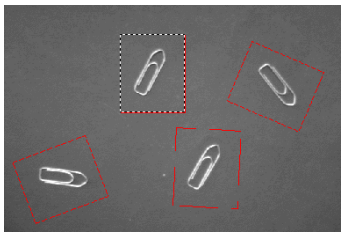
You can set these parameters:

- Rotation range: `MinAngle`, `MaxAngle`.
- Scaling range: `MinScale`, `MaxScale`.
- Anisotropic scaling range: `MinScaleX`, `MaxScaleX`, `MinScaleY`, `MaxScaleY`.

The following functions return the result of the matching:

- `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. 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.



Matching a pattern

ADVANCED FEATURES

The best way to speed up this process is to minimize rotation and scaling, and limit the number of occurrences searched for.

Learning time:

- Optimize number of searches: Searching all positions takes too long, so a sequence of searches is performed at various scales (reductions). The coarsest reduction is quick and approximate. Subsequent reductions work in a close neighborhood to improve location, drastically reducing the number of positions to be tried. The location accuracy is given by 2^K, where K is the reduction number.
- `MinReducedArea`. Indicates how small the pattern can be made for rough location.

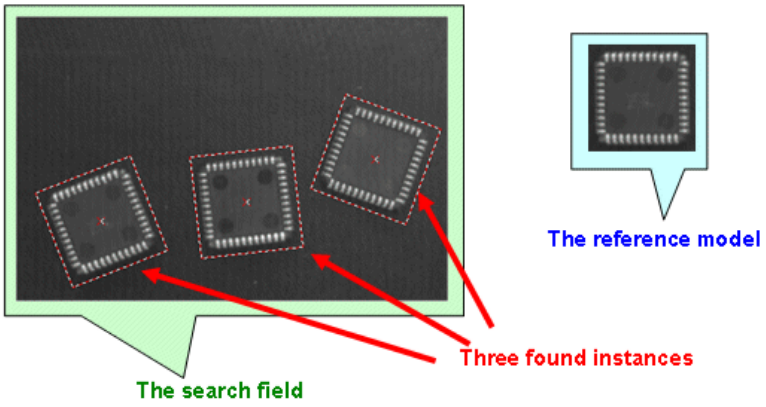
Matching time:

- Correlation mode (way to compare the pattern and the image): `CorrelationMode`. **Can be standard, offset-normalized, gain-normalized and fully normalized:** the correlation is computed on continuous tone values. Normalization copes with variable light conditions, automatically adjusting the contrast and/or intensity of the pattern before comparison.
- Contrast mode (way to deal with contrast inversions): `ContrastMode`. Lighting effects can cause an object to appear with inverted contrast, you can choose whether to keep inverted instances or not, and whether to match positive occurrences only, negative occurrences only or both.

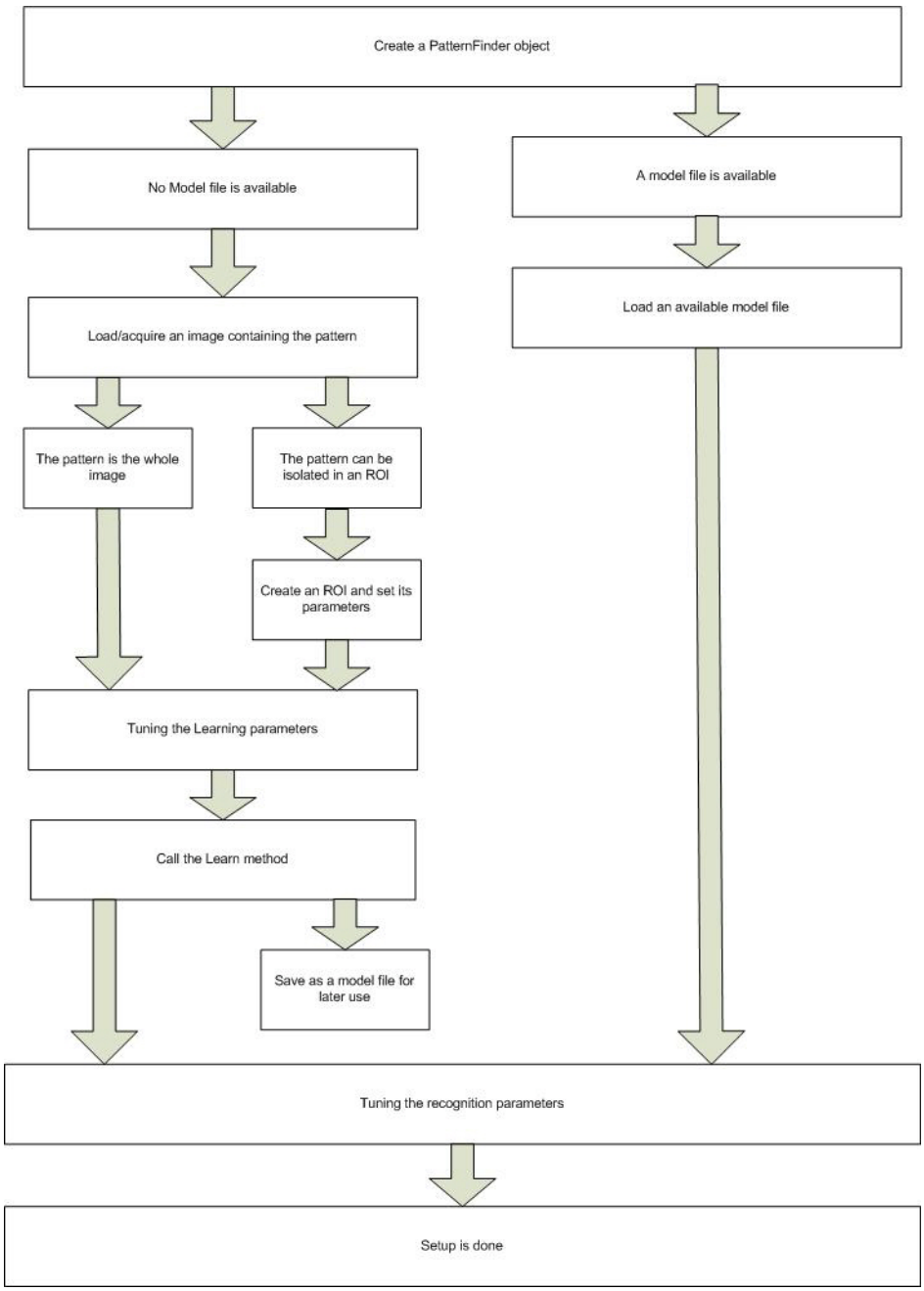
- Maximum positions (expected number of matches): `MaxPositions`, `MaxInitialPositions`. You can compel EasyMatch to consider more instances than needed at the coarse stage using the `MaxInitialPositions` parameter (this number is progressively reduced to reach `MaxPositions` in the final stage).
- Minimum score (under which match is considered as false and is discarded): `MinScore`, `InitialMinScore`.
- Sub-pixel accuracy: `Interpolate`. 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 of a pixel. Lower precision can be enforced. One tenth-of-a-pixel accuracy can be achieved.
- Number of reduction steps: `FinalReduction`. Can speed up matching when coarse location is sufficient, range [0...NumReductions-1].
- Non-square pixels: `GetPixelDimensions`, `SetPixelDimensions`. When images are acquired with non-square pixels, rotated objects appear skewed. Taking the pixel aspect ratio into account can compensate for this effect.
- "Don't care" pixels (ignored for correlation score) below the `DontCareThreshold` value. When the pattern is inscribed in a rectangular ROI, some parts of the ROI can be ignored by setting the pixels values below a threshold level. The same feature can be used if parts of the template change from sample to sample.

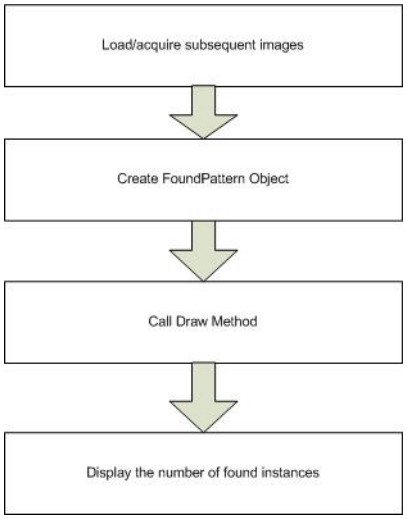
EasyFind

EasyFind learns a reference model from a pattern, which is used to find similar patterns in other images and retieve information about these instances.
It is quick and robust, and very tolerant of noise, blur, occlusion, missing parts and changes in illumination.



WORKFLOW

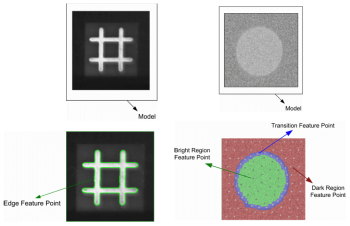

Create a PatternFinder object
No Model file is available
A model file is available
Load/acquire an image containing the pattern
Load an available model file
The pattern is the whole image
The pattern can be isolated in an ROI
Create an ROI and set its parameters
Tuning the Learning parameters
Call the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



FEATURE POINTS DEFINITION

A feature point is a pair of coordinates (X, Y) and a type (Edge, Transition or Region).

EasyFind uses feature points to find instances in a search field.

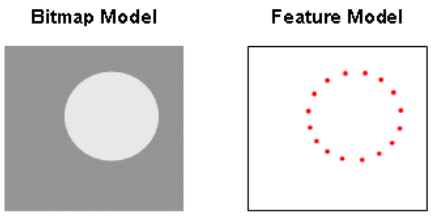


- **Edge feature points**: an abrupt change of gray level between two regions indicates an edge at this location in the search fields.
- **Transition feature points**: a smooth change of gray level between two regions indicates a transition area in their neighborhood (represented by dots in the blue area of the example above). The size of the neighborhood can be modified.
- **Region feature points**: identify 2 regions of roughly uniform gray levels:
  - dark region (represented by a family of dots in the red area of the example above),
  - bright region (represented by a family of dots in the green area).

LEARNING PROCESS

EasyFind supports various pattern types (Consistent edges, Thin structures, or Contrasting regions).
During the learning process, EasyFind computes for itself a feature model which is a set of all extracted feature points from a bitmap representation of the pattern.

EasyFind only needs this feature model to start the finding function, but you can create your own optimal model.



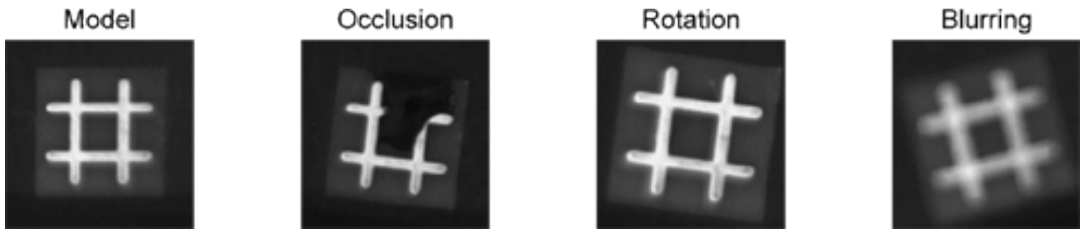
CREATE THE OPTIMAL MODEL

The optimal model depends on the type of pattern being searched for.

Consistent Edges

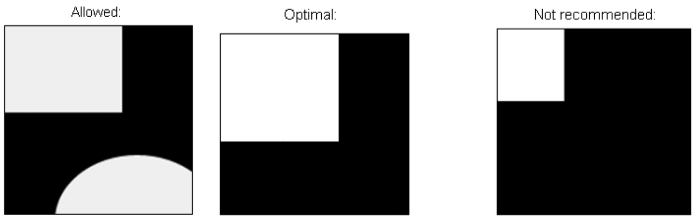
Models must be well contrasted with sharp edges. They should be substantially different from the rest of the expected search fields. Can be scaled or rotated, very robust to: blurring, noise, occlusion, illumination variation (point-by-point scores improves robustness and computation time of the finding phase).
Good for models with consistent edges, sharp contrast transitions, regions delineated by well defined edges that are in approximately the same place for each instance in all search fields.
Choose the similar points with `EPatternFinder::.ContrastMode` property:

- PointByPointNormal: if points share the same contrast polarity.
- PointbyPointInverse: if points exhibit opposite contrast polarity.
- PointByPointAny: regardless of their respective contrast polarity.

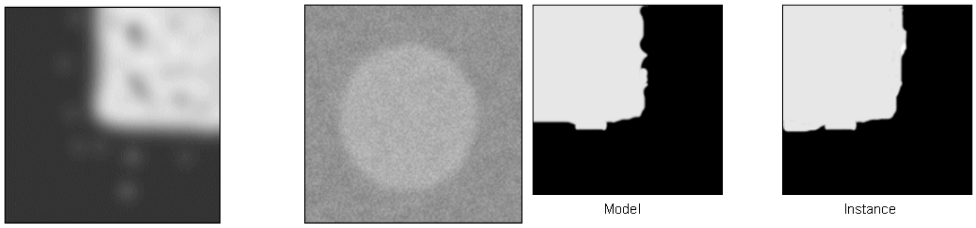


Contrasting Regions (defined by region feature points and transition feature points)

Ideally, models should be 50% bright area and 50% dark area. There can be more than one dark and/or bright region.

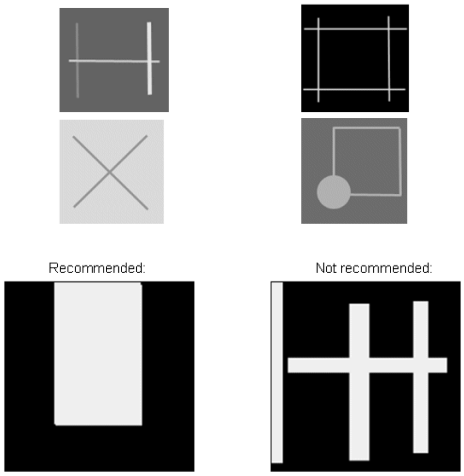


Cannot be scaled or rotated, robust to: blurring, noise, illumination variation (but not occlusion).
Good for models with inconsistent transitions or edges, or with several regions delimited by transitions or edges.



Thin Structures (defined by edge feature points)

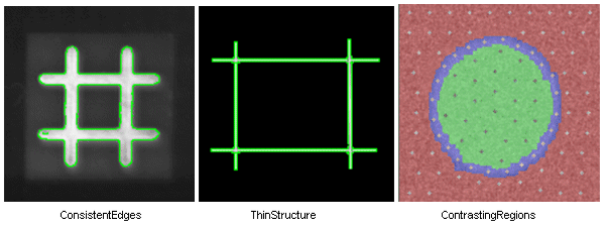
Can be scaled or rotated, robust to: blurring, noise, occlusion, illumination variation. Edges must be consistent between thin elements and regions, and the contrast should be the same for each thin element.
Good for models containing thin elements.



CHECK THE LEARNED MODEL IS CORRECT

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

On the examples, edge feature points appear as green points for Consistent Edges and Thin Structures, and crosses for Contrasting Regions.



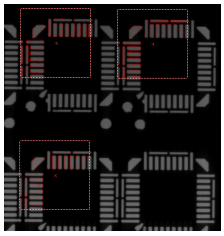
FINDING PROCESS

You can optimize the finding process by setting and saving some parameters in a configuration file. You simply load this file and run to see what EasyFind has found in the reported information.

MAXIMUM NUMBER OF EXPECTED INSTANCES

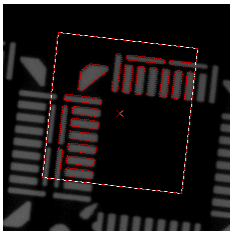
Set the maximum number of instances that EasyFind should return. In this example the number

was three.



ANGLES AND SCALES (THIN STRUCTURE AND CONSISTENT EDGES PATTERN TYPES)

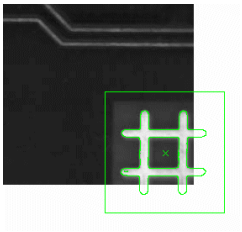
Ranges have a bias and a tolerance. For instance, for an angle bias of 20° and an angle tolerance of 5°, EasyFind returns instances with an angle between 15° and 25° with respect to the learned model (20° ± 5°).



ADVANCED FEATURES

FIND PARTIAL PATTERNS

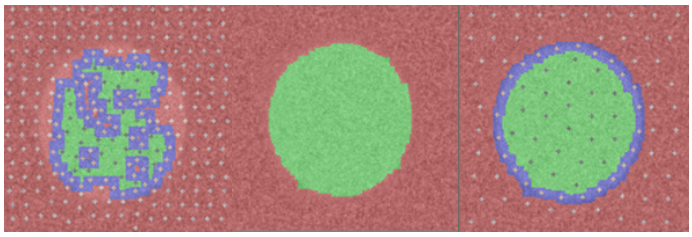
EasyFind can locate instances of thin structures and consistent edges that are partially out of the search field, if the extension of search field is set to > 0 pixels.



TUNE PARAMETERS

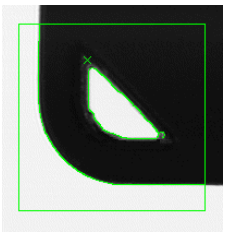
These parameters can be tuned for all models:

- Adjust the Light Balance using model drawing to preview the model so that it fits the useful parts of the pattern, then learn the model again.



1: Bad extract. 2: Adjust light balance. 3: Good extract.

- Set the gray-level threshold (this overrides light balancing) and learn the model again.
- Move the pivot to a specific place in the model like a corner or a hole. The pivot is the location returned by EasyFind when it finds an instance (the center by default).

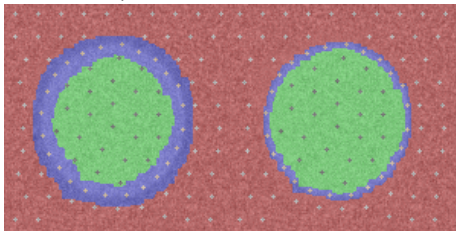


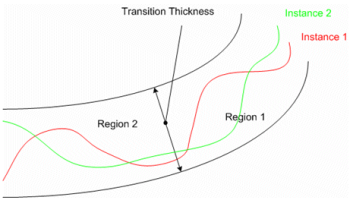
Thin Structures may benefit from tuning these parameters:

- Automatic (thin elements with the same contrast between them and their neighboring regions)
- Thin elements darker than the neighborhood
- Thin elements brighter than the neighborhood

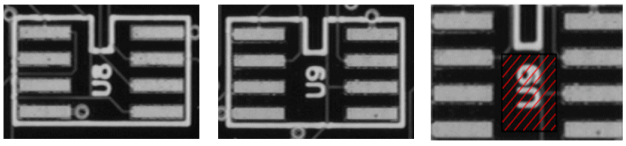
Contrasting regions may benefit from tuning these parameters:

- Transition Thickness: Transition feature points lie in the *transition band* (blue area in the examples below) which is where the transition occurs. The transition thickness parameter should allow the borders of different instances to stay within the transition band, so it is recommended to make it equal to the biggest variation among instances. Two examples of thickness, and the recommended thickness are below:



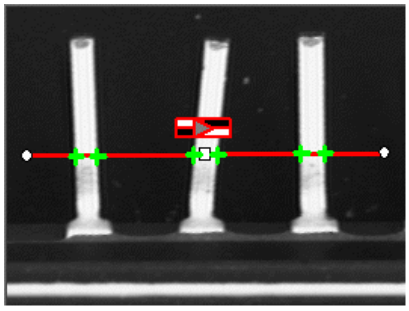


- **Ignored areas**. Zero values indicate ignored areas. 255 values indicate areas taken into account. For example: If the text in the center of the model differs from the instance, you can indicate that EasyFind must not extract feature points from this part of the model.



EasyGauge

EasyGauge library controls dimensions. It accurately determines position, orientation, curvature and size of parts. It can interact graphically to place and size gauges, combine them in grouped hierarchies, and store and retrieve them with all their parameters.



WORKFLOW

The gauge model can be built programatically or in a graphical editor, then "played" in the final application.
Chose the workflow that matches the complexity of your model and the accuracy required: uncalibrated, calibrated or grouped.

UNCALIBRATED GAUGING: FOR A SIMPLE MODEL

EasyGauge basic use is straightforward.

1. Create a gauge object that corresponds to the required measurement.
2. Change the parameters whose default values are not appropriate.
3. Invoke the desired measurement function.
4. Read the resulting position parameters.

Uncalibrated gauging is easy to implement but has several drawbacks:

- measurements are performed in pixels, not millimeters.
- measurement models are not portable: gauge positions and sizes must be reworked if viewing conditions change.
- optical distortion or perspective causes inaccurate measurements.

CALIBRATED GAUGING: FOR ONE OR TWO SIMPLE MEASUREMENT SITES

Calibrated gauging is more accurate, and measures the inspected parts independently of the viewing conditions.
All measurements are taken in the calibrated units, with any distortion implicitly compensated.
Refer to Calibration to learn how to master field-of-view calibration.

1. Create a calibrator object.
2. Place it on the inspected scene.
3. Adjust calibration parameters.
4. Attach a gauge.

COMPLEX GAUGING

Gauges can be grouped (see Gauge Manipulation Processes) and attached to another item:

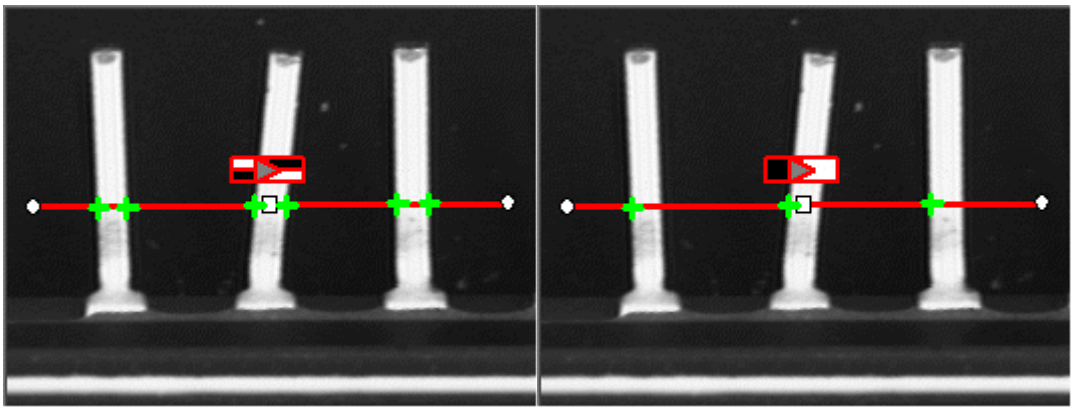
- **Attaching gauges to an** `EFrameShape` object moves the gauges with the frame (translation and/or rotation), the application program must adjust the frame position to track the inspected part.
- **Attaching gauges to another gauge** moves them according to the measured position of the supporting gauge. For example, if gauges are attached to a common rectangle gauge that is detecting the outline of a part, all gauges automatically track the part when the rectangle outline is fitted.

If using several measurement sites, you can save the complete model, with calibration modes, coefficients, and attached gauges, in a single file.

GAUGE DEFINITIONS

POINT GAUGE

You can select the most relevant transition points along a line segment probe that crosses one or several objects edges. Crosswise and lengthwise filtering can be activated for noise reduction.



Point location. Contrast-based selection

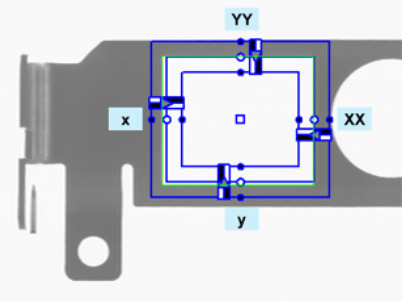
RECTANGLE GAUGE

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, and can be set to active or inactive with the `ActiveEdges` property. When a side is active:

- setting the value of a parameter only applies to the currently active sides1.
- 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 to set different parameters for different sides, and measure parallel sides or a corner point instead of the whole rectangle. The four sides are denoted by letters "x", "y", "XX" and "YY" respectively.



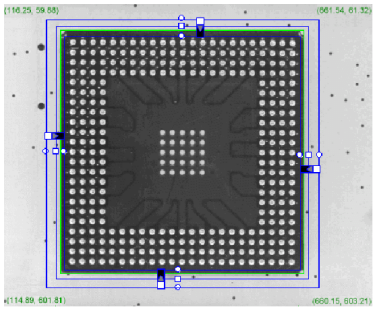
Naming conventions for the sides of a rectangle gauge

Usage

Define and position the gauge, then use `Measure` to fit the lines.
To obtain the rectangle properties, set `ActualShape` to **TRUE**to return the fitted line (**TRUE** value) (default is **FALSE**).

Alternatively, `MeasuredRectangle` provides the results as an `ERectangle` object.

For instance, you can accurately locate the four corners (landmarks) of a rectangle using a rectangle fitting gauge.



Locating a rectangle's corners

WEDGE GAUGE

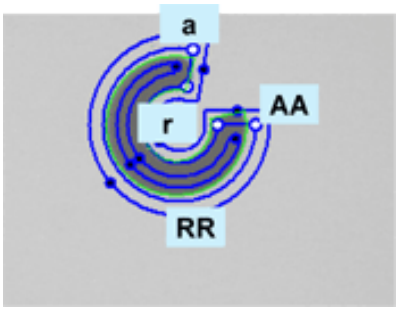
The placement of a wedge gauge is defined by its nominal position (given by the coordinates of its center), its nominal inner and outer radius (inner and outer diameter), its breadth (difference between radii), the angular position from where it extents and its angular amplitude.

The `Set` member can distinguish between a full ring, a sector of a ring and a disk.

Each side of a wedge can have its own transition detection parameters and can be set to active or inactive with 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.

So different sides can have different parameters, and you can measure 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

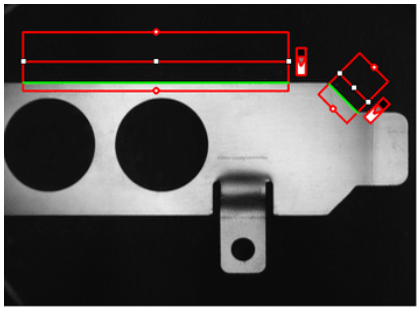
Usage

Define and position the gauge, then use `Measure` to fit the lines.
To obtain the wedge properties, set the `ActualShape` property to **TRUE** to return the fitted line (instead of the nominal line position **FALSE**, default).

Alternatively,`MeasuredWedge` provides the results as an `EWedge` object.

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. To constrain the line slope value, set `Angle` and `KnownAngle`.



Line fitting

Usage

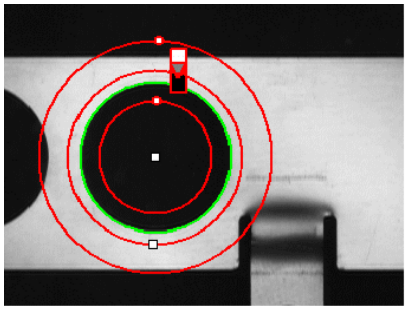
Define and position the gauge, then use `Measure` to fit the lines. To obtain the line properties, set the `ActualShape` property to **TRUE**to return the fitted line (**TRUE** value) (instead of the nominal line position **FALSE** value, default).

Alternatively, `MeasuredLine` provides the results as an `ELine` object.

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.

The Set member can distinguish between a full circle and an arc (the arc amplitude must be specified).



Circle fitting

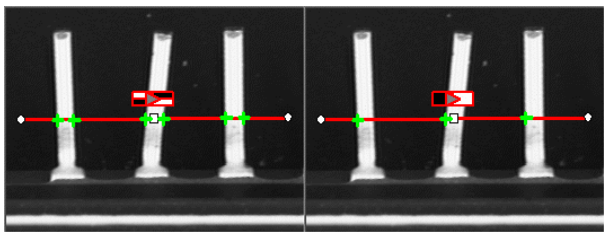
Usage

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.

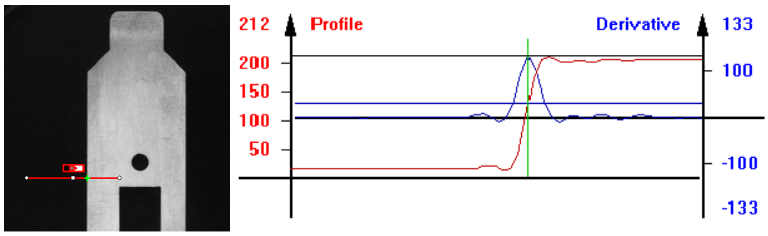
FIND TRANSITION POINTS USING PEAK ANALYSIS

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

POINT LOCATION PRINCIPLE



Point location principle (left) and S-shaped curve and its derivative (right)

On a linear profile extracted from an image, an edge appears as a transition from dark to light (or vice versa). When plotting pixel values along the gauge, this transition appears as an 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.

EasyGauge extracts the pixel values along a profile (red curve) then uses peak analysis to determine the transition location. All the pixel values in the peak area[1] are used to compute the transition location.

- Sub-pixel accuracy is only possible if the transition is surrounded by almost uniform regions of at least 2 pixels wide.
- BWB[2] transitions have an increasing profile curve and the peak takes positive values. Otherwise, the curve decreases and the peak extends negatively.
- You cannot normally detect peaks using the default threshold value (20) as BWB or WBW transitions base the peak analysis on the gray level profile along the EPointGauge (or sample path) and not its first derivative.

`EPointGauge` contains all point measurement parameters, with default values that detect reasonably contrasted edges.

EPointGauge parameters

`Center`: Nominal point position (will normally be different before and after measurement).
`Tolerance`: Tolerance value and gauge orientations.
`TransitionType`, `TransitionChoice`, `TransitionIndex`: Peak selection strategies.
`Threshold`: Noise immunity.
`MinAmplitude`, `MinArea`: Peak strength.
`Thickness`, `Smoothing`: Local filter widths.
`RectangularSamplingArea` Sets sampling area (rectangular by default) to transverse filtering mode.
`Measure`: Measures the object.
- In single transition mode, `Valid` returns True when an appropriate point was found. To obtain measurement results, set `ActualShape` to True so that `Center` returns the located point. (False default value returns nominal point position).
- In multiple transition mode, `NumMeasuredPoints` returns the number of points found,

[1]Area between the derivative curve and a horizontal user-defined threshold level

[2]Black / White / Black

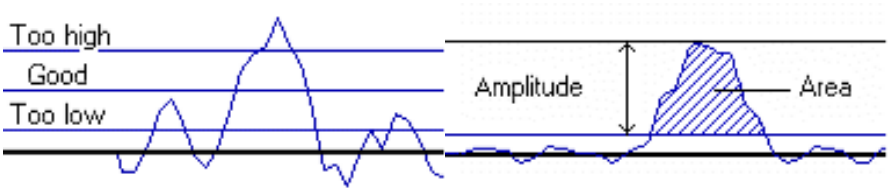
GetMeasuredPoint returns an EPoint object which contains located point information. An integer index between 0 and GetNumMeasuredPoints-1 must be passed.
GetMeasuredPeak: Returns EPeak containing the peak's Area and Amplitude, and the delimiting coordinates along the probe segment (Start, Length and Center values).

SELECT PEAKS TO IMPROVE EDGE PRECISION

The threshold level is very important:

- Too high can cause significant peaks to be missed, and insufficient pixel values to achieve good precision.
- Too low can cause false peaks because of noise.

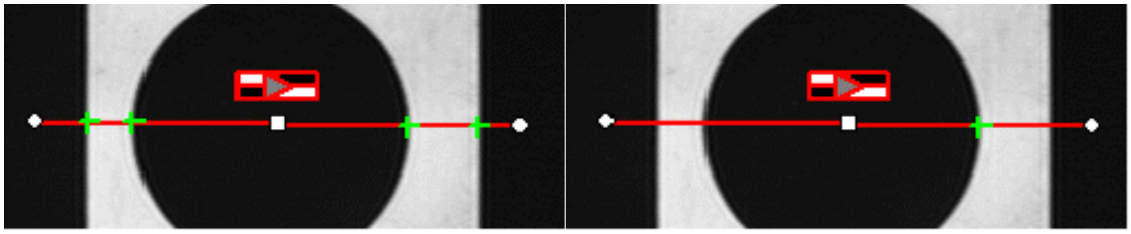
To resolve this dilemma, the EasyGauge peak selection mechanism can reject low contrast or false edges: transition strength is measured by peak *amplitude* and *area*. Every edge measurement determines peak amplitude and area. If either value falls below the minimum amplitude or minimum area, the peak is disregarded and no point is assumed at that location.



Threshold level selection (left) and Peak amplitude and area (right)

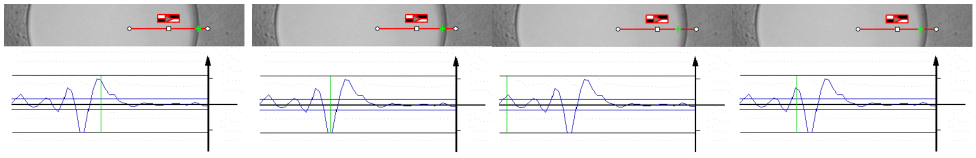
Multiple versus single transition

EasyGauge can measure several edge points in a single go and retrieve all results afterwards while in *multiple transition mode*.



Multiple transition (left) versus single transition (right)

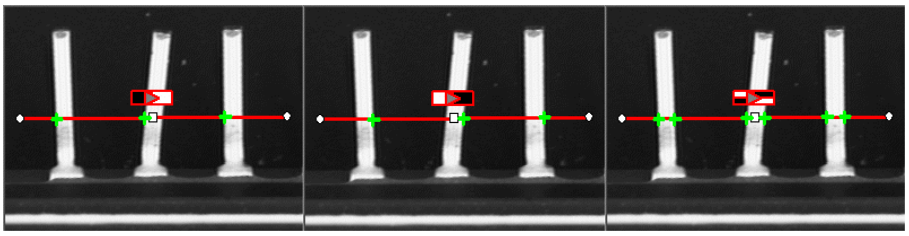
You can select the single most relevant transition based on 4 criteria: the highest peak, the peak with the largest area, the peak closest to the gauge center, or the N-th peak encountered starting from one tip of the gauge.



Best area (first image) and best amplitude choices (2nd image), closest (3rd image) and 3rd from the start (4th image)

Positive or negative peak selection

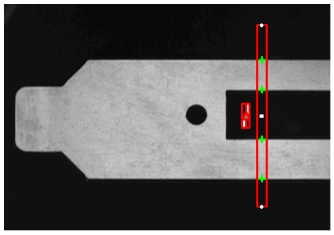
Peak selection can also be refined by choosing the transition polarity: White to Black or Black to White (i.e. positive or negative peak), or indifferent.



Black to white, white to black or indifferent polarities

Pre-filtering

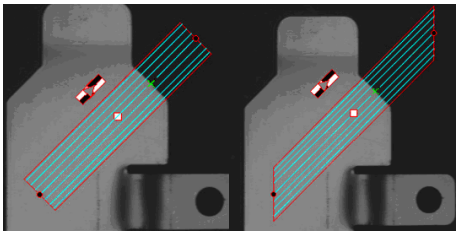
Pre-filtering the image locally can reduce noise effects.
Transverse (lengthwise) filtering averages several parallel lines when sampling the image.
Longitudinal (crosswise) uniform filtering can also be applied to the resulting profile curve.



Thick point gauge for filtering

Transverse Filtering

Transverse filtering places parallel line segments in either a parallelogram or a rectangle (default). This behavior can be toggled.
Parallelogram mode is faster than rectangular if the angle is close to 0° or 90°, or thickness is less than 5. If thickness=1, no difference exists between the two modes.
thickness determines the number of parallel lines.
sampling area is the smallest region containing all the parallel line segments.



Rectangular sampling area (left) and Parallelogram sampling area (right)

Point Probe Position

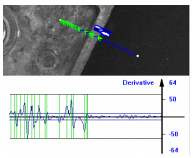
The expected **nominal** position of a point gauge is specified by its **center**, orientation **angle** with respect to the X-axis, and length **tolerance**that the point position can vary.

The results are the coordinates of the located points (the **actual** location) and the strength of the transition (amplitude and area).
Low values indicate a weak edge, possibly corresponding to an unreliable or inaccurate measurement.

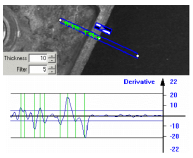
TUNING POINT MEASUREMENT PARAMETERS FOR UNCLEAR EDGES

The EasyGauge default parameters and working modes are good for clear edges. More complex situations may need parameter tuning.

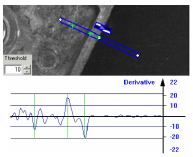
1. Set the gauge point location and tolerance.
The center position and orientation are easy to decide based on a sample image or on coordinate considerations. The tolerance depends on the edge position variations. A larger tolerance increases the likelihood of hitting an edge, but it may be a false edge or extraneous feature.



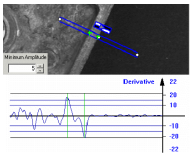
2. Decide whether noise reduction is required. Lay the gauge over the desired location and observe the profile curve and its derivative (play with the filtering parameters while looking at the plotted curve). The curve regularity gives an indication of the spread of the gray-level values.
When these coefficients are set, the gauge-level profile will not change anymore.



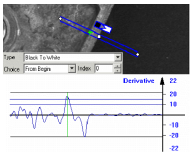
3. Set the threshold value to be low enough for useful parts of the peaks to cover enough pixels (to achieve better sub-pixel accuracy), but not lower than the ambient image noise.



4. Remove weak or false edges using the list of peak amplitudes and areas. Plotting these values along with good and extraneous peaks can help find appropriate peak rejection limits.

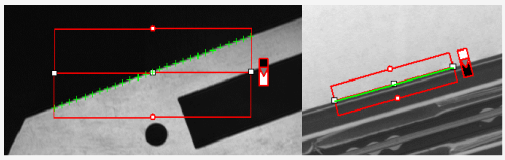
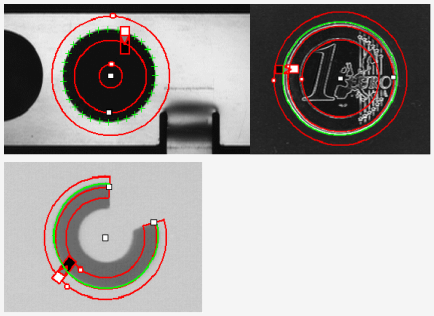
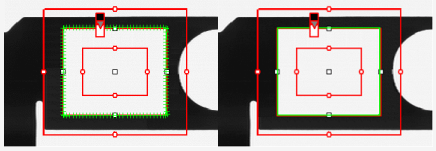
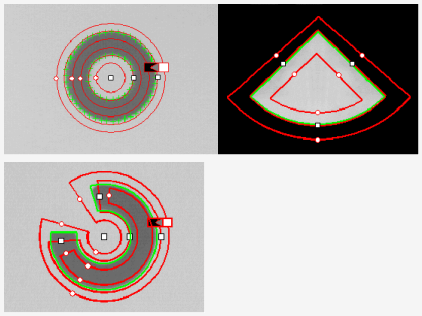


5. Choose whether all transition points are needed or just the most relevant. If all are required, they can be queried one after another. Otherwise, a point selection strategy should be chosen based on strength, order or transition polarity (black to white and/or conversely).



FIND SHAPES USING GEOMETRIC MODELS

`ELineGauge`, `ECircleGauge`, `ERectangleGauge`, or `EWedgeGauge` predefined geometric models can be fit over the edges of an object. The targeted edge must be defined, and points sampled along it at regularly spaced point measurement gauges. Model fitting in the least square sense can be applied.

Line: Measures position and orientation of straight edges.	
Circle: Measures position and curvature of a circle or arc.	
Rectangle: Measures position, orientation and size of a rectangle.	
Wedge: Measures position, orientation and size of a ring/ disk sector / curvilinear rectangle.	

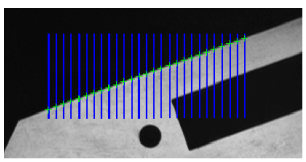
All gauge types share these common features:

Point sampling

Point gauges are placed along the edges and point measurement carried out at regularly spaced spots, which can be adjusted differently per side in rectangle and wedge gauges. All point measurement parameters and operating modes are available.

`SamplingStep` sets the spacing of point location gauges along the model.

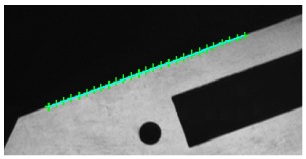
`NumSamples` returns the number of points sampled during the model fitting operation.



Sampling paths and sampled points

Model fitting

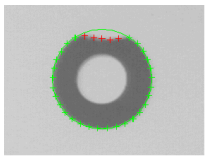
The model is adjusted to minimize error residue and provide the best edge parameter estimates. Rectangles and wedges have parallelism and concentricity constraints. Image shows sampled points and fitted line.



Outlier rejection

After model fitting, some points will be too far away from the fitted model and may harm location accuracy. EasyGauge can tag them as outliers to be ignored using the `FilteringThreshold` property.

The outlier elimination process can be repeated several times using `NumFilteringPasses`.
The number of valid sample points remaining after a model fitting operation is kept in `NumValidSamples`.
The average distance of these points to the fitted model is returned by `AverageDistance`.



GAUGE MANIPULATION: DRAW, DRAG, PLOT, GROUP

EasyGauge provides means to graphically interact with 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.

DRAW

`Draw` gives a graphical representation of a gauge. Drawing is done with the current pen in the device context associated to the desired window. Depending on the operation, handles may be displayed.

DRAG

An operator can drag a gauge interactively over an image. Several dragging handles are available.

- `HitTest` determines when the mouse cursor is over a handle. When it is, the cursor shape should be changed for feedback, and a drag can take place.
- `Drag` moves the handle and the corresponding gauge accordingly.

PLOT

EasyGauge can `Plot` gray-level values along the sampled paths and/or its derivative - useful for parameter tuning.
Point measurement gauges can plot after calling `Measure`.
Model fitting gauges can plot after calling `MeasureSample` with an index argument that lies between **0** and **GetNumSamples-1** (included).
To view the corresponding sampling path, use method `Draw` with mode **EDrawingMode_SampledPath**.

GROUP

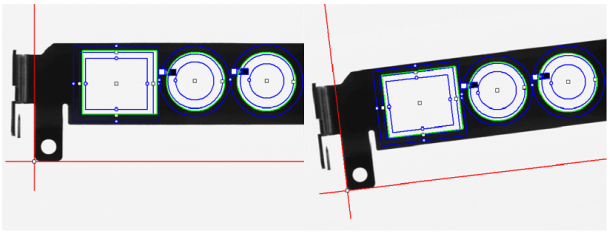
Measurement gauges can be grouped (their relative placement remains fixed) to form a dedicated tool that can be moved (translated and rotated) to follow the movement of inspected

items / probes before computing measurements.

`Attach` associates a gauge to a mother gauge or EFrameShape object.

`NumDaughters`, `GetDaughter`, or `Mother` retrieves information relative to attached daughters or mother.

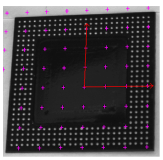
`Detach`, `DetachDaughters` dissociates the gauge or daughters from the mother.



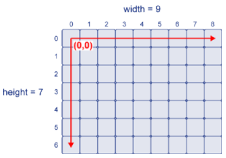
CALIBRATION AND TRANSFORMATION

Field-of-view calibration

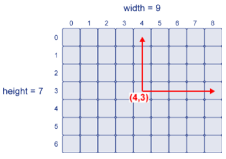
Calibration establishes the relationship between real-world point coordinates and image pixels. A simple calibration model computes faster, a repeatable part position is easier to locate.



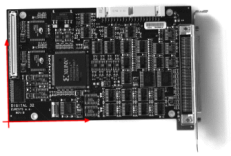
The Raw sensor coordinate system starts from upper left and extends rightwards and downwards.
The range of abscissae is **0** to **width-1** and the range of ordinates is **0** to **height-1** where integer coordinate values correspond to pixel centers.



The Centered sensor coordinate system starts at the center ([width-1]/2, [height-1]/2 in the Raw system) and extends rightwards and upwards.



The real world 3D coordinates are defined in a 2D reference frame tied to a reference plane. The origin and direction of the axis are normally aligned with major features of the inspected parts.



Before World-to-Sensor Transform

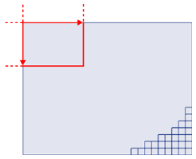
Before converting from world to sensor coordinates, sources of distortion should be eliminated:

- adjust sweep frequency or scanning speed to avoid non-square pixels.
- adjust optical alignment to minimize perspective effect. The field of view should be parallel to the sensor plane.
- use long focal distances and good quality lenses to minimize Optical distortion.
- use appropriate scale factor based on lens magnification, observation distance and focusing.

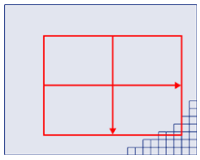
- minimize skew and translation effects by secure fixtures, and part-movement / acquisition-triggering synchronization.

Effects of World-to-Sensor Transform

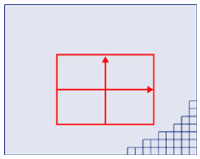
- **No calibration**. World and sensor coordinates are identical.



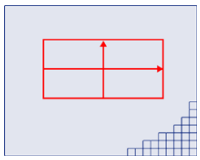
- **Translated calibration**: The coordinate origin can be moved. World coordinates correspond to pixel units.



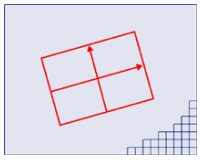
- **Isotropic scaling** (square pixels). A scale factor converts pixel values to physical measurements.



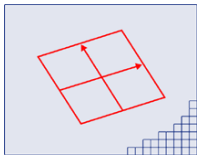
- **Anisotropic scaling** (non-square pixels). Uses two scale factors with pixel aspect ratio (X /Y) in the range [-4/3, -3/4] (or [3/4, 4/3]). Pixels are always displayed as square, so the image appears stretched.



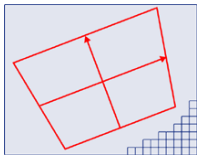
- **Scaled and skewed** (square pixels). Real-world axis aligns with rotated inspected part using translation, rotation and scaling.



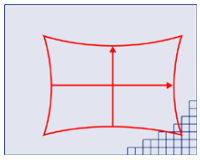
- **Scaled and skewed** (non-square pixels). Distortion is apparent. Occurs when camera scan speed does not match pixel spacing.



- **Perspective distortion** causes further away objects to look smaller; lines remain straight but angles are not preserved.



- **Optical distortion** causes cushion or barrel appearance of rectangles.



- **Combined distortions** result in a complex, non linear, transform from real-world to sensor spaces.

CALIBRATION USING EWORLDSHAPE

The EWorldShape object can calibrate the whole field of view (in given imaging conditions with

fixed camera placement and lens magnification), if the optical setup is modified. `EWorldShape` computes appropriate calibration coefficients and transforms measurement gauges that are tied to it.
It can set world-to-sensor transform parameters, perform conversions from and to either coordinate system, determine unknown calibration parameters, and save the parameters of a given transform for later reuse.

After calibration `EWorldShape` can perform coordinate transform for arbitrary points using `SensorToWorld` and `WorldToSensor` to:

- measure non-square pixels and rotated coordinate axis.
- correct perspective and optical distortion, with no performance loss.

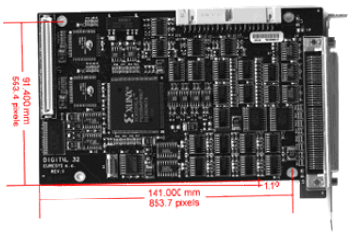
There are several ways to obtain the calibration coefficients:

ESTIMATE (FEASIBLE IF NO DISTORTION CORRECTION IS REQUIRED AND ACCURACY REQUIREMENTS ARE LOW)

To estimate the calibration coefficients either locate the limits of the field of view and divide the image resolution by the field of view size, or use the following procedure:

1. Take a picture of the part to be inspected or a calibration target (e.g. rectangle).
2. Locate feature points such as corners in the image (by the eye) and determine their coordinates in pixel units —let `(i,j)`.
3. 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.
4. For non-square pixels repeat this operation for pairs of horizontal and vertical points.

To estimate a skew angle, apply this formula 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, use `SetSensor` to adjust them and set the calibration mode, or set them individually using: `SetSensorSize`, `SetFieldSize`, `SetResolution`, `SetCenter`, `SetAngle`.

PASS A SET OF REFERENCE POINTS (LANDMARKS) TO A CALIBRATION FUNCTION

Locate at least 4 landmarks and obtain their coordinates in sensor (using image processing) and world coordinate systems (actual measurements). More landmarks give more accurate calibration.

The resulting pixels aspect ratio (X resolution/Y resolution) must be in the range [-4/3, -3/4] (or [3/4, 4/3]).

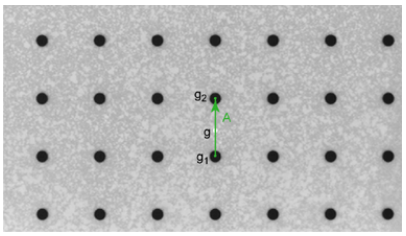
ANALYZE A CALIBRATION TARGET

A calibration target can be automatically analyzed to get an appropriate set of landmarks. It is an easy way to achieve automatic calibration, provided an appropriate procedure is available to extract the desired landmark point coordinates.

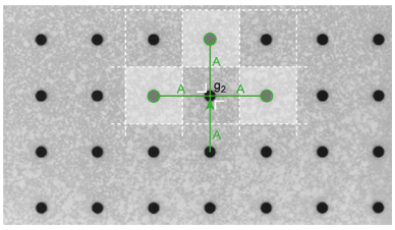
Open eVision relies on the use of a specific target holding a rectangular grid of symmetrical dots (of any shape)with no other object on the grid.

Dot Grid based calibration example

1. Grab an image of the calibration target in such a way that it covers the whole field of view (or restrict the image of view to an ROI where only dots are visible).
2. Apply 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 to reconstruct a grid to calibrate a field of view using an iterative algorithm which computes the world coordinates of each dot.
   a. The grid points nearest to the gravity center (g) of grid points are selected (g_1 and g_2) to form the first reference oriented segment, of length A.



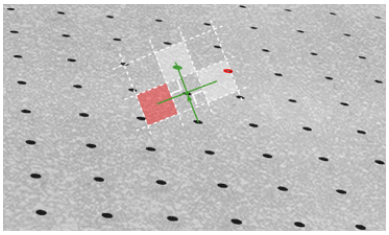
   b. Starting from the extremity of the reference segment (g_2), the algorithm determines 3 tolerance areas (white squares in the figure), in perpendicular directions. The tolerance areas are centered at a distance A (length of the reference segment) from (g_2). They are square, with a side-length of A.
   The algorithm searches for 1 neighboring point, in each of the 3 tolerance areas.
   The grid will be correctly calibrated if each tolerance area contains a neighboring point.



   c. The 3 perpendicular segments are the references of the next iterative searches. The algorithm goes back to step 2.
5. Call `Calibrate`the landmark approach.

If the grid exhibits too much distortion, grid reconstruction does not work as expected. The following errors could happen:

1. A tolerance area does not contain a neighboring point (red square in the figure).
2. A tolerance area contains more than one neighboring point.
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).



ADVANCED FEATURES

The field-of-view calibration model can be tuned using these parameters:

SENSOR WIDTH AND HEIGHT

The sensor width and sensor height give the logical image size, in pixels (always integers).

FIELD-OF-VIEW WIDTH AND HEIGHT

The field-of-view (f-o-v) width and height give the actual image 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
```

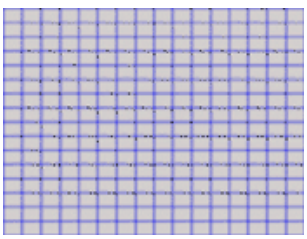
or

```
sensor width = f-o-v width * horizontal
               resolution
sensor height = f-o-v height * vertical
               resolution
```

By default pixel height is not specified, the pixels are assumed to be square (pixel width = pixel height).

`Scale`

RATIO



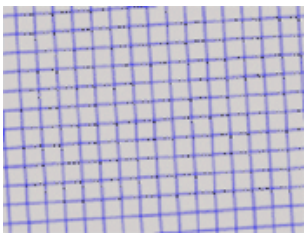
Anisotropic aspect ratio

CENTER ABSCISSA AND ORDINATE

The center abscissa (x) and ordinate (y) indicate the image origin point (world coordinates (0,0). Default is the image center.

SKEW ANGLE

The skew angle is the angle formed by the real-world reference frame (X-axis) and the image edge (horizontal). The default is no skew.

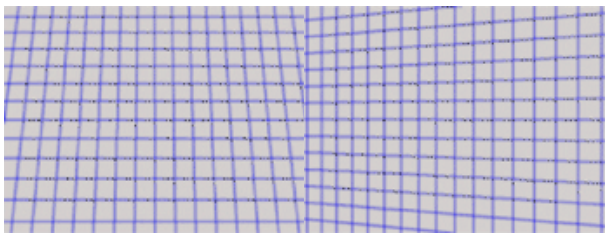


Skew angle

When the pixels are not square, the `EWorldShape` object can convert the angle between the world and sensor spaces.

X AND Y TILT ANGLES

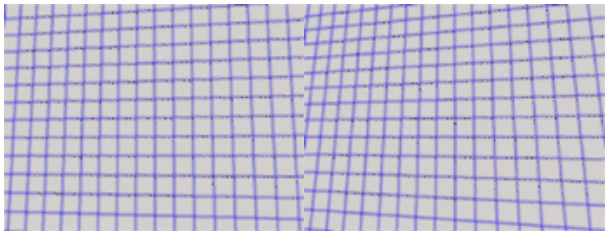
The X and Y tilt angles describe the viewing plane direction. They correspond to the required rotations around X and Y axis that bring the Z axis parallel to the optical axis.



Tilt X and tilt Y angles

PERSPECTIVE STRENGTH

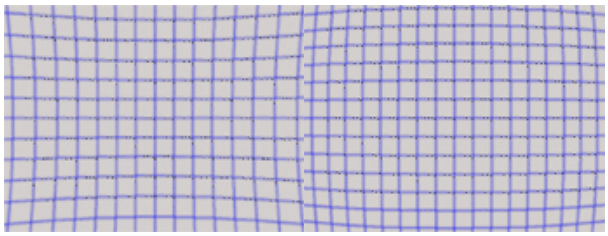
The perspective strength gives a relative measure of the perspective effect. The shorter the focal length, the larger the value.



Weak and strong perspective

DISTORTION STRENGTH

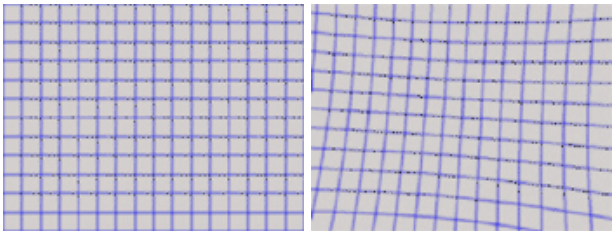
Distortion strength and `GetDistortionStrength2` give a relative measure of radial distortion in the image corners, i.e. the ratio of image diagonal length with and without distortion.



Positive and negative distortion

Calibration mode, expressed as a combination of options, can be accessed via `CalibrationModes`.

Effect of the Calibration Coefficients



No calibration coefficient: All coefficients combined.

UNWARP AN IMAGE

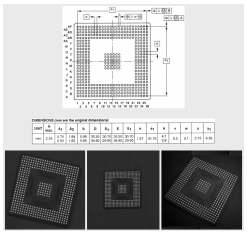
An **EWorldShape** object manages a field-of-view calibration context. Such an object is able to represent the relationship between world coordinates (physical units) and sensor coordinates (pixels), and account for the distortions inherent in the image formation process.

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 a set of known points (landmarks), or a calibration target.

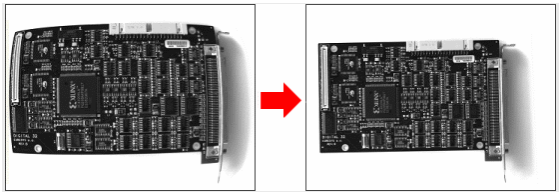
The goal of calibration is twofold:

- 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.



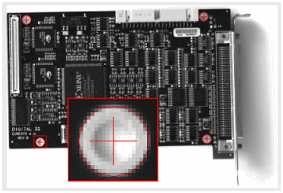
Single model versus multiple viewing conditions

- To correct some distortion related to the imaging process (perspective effect, optical aberrations, ...).



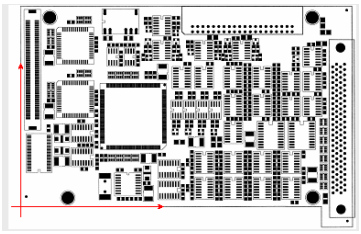
Removal of image distortion

The pixel indices in an image are usually integer numbers, but fractional values can occur when using sub-pixel methods. They are normally obtained by processing an image and locating known feature points. These values are called sensor coordinates.



Feature point in sensor space

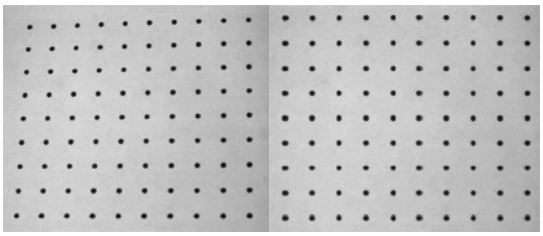
The world coordinates describe the location of points on the inspected item are expressed in an appropriate length measurement unit.
The world coordinates are actual dimensions, usually gathered from design drawings or by mechanical measurements.
They require a reference frame to be defined.



Reference frame in world space

UNWARP

Unwarp an image using `Unwarp`, `SetupUnwarp` and `UnwarpAfterSetup`.
Using a lookup table before unwarping may speed up the process.



Distorted vs. Unwarped image

EasyOCR

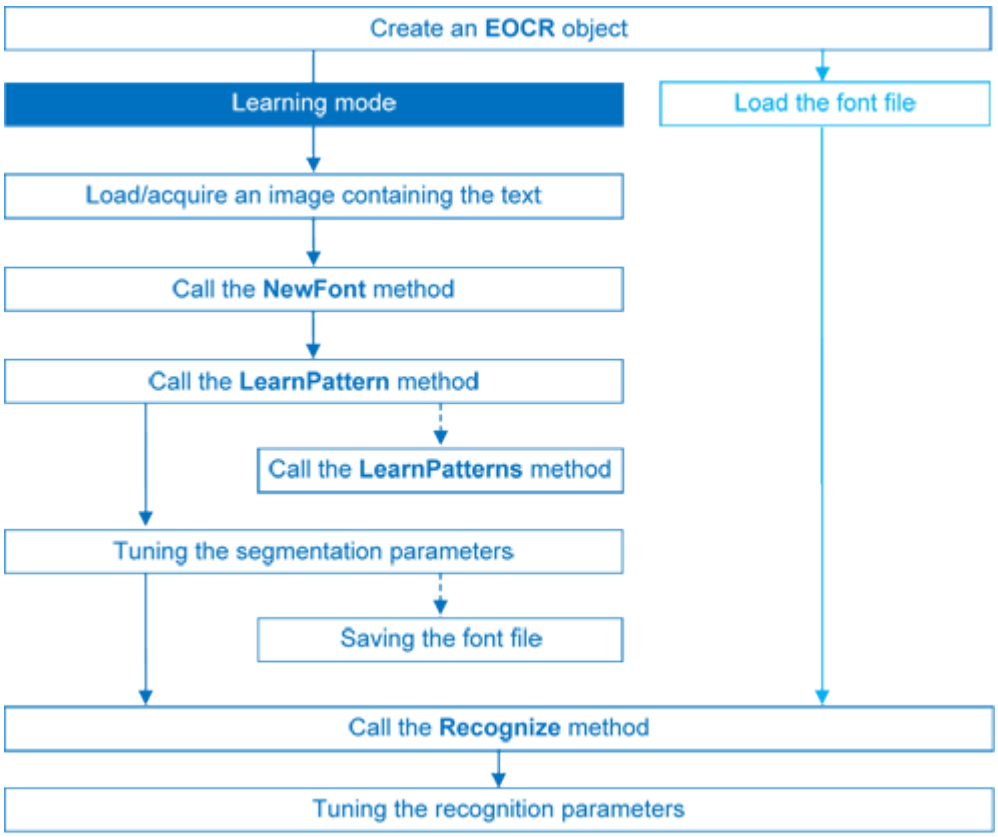
EasyOCR optical character recognition library reads short texts (such as serial numbers, part numbers and dates).

It uses font files (pre-defined OCR-A, OCR-B and Semi standard fonts, or other learned fonts) with a template matching algorithm that can recognize even badly printed, broken or connected characters of any size.

There are 4 steps to recognizing characters:

1. Raw image
2. Object segmentation
3. Character isolation
4. Character recognition

WORKFLOW



LEARNING PROCESS

You can learn characters to create font file if required.

Characters are presented one by one to EasyOCR which analyzes them and builds a database of characters called a font. Each character has a numeric code (usually its ASCII code) and belongs to a character class (which may be used in the recognition process).

Font files are created as follows:

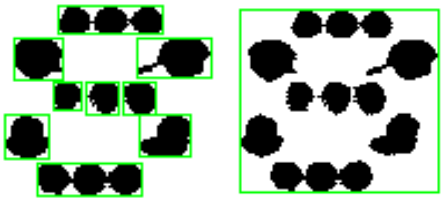
1. `NewFont` clears the current font.
2. `LearnPattern` or `LearnPatterns` adds the patterns from the source image to the font. Patterns are ordered by their index value, as assigned by the `FindAllChars` process. The patterns in a font are stored as a small array of pixels, by default 5 pixels wide and 9 pixels high. This size can be changed before learning, using parameters `PatternWidth`and `PatternHeight`.
3. `RemovePattern` removes unwanted patterns (optional).
4. `Save` writes the contents of the font to a disk file with parameter values: `NoiseArea`, `MaxCharWidth`, `MaxCharHeight`, `MinCharWidth`, `MinCharHeight`, `CharSpacing`, `TextColor`.

SEGMENTING

For learning as well as recognition, EasyOCR segments the characters, i.e. locates the characters and determines 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 to remove unwanted features (small blobs of noise, large extraneous objects, ...).

1. EasyOCR analyses the blobs to locate the characters and their bounding box, using one of two segmentation modes:

- **keep objects** mode: one blob corresponds to one character.
- **repaste objects** mode: the blobs are grouped into characters of a nominal size. This is useful when characters are broken or made up of several parts. When a blob is too large to be considered a single character, it can be split automatically using `CutLargeChars`.



Character segmentation by blob grouping

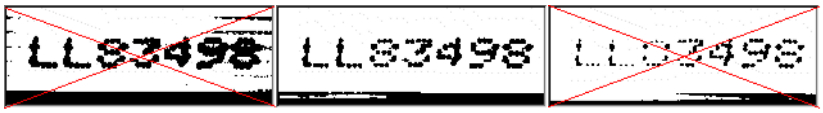
2. Filters remove very large and very small unwanted features.

3. EasyOCR processes the character image to normalize the size into a bounding box, extracts relevant features, and stores them in the font file. The patterns in a font are stored as arrays of pixels defined by `PatternWidth` and `PatternHeight` (by default 5 pixels wide and 9 pixels high).

SEGMENTATION PARAMETERS

Segmentation parameters must be the same during learning and recognition. Good segmentation improves recognition.

- The `Threshold` parameter helps separate the text from the background.
  A too high value thickens black characters on white background and may cause merging, a too small value makes parts disappear.
  If the lighting conditions are very variable, automatic thresholding is a good choice.



Too high threshold value (left), Threshold adjustment (middle), Too low threshold value (right)

- `NoiseArea`: Blob areas smaller than this value are discarded. Make sure small character features are preserved (i.e., the dot over an "i" letter).
- `MaxCharWidth`, `MaxCharHeight`: Maximum character size. If a blob does not fit in a rectangle with these dimensions, it is discarded or split into several parts using vertical cutting lines. If several blobs fit in a rectangle with these dimensions, they are grouped together.
- `MinCharWidth`, `MinCharHeight`: Minimum character size. If a blob or a group of blobs fits in a rectangle with these dimensions, it is discarded.
- `CharSpacing`: The width of the smallest gap between adjacent letters. If it is larger than `MaxCharWidth` it has no effect.
  If the gap between two characters is wider than this, they are treated as different characters. This stops thin characters being incorrectly grouped together.
- `RemoveBorder`: Blobs near image/ROI edges cannot normally be exploited for character recognition. By default, they are discarded.

RECOGNITION

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.

1. `Load`: reads a pre-recorded font from a disk file.
2. `BuildObjects`: The image is segmented into **objects** or blobs (connected components) which help find the **characters**. This step can be bypassed if the exact position of the characters is known. If the character isolation process is bypassed, you must specify the known locations of the characters: `AddChar` and `EmptyChars`.
3. `FindAllChars`: selects the objects considered as characters and sorts them from top to bottom then left to right.

4. `ReadText`: performs the matching and filters characters if the marking structure is fixed or a character set filter was provided.
   Character recognition: The characters are compared to a set of patterns, called a **font**. The best match is stretched to fit in a predefined rectangle and compared to each normalized template in the font database.
   A **Character set filter** can improve recognition reliability and run time by restricting the range of characters to be compared. For instance, if a marking always consists of two uppercase letters followed by five digits, the last of which is always even, it is possible to assign each character a class (maximum 32 classes) then set the character filter to allow the following classes at recognition time: two uppercase, four even or odd digits, one even digit.

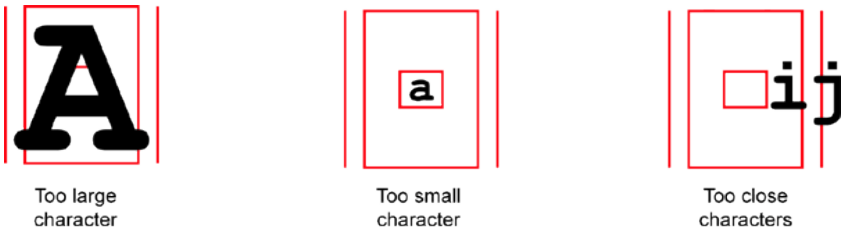
Steps 2 to 4 can be repeated at will to process other images or ROIs. The `Recognize` method can be used as well.

Additional information, such as geometric position of the detected characters, can be obtained using: `CharGetOrgX`, `CharGetOrgY`, `CharGetWidth`, `CharGetHeight`, ...

`CompareAspectRatio` makes character and font comparison sensitive to the difference between narrow and wide characters. It improves recognition when characters look like each other after size normalization.

RECOGNITION PARAMETERS

- `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`: Small characters are discarded if they are narrow **or** flat. By default they are discarded when they are both narrow **and** flat.
- `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 `MaxCharWidth`, it is discarded. When enabled, the blob is split into as many parts as necessary to fit and the amount of white space to be inserted between the split blobs is set by `RelativeSpacing`. 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

ADVANCED TUNING

These recognition parameters can be tuned to optimize recognition:

`CompareAspectRatio`: when this setting is on, EasyOCR is less tolerant of size and takes into account the measured aspect ratio. Using this mode improves the recognition when characters look similar after size normalization as it enforces the difference between narrow and wide characters.

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.

ESegmentationMode, character isolation mode defines how characters are isolated:

- **Keep objects** mode: a character is a blob; no attempt is made to group blobs, thus damaged characters cannot be handled and small features such as accents and dots may be discarded by the minimum character size criterion.
- **Repaste objects** mode: blobs are grouped to form distinct **characters** if they fit in the maximum character size and are not separated by a vertical gap, thus preserving accents and dots.

EasyOCR2

EasyOCR2 is an optical recognition library designed to read short texts such as serial numbers, expiry dates or lot codes printed on labels or on parts.

It uses an innovative segmentation method to locate the text, following a user-defined topology (number of lines, words and characters in the text). This segmentation supports text rotation up to 360 degrees, can handle non-uniform illumination, textured backgrounds, as well as dot-printed or fragmented characters..

A character type (letter, digit, symbol) can be specified for each character in the text, improving recognition rate and speed.

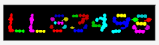
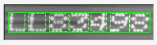
A character database is used for recognition, it can be learned from sample images or read from a TrueType font (.ttf) file.

EasyOCV

Optical Character Verification compares a geometric pattern, a *sample*, with a predefined model, a *template*, while taking into account relative displacement of the constituent parts. For example, a printed part number may be checked for: correct placement with respect to the component body, sufficient contrast, good character shapes, or absence of inking defects.

WORKFLOW

From the raw image to the final model, the model definition follows a logical sequence of steps.

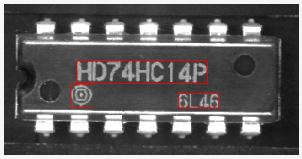
					
Raw image	Thresholded image	Free blobs	Free characters	Single text	Final model

1. Threshold the image to separate the foreground (marking) from the background.
2. Free blobs: Blob analysis is performed and the detected objects may be selected depending on their size to generate a set of candidates. (Blobs can be selected or unselected manually, to add features of unusual size or delete spurious objects.)
3. Free characters: The blobs are aggregated into boxes corresponding to the characters. (Manual correction is possible.)
4. Create single text: The characters are aggregated into boxes corresponding to the texts. (Manual correction is possible.)
5. Create final model: 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 discarded.
6. Before the template can be saved, 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 of character location with respect to their nominal position in the texts (during translation).
   - the allowed ranges of 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.

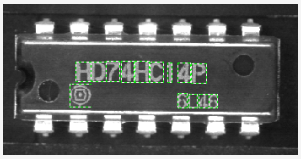
LEARNING PROCESS

MODEL STRUCTURE

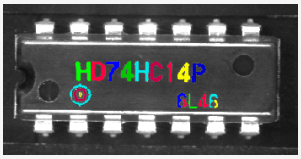
Inspection takes place in a rectangular ROI. The marking is a set of texts, made of characters, which are made of blobs. Typically there is only one text, and each character is a blob.



Texts are sets of characters



Characters are sets of blobs



Blobs are "atomic" constituents of markings

In a simple application, a ready-made template (Open eVision Studio provides a comprehensive template editor) is used in the inspection phase.

EasyOCV requires training on correctly printed marks to create a good quality template. Using a single image to create a template has drawbacks:

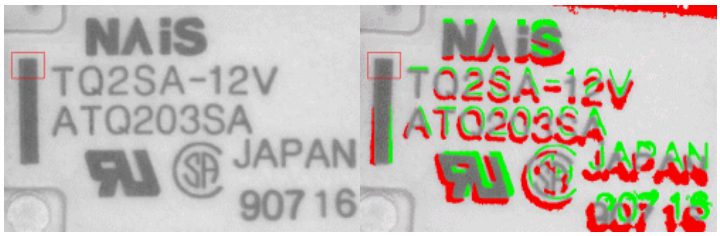
- the chosen image may itself have small, unnoticed defects and not be fully representative of the whole population.
- a single image gives no insight on the random variations between acceptable samples, and gives no way to adjust quality indicator tolerances.
- the default quality tolerances may not give the sharpness you would like to detect.

Statistics will help you define acceptance criteria. Quality indicator reference values are computed from the template image.

TEMPLATE DESIGN CONSIDERATIONS

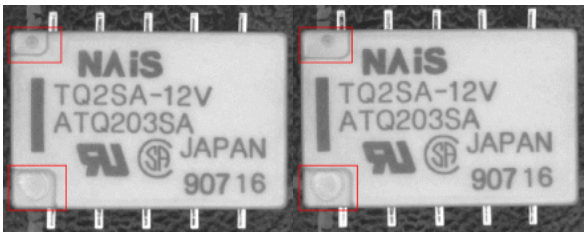
- **Should marking be one piece of text or several**? This depends on the possible movements; marks that move together should be considered a single text. During inspection, texts can move independently of each other.
- **How much can the text position vary in terms of translation, rotation** [and possibly scaling and/or shearing]?
- **Can the text be decomposed into characters** ? A character is the smallest part of a marking that can be inspected in isolation. A small displacement from its nominal position can be allowed and measured.
- **Can individual characters move with respect to their containing text** ?

- **What movement may occur** ? Placement repeatability must be evaluated. If the part travels along a guide, sometimes only translation occurs which can be handled by a single alignment pattern (fiducial). If rotation or scaling can occur, two alignment patterns are preferable.



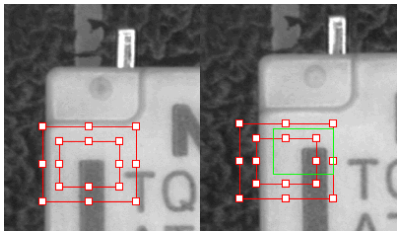
Bad: rotation not handled

- **Are the alignment patterns fixed, well-contrasted features, not subject to degradation, that move rigidly with the inspected part** ?
  When two are used, they should be located as far apart as possible for 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, but large enough to contain information at different scales.



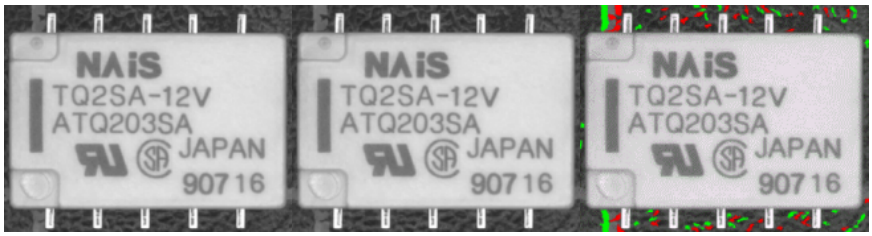
Bad: the location pattern is not repeatable

- Is the search area as small as possible to reduce search time and avoid false matches, but big enough to contain the match?
  Check on a representative set of images that location by pattern matching never fails by touching the search area edges.



Bad: too tight search areas

- Does the inspected ROI on the mother image surround all areas where defects may be detected, but not raise false alarms?

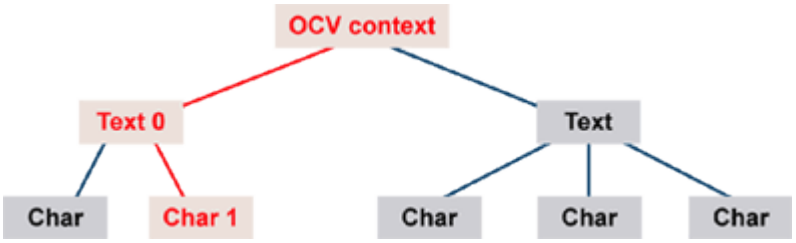


Bad: undue inspection of the background

ACCESS THE TEMPLATE COMPONENTS

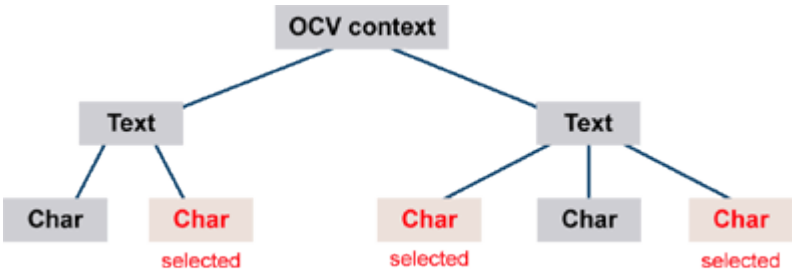
The OCV context contains a list of texts. Each text contains a list of characters.

To access a specific text, you provide its **index**, traversing the hierarchy from the OCV context. To address a particular character, you provide the indexes of both the text and the character.



Access by index (second character of first text)

You can 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 a property for all currently selected elements. 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 over selected components is very handy for interactive editors that list and modify parameters for single items or groups of items.

Learning Passes

After ROI placement and pattern learning (`Register` operations), training still requires two passes:

- To compute an **average** ideal, noise-free, image that reveals the central tendency of the part image.
  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).
- To measure **devations** around the average image.
  For each image, realign and normalize (`Register`).
  If the operation is successful, call
  `Learn(`ELearningMode_AbsDeviation`)` for immediate processing and recommended method, or
  `Learn(`ELearningMode_RmsDeviation`)` for enhancing large deviations.

The images used can be the same for both passes, or two distinct sets of images of different sizes can be used (on-the-fly learning). `BatchLearn` performs both passes for all images in the file list.
A learning set size of at least 16 images is recommended.

Inspect and compare image with model

Normally when you inspect the sample image and compare with the model, text-level inspection is sufficient.
Character-level inspection is more detailed and complex.
The inspection process involves two operations.

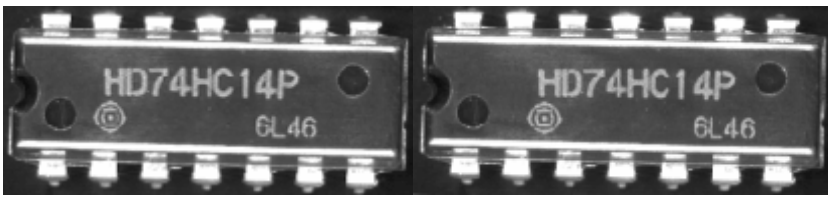
- **Locate**: A region of interest is scanned and the best match is found between it and the template, using all desired "Degrees of Freedom" below.
- **Score**: Every sample character is compared to the corresponding template character. Character quality indicators are computed, collated into "Quality Indicators" on page 159 quality indicators of the text, and compared to acceptance intervals. Unacceptable values have their corresponding characters flagged, a diagnostic code is generated, and global diagnostics summarizing all text and character defects are issued.

Degrees of Freedom

Degrees of freedom can compensate for misalignment and distortion. Each degree of freedom increases the running time, so 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.

TEXT TRANSLATION

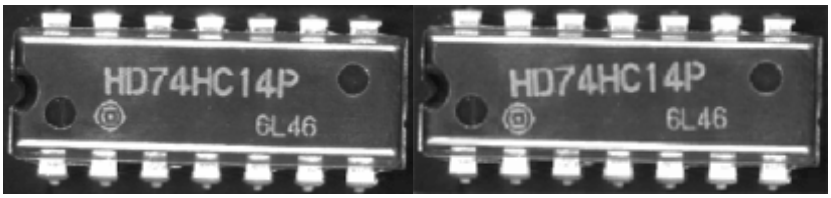
All texts can be moved horizontally (`ShiftX`) and vertically (`ShiftY`) in a specified range.



Text translation

TEXT SKEWING

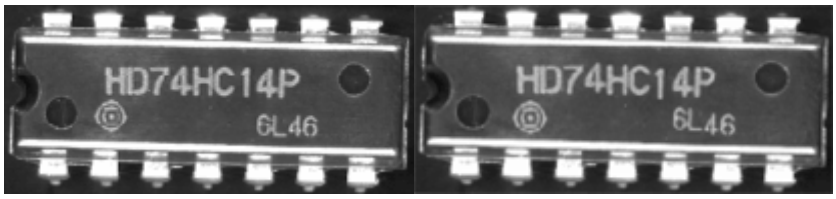
All texts can be rotated about the center of their bounding box using the angle defined by `Skew`.



Text skewing

CHARACTER TRANSLATION

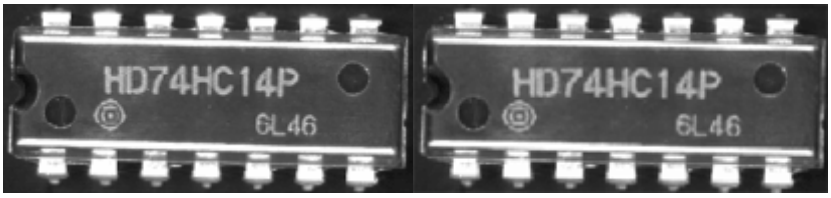
All characters can be moved individually horizontally (`ShiftX`) and vertically (`ShiftY`) with respect to their nominal position.



Character translation

TEXT X/Y-SCALING (ADVANCED - ONLY USE IF NECESSARY)

All texts can be re-scaled horizontally `ScaleX` and vertically `ScaleY`, while the center of their bounding box remains fixed. Re-scaling can be isotropic (both scale factors are identical) or anisotropic.

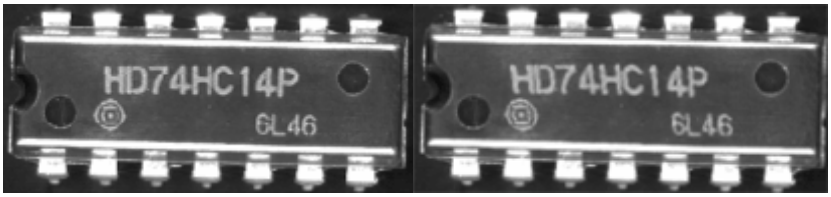


Anisotropic text scaling

By default, text scaling is supposed to be isotropic. 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 (ADVANCED - ONLY USE IF NECESSARY)

All texts can be sheared using the angle parameter `Shear`, i.e. become italic, while the center of their bounding box remains fixed.



Text shearing

DEGREES OF FREEDOM PARAMETERS

The degrees of freedom for location are specified with respect to the position at learning time *nominal position*. The characters are remembered relative to the corresponding text center. The text centers are remembered relative to the template image/ROI center. The default skew and shear angles are **0** and scale factors are **1**.

The parameters must vary within a range of `Bias ± Tolerance`. If bias is **0**, the range is centered around the nominal value.

The number of positions tried for each degree of freedom is specified as follows:

- **Translation**: 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 number of values tried for each degree of freedom. 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**.
  When `SkewCount` is set to **0**, EasyOCV automatically chooses an appropriate count value.

Quality Indicators

After locating the model, the inspection process compares the sample with the template, and rates the resemblance at a character level.

The parameters computed for the **template** serve as a reference and are compared to those computed on the **sample**.

When the template image is binarized, the marking appears as white foreground on black background in the character's bounding box. The bounding box is the tightest rectangle that wholly contains an item, with a safety margin.

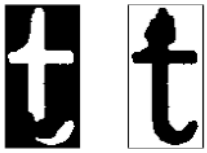
Template image, foreground and background (white pixels)

Sample image, foreground and background (white pixels)

AREA-BASED QUALITY INDICATORS

When the sample image is rated, thresholding also separates white and black pixels. The foreground (or background) sample areas are defined as the count of the white(or black) sample pixels in the foreground (or background) region of a characte. This is not the same as the total count of white and black pixels in the sample.

The difference between the template and sample areas is the area of defects in the character foreground [or background].



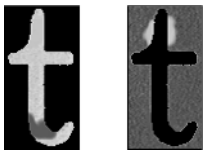
Foreground and background areas (white pixels)

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

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. Characters are accepted or rejected by comparing the indicator values of the sample and template: if the difference is larger than the specified tolerance, a defect is reported.
A smaller foreground value indicates under-printing or missing character parts. A larger background value indicates over-printing or spurious character parts. Character mismatches provoke both kinds of anomalies.

CORRELATION-BASED QUALITY INDICATORS

Normalized correlation rates 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.

The correlation-based quality indicator should be as close as possible to **1**. It is not sensitive to changes in gain or offset.

REPORTING

Defects are reported in three ways:

- explanatory diagnostics are given for each inspected character and text;
- the items for which diagnostics are reported are highlighted on the display;
- the relevant items are drawn as a box crossed by its main diagonal.

Advanced Features

EasyOCV can accumulate the results of a series of consecutive inspections: quality indicators average values and standard deviations. EasyOCV functions can use these statistics to define acceptance criteria and automatically adjust position and quality tolerance parameters, so you can better control the manufacturing process.

- Average values show the long-term trend behavior of the system and detect marking drifts.
- Standard deviations show process repeatability and detect appearance of slack.
  It is customary to select a tolerance value that is a small multiple of the observed standard deviation (±2 sigma or ±3 sigma criterion).

Programming with EasyOCV

INTRODUCTION

Writing an inspection application for the production line can be simple if no operator intervention is required to adjust parameters, and if no in-situ learning phase is necessary:

- A model can be loaded once for all.
- Every acquired image is inspected.
- The inspection results are graphically displayed on top of the image.
- A diagnostics report of quality indicators and statistics can be generated.

A more advanced inspection application may allow the operator to modify parameters:

- Dialog boxes must be provided to edit parameters.
- Parameters may be changed globally (same value everywhere), or the operator may select the texts and characters on which to work.

In a complex application, model edition before learning must be made possible which leads to more complex programming:

- select and unselect items.
- display and modify parameters.
- load and save to a model file.

The required steps to create an inspection application, from simplest to advanced are:

1. SET ROI FOR INSPECTION

The ROI should be placed center on the marking, in a position that is repeatable 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 pattern matching or edge measurement. .

The ROI :

- Defines the effective search area for text using their centers and location parameters (ShiftX, ShiftY, ...). This way, inspection is not confined to the inspected ROI.
- Locates the marking, then evaluates global contrast in this ROI (centered on the marking with the learning time dimensions). The ROI is not used to evaluate global contrast of the marking.

2. INSPECT

Uses a threshold level to determine the global contrast of the marking. Automatic thresholding can be used.

3. DRAW INSPECTED ITEMS

The inspected texts and characters can be represented by their bounding box, at the position determined by the location process.

Selected and unselected items can be drawn in different colors.

Items with detected defects appear crossed by their main diagonal.

4. RETRIEVE DIAGNOSTICS AND QUALITY INDICATORS

Global inspection diagnostics summarize all defects found on the marking and raise alarms.

Detailed diagnostic reports are available for each text and character, and all measured quality indicators can be retrieved.

- Retrieving text diagnostics and parameters involves a loop where all texts are visited.
- Retrieving character diagnostics and parameters involves a double loop where all texts and all characters of all texts are visited.

5. SET INSPECTION PARAMETERS

During operation working parameters can be adjusted in various ways:

- **Global change**: a parameter value may be set for all texts and/or all characters. This is straightforward and requires a single call to `ScatterTextsParameters`, `ScatterTextsCharsParameters`, but before calling, make sure that the parameters you don't want to change are set to an undefined value.
- **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.
- **Selective change**: parameter values can be set for texts or characters in a selected state by interactively selecting the text or characters.

SELECTING ITEMS INTERACTIVELY

To retrieve or modify parameters, individually or grouped, the operator must have the ability to select them using a mouse. EasyOCV provides a general selection/de-selection mechanism: several functions can toggle the state of all/selected/unselected items in a given rectangle.

The rectangle is usually obtained by a dragging operation. A degenerate rectangle (reduced to a single point) can be used 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).

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...`).

COMPUTE 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`.

ADJUST INSPECTION PARAMETERS FROM STATISTICS

Statistics may be used to adjust location and quality indicators. When making adjustments to individual texts or characters, the selection mechanism described above is applicable.

To adjustquality ranges indicators, call `AdjustTextsQualityRanges`, `AdjustCharsQualityRanges`. If you do not want to adjust the quality range of a particular indicator, you should de-activate it by setting `UsedQualityIndicators`.

- the bias value of each indicator is assigned the average value of the inspected samples (provided they had been added to statistics).
- the indicator tolerance is assigned s times the standard deviation, where s is a security factor to provide.

To adjust location parameters, call `AdjustTextsLocationRanges`, `AdjustCharsLocationRanges`. When adjusting location parameters, you must specifyminimum and maximum values and a security factor may also be specified.

INTERACTIVELY EDIT 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.

ADVANCED FEATURES: CHANGE CONTRAST, LOCATION MODE, QUALITY INDICATORS AND RESAMPLE CHARACTERS

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, that may be determined automatically. After a threshold is given, the average gray level of the background and foreground are computed separately, and become the reference gray levels. These are used 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 contrast may then be compared to the templates contrast reference value, to diagnose an over-contrasted or under-contrasted image before further analysis.

The template threshold directly influences the thickness of the blobs in the model.

The sample threshold influences the reference gray levels of the sample image.

When gray level normalization is used, it influences location score,gray-level sums, and blob thickness in the sample image, which has immediate consequences on the sample areas.

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 binarized and gradient modes are the most reliable at locating components.. Additionally, the gradient mode is not sensitive to the threshold level. Use of the Laplacian mode is not recommended.

In case you experience location problems, you should try another location mode.

LOCATION SCORE

It is highly recommended to keep the default reduction of location scores option turned on. Reduction consists of 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 if the lighting conditions of sample images are not the same, 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). Location scores do not depend on reference gray levels, so are more reliable.

Finally, `AccurateTextsLocationScores` provides an alternative way to compute text location score. During the location process, EasyOCV tries first to locate texts using their contours, which are sets of fixed points one from each other. This rigid definition of text contour has the drawback of making 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**.

EChecker Concept

`EChecker` requires two processing steps, Training and Inspection. These two operations are totally independent and can be programmed in separate applications.

Training involves pre-processing the set of reference images to compute the acceptance ranges at each pixel and to store them in two threshold images for use with EasyObject. You can test that the Golden Template is good and tune the learning process parameters if necessary. The training can be done once for all and the results archived in a Golden template file for later use.

Inspection involves processing an image, realigning and normalizing, detecting pixels that fall out of the acceptance intervals, checking the quality,and then tuning the inspection parameters if necessary.

The following sections present the relevant API functions for use in the training and inspection steps.

TRAINING

Reference images are preprocessed to compute the pixel acceptance ranges, and store them in two threshold images for use with EasyObject (legacy). The training results are archived in a model file for later use.

Two modes of operation are provided, depending on whether reference images are stored on disk or are acquired and processed on the fly.

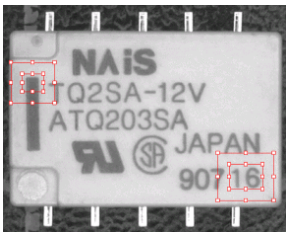
These operations cannot be used during on-line operation.

To define a model, several operations are performed, in this order:

1. 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 others to define the possible movement freedom, and are used as search areas for pattern matching. ROIs can be placed interactively using dragging handles.
   `EChecker` manages the dragging operations.
   - `Draw` graphically renders the ROIs and handles.
   - `HitTest` detects the presence of the cursor over one of the handles.
   - `Drag` moves a handle.
   These functions operate on all three ROIs (pattern, search and inspected). You will quickly notice that:

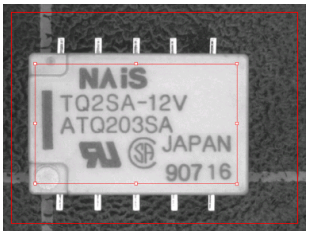
- Dragging a pattern ROI causes the corresponding search ROI to adjust automatically so tolerances remain constant.
- Dragging a search ROI causes its size to adjust symmetrically with respect to the pattern.

- Adjusting a search area also sets the inspected ROI to the largest available space in the image.



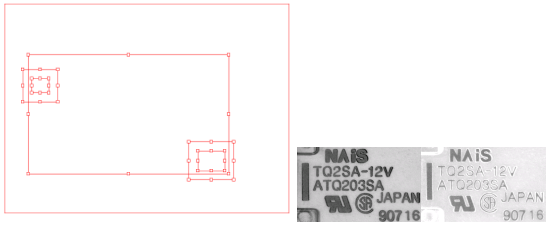
Pattern ROIs and search ROIs

- If the training images are on disk, the list of file names can be registered and used later to execute learning in a single command (*batch learning*, as opposed to *on-the-fly learning*).
- Another ROI is defined to delimit the area to be inspected. This area must only include pixels of the rigid part (that moves with the fiducial marks), and not the background.



Inspected ROI

- All images are processed and are averaged using Statistical training. It uses realignment to deal with displacement of the inspected part in the field of view, and gray-level normalization to deal with global illumination changes.
- Ideally at least 16 images should be used in learning passes to create the low and high threshold images. The user can strengthen or loosen the acceptance intervals using a global tolerance parameter. Call `Learn(ELearningMode_Ready)`. Property `RelativeTolerance` adjusts the acceptance ranges.
- The model can 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

CHECK THE REFERENCE IMAGES FOR RELIABILITY.

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 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.

The first call to member `Register` invokes 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*.

If the learning images are saved on disk: Every time a file is successfully loaded and is accepted as a reference image, the `EChecker` object can add the file path name to a list. Later, all files can be processed in a single command.

MORE LEARNING

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.
  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).
- The **"deviation" pass** measures variations around the average image.
  For each image, realign and normalize (`Register`). If the operation was successful, call `Learn(`ELearningMode_RmsDeviation`)` (enhances the large deviations), or
  `Learn(`ELearningMode_AbsDeviation`)` for immediate processing (more robust and is recommended in most circumstances).

In principle, the images shown during those two passes should be the same.
If you do not want to archive them, 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.

Alternatively, calling `BatchLearn` will perform the two required passes for all images added to the file list.

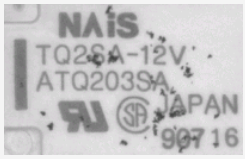
Adjust thresholding

`Learn(`ELearningMode_Ready`)`. Property `RelativeTolerance` can be adjusted to adjust the acceptance ranges.

INSPECTION

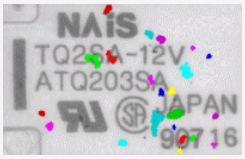
Inspection is straightforward: the sample image is realigned with the model file and gray-level normalized.

The inspected ROI must be positioned on the mother image. It 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. Changing the search tolerance will reset the inspected ROI to the largest available area. `EChecker` can tune the global `RelativeTolerance` property at inspection time, thus changing lower and upper threshold images used by EasyObject (legacy).
This is the only `EChecker` parameter that can be changed after learning.



Inspected image, ...

... high and low threshold images, ...



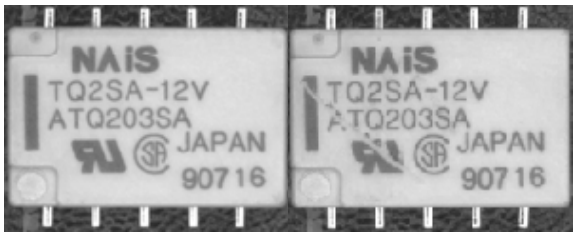
... and detected blobs.

The resulting image is passed to an `ECodedImage` object for blob analysis; which groups neighboring pixels to form blobs, discards the smaller blobs (usually noise), measures geometric characteristics (location, size, orientation, ...) and others, see EasyObject.

This image is then compared to the lower and upper reference images (a comparison is made pixel by pixel between the sample and the template to detect pixels that fall out of acceptance intervals). Defective pixels are handled by standard EasyObject functions.

COMPARE USING ECHECKER

`EChecker` finds visible differences between templates and samples, and reports them in a defect map which highlights significant differences. EasyObject blob analysis tools can then locate the defects and qualify them in terms of extent, orientation, lightness and so on. It is ideally suited when inspected items are rigid (shape does not vary) and illumination is uniform, ensuring repeatable visual appearance of the image, so that point to point image comparison makes sense.



Reference image vs. sample image with defects

Image Comparison

The best way to compare images is using `EChecker` to combine a set of images and determine the range of acceptable values at each pixel.

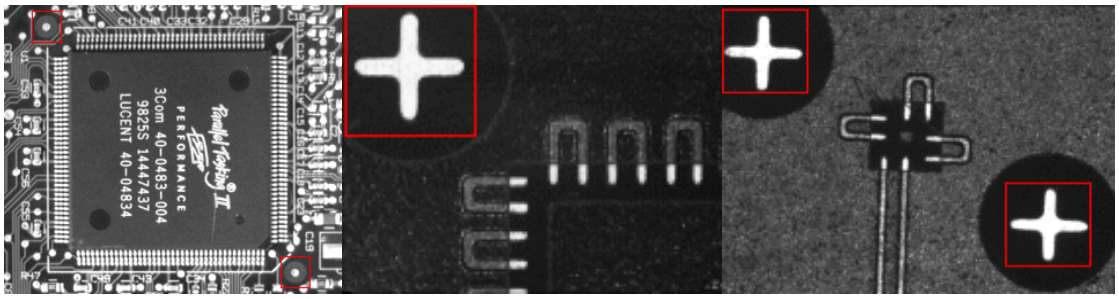
Images can also be compared by subtracting 2 images (using Arithmetic and Logic functions) to get an absolute value, then thresholding this difference to highlight the non-zero pixels where the images differ.

ECHECKER PERFORMS:

ALIGNMENT

Pattern matching measures movement of the inspected part in terms of translation, rotation and/or scaling (as described in the EasyMatch chapter).
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.
A single matching pattern easily handles translation.
Two matching patterns provide better accuracy for rotation measurement.

When fiducial marks are available, they can be used as landmarks for accurate and repeatable location.
`EChecker` hosts one or two matching contexts internally.

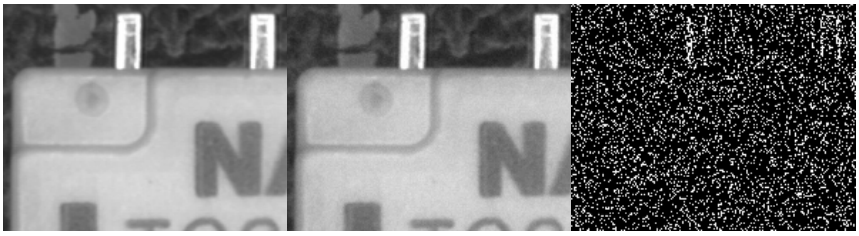


Realignment using fiducial marks

THE ALIGNMENT METHOD HAS THREE SHORTCOMINGS:

1) UNAVOIDABLE VARIATIONS

Tolerance must be introduced as noise can make identical images have slight variations in color or shape.

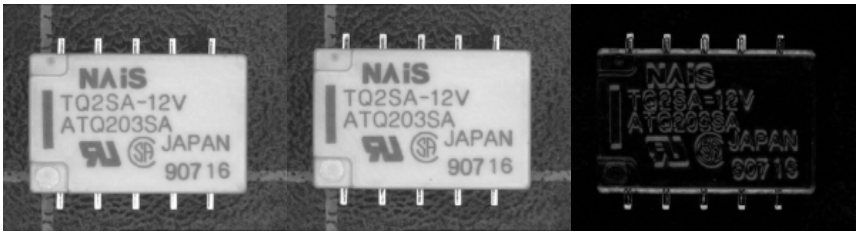


Comparing two noisy images for strict equality.
Black pixels of the result image are non equal pixels.

2) PLACEMENT OF INSPECTED PARTS IS RARELY REPEATABLE

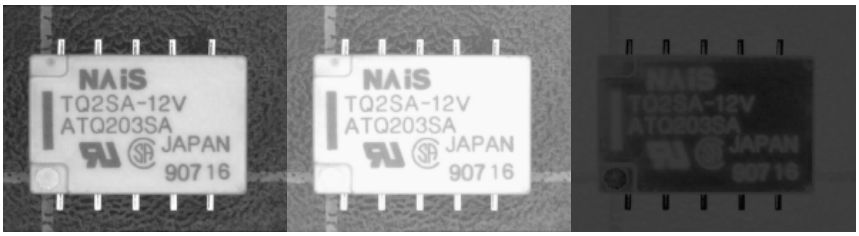
Slight misplacement means the compared pixels no longer correspond, the resulting effect is

especially noticeable around edges.



Comparing misaligned images

3) LIGHTING CANNOT BE SEPARATED FROM PARTS



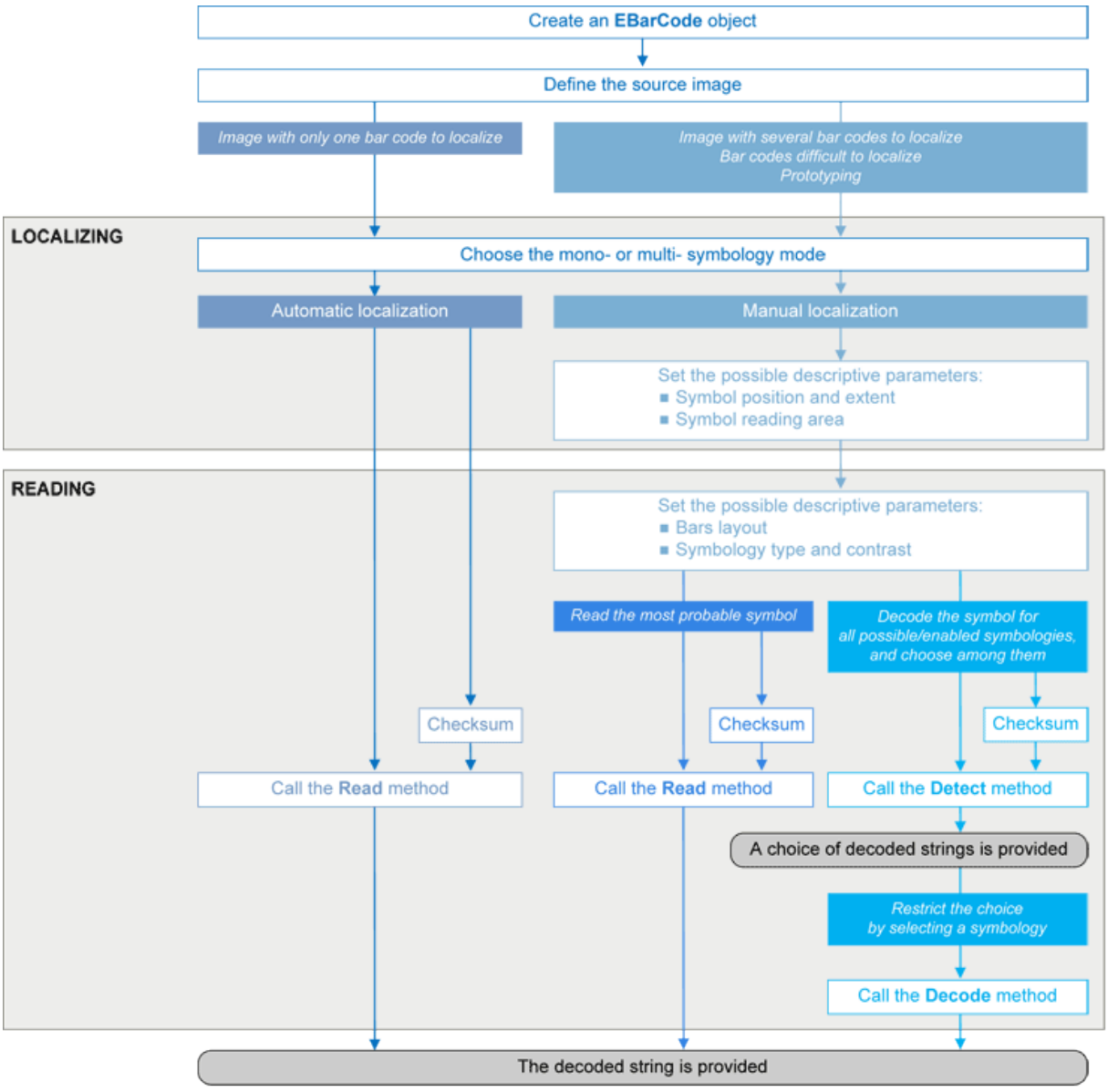
Comparing images in different lighting conditions

EasyBarCode

Bar code (EAN 13 symbology)

EasyBarCode can locate and read bar codes automatically.
Location can be performed manually for prototyping or when automatic mode results are unsatisfactory.

WORKFLOW



BAR CODE DEFINITION

A bar code is a 2D pattern of parallel bars and spaces of varying thickness that represents a character string. It is arranged according to an encoding convention (**symbology**) that specifies the character set and encoding rules.

- The bar code may be black ink on white background or inverted (white ink on black background).
- The bar code should be preceded and followed by a quiet zone of at least ten times the module width (smallest bar or space thickness).
- Bars should be surrounded below and above by a quiet zone of a few pixels.
- Bar and space widths must be greater than or equal to 2 pixels.

SYMBOLOGIES

A symbology defines the way a barcode is encoded.

Symbologies can be enabled in `StandardSymbologies` or `AdditionalSymbologies` parameters.

The standard symbologies are enabled by default:

- Code 39
- Code 128
- Code 2/5 5 Interleaved
- Codabar
- EAN 13*
- EAN 128
- MSI
- UPC A*
- UPC E

* EAN 13 and UPC A only differ by the layout of surrounding digits.

Additional symbologies that are supported:

- ADS Anker
- Binary code
- Code 11
- Code 13
- Code 32
- Code 39 Extended (a super-set of Code 39)
- Code 39 Reduced (a subset of Code 39)
- Code 93
- Code 93 Extended
- Code 412 SEMI
- 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
- Code STK
- EAN 8
- IBM Delta Distance A

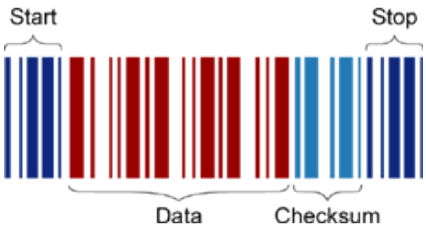
- Plessey
- Telepen

CHECKSUM

A checksum character enables the reader to check the barcode validity depending on the symbology:

- The checksum may be mandatory and must be checked by the reader.
- The checksum may be mandatory but may not need to be checked.
- The checksum and its verification may both be optional.

`VerifyChecksum` enables or disables (default) checksum verification.



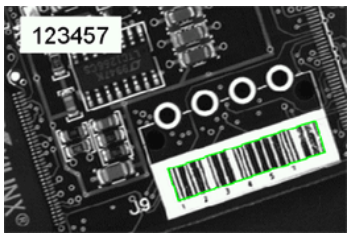
Bar code structure (Code 39)

READ A BAR CODE

The **Automatic** mode reading algorithm locates a bar code in the field of view and Reads it. If several bar codes are present, only one is located, like a straightforward hand-held bar code reader.

Before reading, the decoding symbologies must be specified in the StandardSymbologies, or AdditionalSymbologies properties.

Mono-symbology mode reads the bar code using the expected symbology type(s) and reports the encoded information (if readable) or the reason for failure (if not readable). There is only one interpretation for the character string.



Decoded bar code

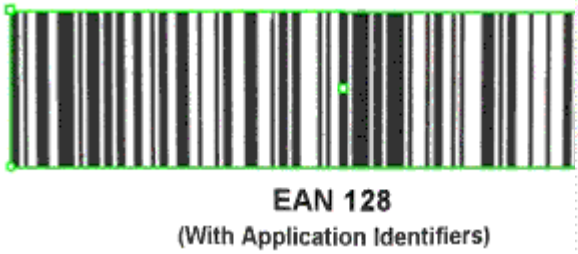
Note: When the bar code contains \0x00 characters, the `std::.string::.c_str` method should not be used (since C-strings are terminated by the \0x00 character). An iterator over the characters should be used instead of a C-string.

ADVANCED FEATURES

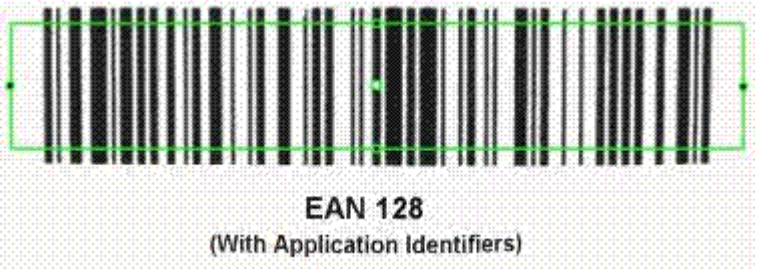
LOCATE AND READ BAR CODE MANUALLY

If automatic localization fails or for prototyping purposes, the user can provide the **bar code position** and **reading area** to manually locate the code.

- **Bar code position** can be provided graphically by a bounding box around the bar code or by its parameters. If several symbols appear in the image, they can be processed one after the other.
- The **reading area** of the bar code is the area that is read. It should be wider than the bar code bounding box width, and less high than the bar code bounding box height. It may also be rotated relatively to the bar code bounding box, to take into account slanting bars (Advanced mode!).



Bounding box — graphical appearance (manual location)



Reading area — graphical appearance (manual location)

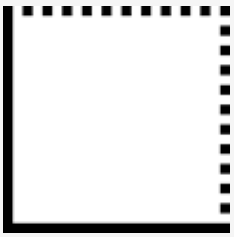
READ ALL INTERPRETATIONS (MULTI-SYMBOLOGY MODE)

Use `Detect` to report the number of possible symbologies in the `NumEnabledSymbologies` property, and list the data contents by decreasing likeliness.

Then call the `Decode` method in a loop, using `GetDecodedSymbology` to walk through the list of successful symbologies in decreasing order of likelihood.

EasyMatrixCode

Data Matrix code (ECC 200, 26x26 cells)



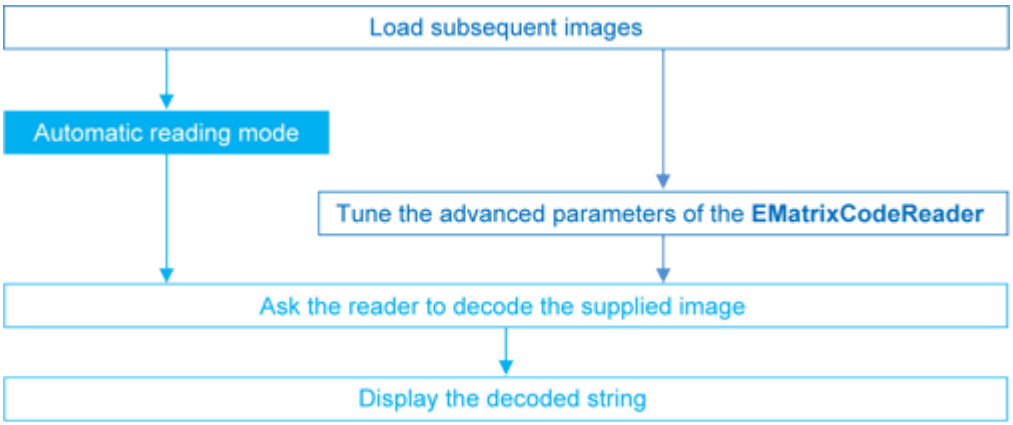
Data Matrix code finder pattern

In a single read operation, EasyMatrixCode locates, unscrambles, decodes, reads and grades the quality of grayscale 2D Data Matrix codes of any size, contrast, location and orientation (even viewed from the back on a transparent medium), providing they meet these 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 height): 2.

A data matrix code can be read even when damaged, using a built in error correction system.

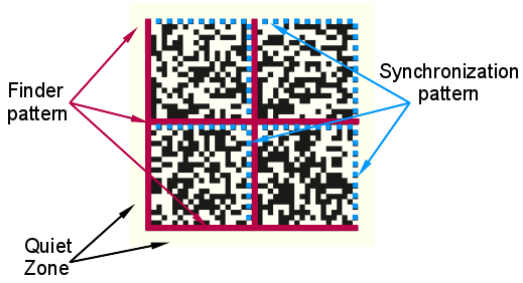
WORKFLOW



MATRIX CODE DEFINITION

A data matrix code is a two-dimensional rectangular array of black and white cells which conveys a string of characters (digits, letters and special characters). It is encoded to achieve maximum packing. Each cell corresponds to a bit of information, redundant bits are added to allow error correction for robust reading of degraded symbols.

- It is located using the **Finder pattern**: The bottom and left edges of a Data Matrix code contain only black cells, while the top and right edges have alternating cells.



- It is characterized by its logical size (number of cells), contrast type and its **encoding type**; ECC 000, ECC 050, ECC 080, ECC 100, ECC 140 (odd symbol sizes) and ECC 200 (even symbol sizes).

The data matrix code definition is provided by AIM International Inc. (PA) and is approved as standard ANSI/AIM BC11-1997.

READ A MATRIX CODE

You can Read a matrix code automatically, or you can read a saved matrix code.

`EMatrixCode` and `EMatrixCodeReader` objects feature `Save` and `Load` methods which use a file to store and restore the object state (learned parameters, decoded string, grading values, ...). The process of saving and loading files is called serialization.

To restore the state of an EMatrixCode and use it to read a matrix code:

1. Load an image.
2. Restore the reader state from the given file `EMatrixCodeReader::.Load`.
3. Read the image.
4. Display the decoded string.

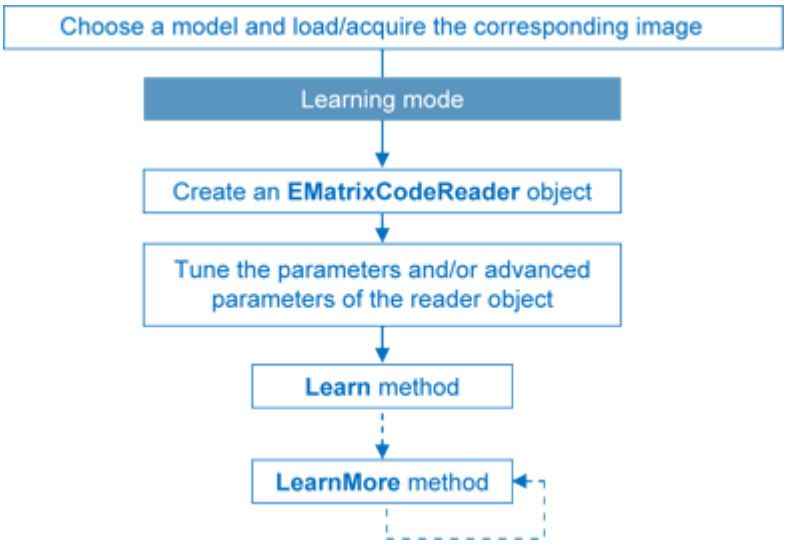
READ PRINT QUALITY

Print quality can be computed as defined by BC11, ISO 15415, ISO/IEC TR 29158 and SEMI T10-0701 standards.
Simply **enable grading** before the Read method, then GetIso15415GradingParameters, GetIso29158GradingParameters or GetSemiT10GradingParameters.

ADVANCED FEATURES

You can search for specific features by learning a Matrix code model.

Learn Workflow



Learn and read method

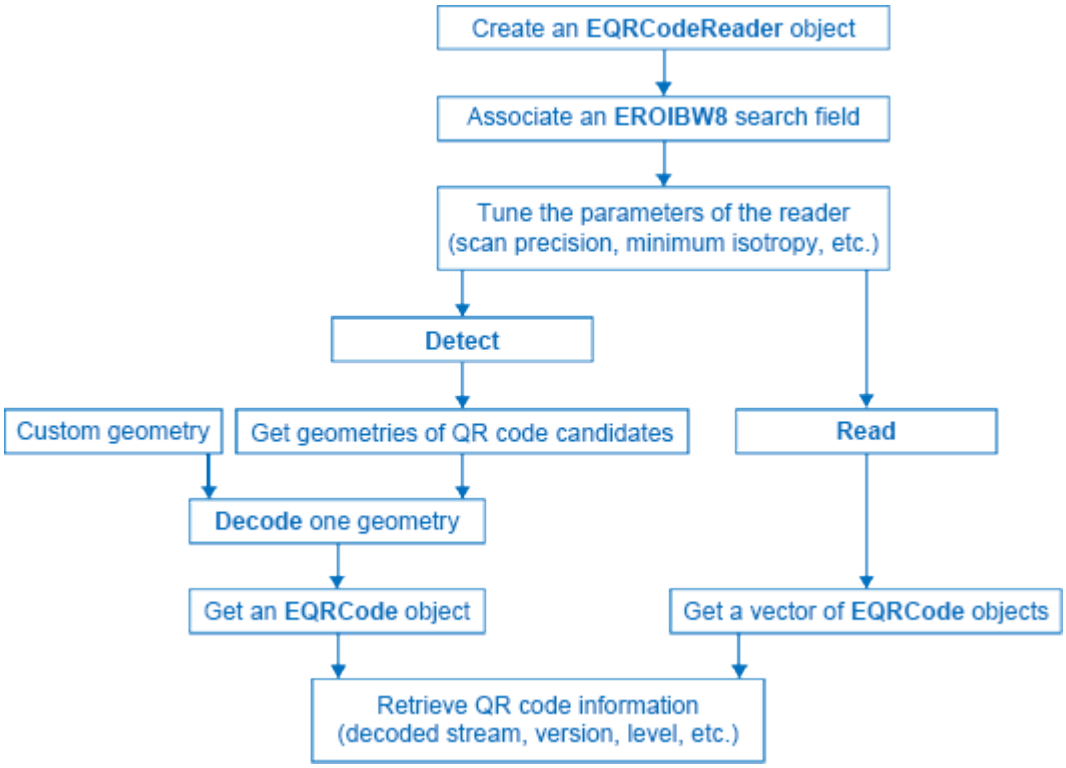
1. Load the image of the Matrix Code to be learned.
2. Learn the model : using the `Learn` method with Contrast, Family, Flipping, Logical Size parameters.
   If several matrix codes need to be learned, then use `LearnMore`, and pass additional sample images.
   Calling `Learn` replaces `EMatrixCodeReader` parameters, so calling `Learn` several times does not accumulate results, which `LearnMore` does.
3. Tune search parameters to be efficient:
   - read only matrix codes that match a sample matrix code,
   - or read only matrix codes that have the same properties (Contrast, Family, Flipping, Logical Size) as the learned one,
   - or disregard a search parameter of the learned matrix code `SetLearnMaskElement`, for example to read only unflipped matrixcodes. Just remove the default parameters, then add new ones.
4. Ask `EMatrixCodeReader` to decode the supplied image.
5. Display the decoded string.
6. `Save` the state of the reader object.

EasyQRCode

EasyQRCode detects QR (Quick Response) codes in an image, decodes them, and returns their data.

Error detection and correction algorithms ensure that poorly-printed or distorted QR codes can still be read correctly.

WORKFLOW



QR CODE DEFINITION

A QR code is a square array of dark and light dots. One dot (or "**module**") represents one bit of

information.

QR codes contain various types of data and can be different models, versions, and levels. They always contain a message, metadata about alignment, size, format, and error correction bits. They comply with the international standard ISO/IEC 18004 (1, 2 and 2005).

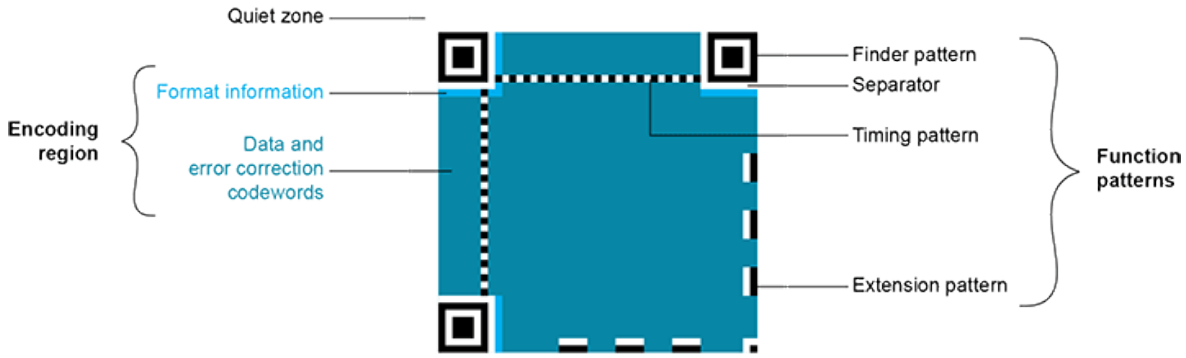
QR CODE STRUCTURE

The QR code symbol consists of an **encoding region**, containing data and error correction codewords, and of **function patterns**, containing symbol metadata and position data.

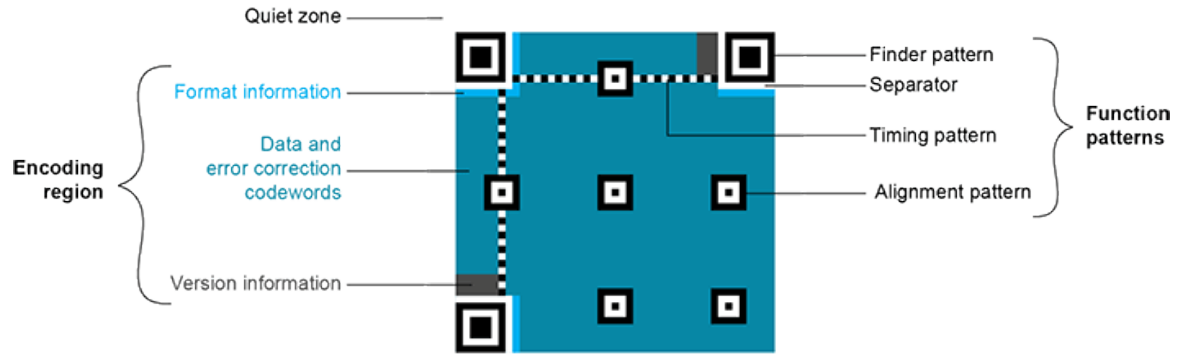
A QR code must be structured with the following elements:

- **Quiet zone**: blank margin around the QR code
- **Finder patterns**: recognizable zones identifying a QR code
- **Extension patterns**: markers for the alignment of the QR code (model 1)
- **Alignment patterns**: markers for the alignment of the QR code (models 2 and 2005)
- **Timing Patterns**: data giving the module size (in pixels)
- **Format information**: zones providing the QR code level
- **Version information**: data giving the QR code size, for instance 25 x 25 modules (models 2 and 2005)
- **Data contents and error correction codewords**: the primary information carried by the symbol, with additional information for error correction

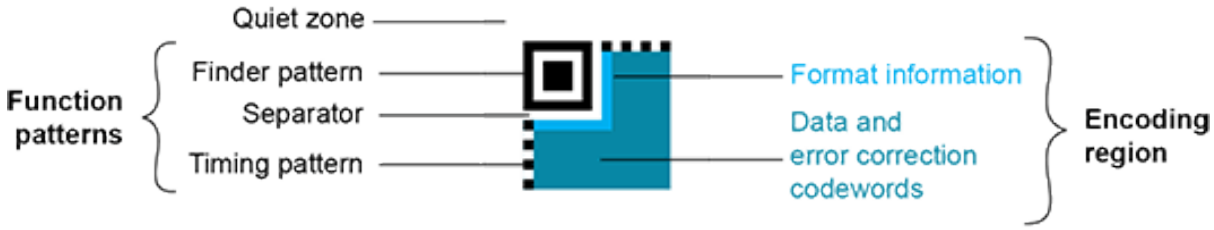
Variants of this structure exist, according to the model, format, or version of the QR code. For instance, model 1 QR codes do not feature alignment patterns but extension patterns. Micro QR codes include only one finder pattern, and no alignment pattern. EasyQRCode can read all of them.



Structure of a model 1 QR code symbol



Structure of a QR code 2005 symbol



Structure of a Micro QR code symbol

DATA TYPES

The QR code data can be any mix of these types:

- Numeric data (0-9)
- Alphanumeric data (0-9, A-Z, /, $, %, etc.)
- Byte data
- Kanji characters

MODELS (STANDARDS)

- **Model 1**: original QR code international standard, with versions ranging from 1 to 14. Note that the "version" of a QR code is the symbol size (in number of modules). It does not relate to the version of the standard, which is called the "model".
- **Model 2**: improvement of model 1. It provides versions from 1 to 40. It defines alignment patterns to improve reading of distorted QR codes, or QR codes printed on curved surfaces.
- **Model 2005**: improvement of model 2, including white-on-black QR codes, and mirror symbol orientation.
- **Micro QR codes**: (not yet supported) smaller QR codes, from version M1 to version M4. They have been introduced to save printing space.

VERSIONS (SYMBOL SIZE)

- **QR codes**: from version **1** (21 x 21 modules) to version **40** (177 x 177 modules), with an increment of +4 x +4 modules (version 2: 25 x 25 modules, version 3: 29 x 29 modules, ..., version 39: 173 x 173 modules).

- **Micro QR codes**: (not yet supported) version **M1** (11 x 11 modules), version **M2** (13 x 13 modules), version **M3** (15 x 15 modules), version **M4** (17 x 17 modules).

Examples of QR codes
From left to right:
Micro QR code, version M3, 15 x 15 modules,
Model 2 QR code, version 4, 33 x 33 modules, 67-114 characters,
Model 2 QR code, version 40, 177 x 177 modules, 1852-4296 characters

LEVELS (ERROR CORRECTION)

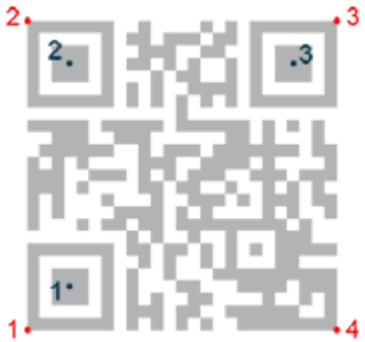
QR codes contain error correction data. The standard offers the following levels of error correction:

- **L**: (low) about 7% of codewords can be restored
- **M**: (medium) 15%
- **Q**: (quality) 25%
- **H**: (high) 30% (not available for Micro QR codes)

QR CODE GEOMETRY

When the QR code reader finds an array of dots that could match a QR code, it returns the "geometry" of this QR code candidate.

A QR code geometry is a set of points. It contains the coordinates of the corners of the QR code quadrilateral (bottom left, top left, top right, bottom right), and the coordinates of the finder pattern centers (bottom left, top left, top right).



QR code geometry

READ A QR CODE

Reading a QR code returns information about QR codes for which detection and decoding were successful.

This is equivalent to detecting and decoding all QR codes in the given search field (see advanced features).

ADVANCED FEATURES

DETECT A QR CODE

1. Set a search field on an EROIBW8 image or tune the parameters to restrict the numbers of operations to process.
2. The QR code reader scans the image and searches for 3 finder patterns that could match a QR code, with the following requirements:
   - Minimum quiet zone (blank zone around the QR code) width: 3 pixels.
   - Minimum module size: 3 x 3 pixels.
   - Minimum isotropy: 0.5.
   - Maximum corner deformation: 15° (corner angles can range from 75° to 105°).
3. The reader returns QR code candidates, or the result of a detection, as a vector of geometries.

DECODE A QR CODE

1. The QR code reader decodes a QR candidate and returns the QR code: model, version, level, geometry and decoded stream of data.

   The decoded stream class consists of a coding mode (basic, FNC1/GS1, or FNC1/AIM), and an application indicator (if the coding mode is FNC1/AIM, otherwise 0). The decoded data can be accessed from each part of the decoded stream, according to its encoding (numeric, alphanumeric, byte, or Kanji). You can also get the raw bit stream (the bit data after unmasking and error correction, but before decoding as a vector of bytes).

2. The reader can report the amount of unused error correction.
   - Close to 1, very few errors were corrected when decoding the data. The decoding is highly reliable, and the QR code is of good quality.
   - Close to 0, many errors were corrected when decoding the data. The decoding is reliable, but the QR code quality is poor.
   - -1, error correction failed. Decoding was not performed.

TUNING PARAMETERS

Scan precision: You can change the scan precision to scan the search field with a fine (recommended for small QR codes), or coarse (recommended on medium to large QR codes) precision.

Minimum score: The QR code reader searches for this QR code finder pattern: 

A perfect match returns a pattern finder score of 1.
Less accurate matches return lower scores.
The minimum score allowed by default is 0.65 - you can tune this.

Minimum isotropy: The isotropy of a QR code represents its rectangular deformation.
Perfectly square QR codes have an isotropy of 1 (short side divided by long side, whether the rectangle is vertical or horizontal).
EasyQRCode can detect rectangle QR codes with an isotropy down to 0.5. The default minimum isotropy is 0.8, it can be tuned from 0 to 1.

Square and rectangular QR codes (isotropy = 1, 0.5, and 0.5 from left to right)

Model and version: The QR code reader searches for QR codes of all models, and all versions. You can shorten the process by specifying the QR code model(s) and a range of versions (from 1 to 40) to be searched for.

Statistics

EasyObject statistics are related to the objects in an image.

"Statistics" above are related to whole images (global illumination / contrast, saturation, presence or absence of an object).

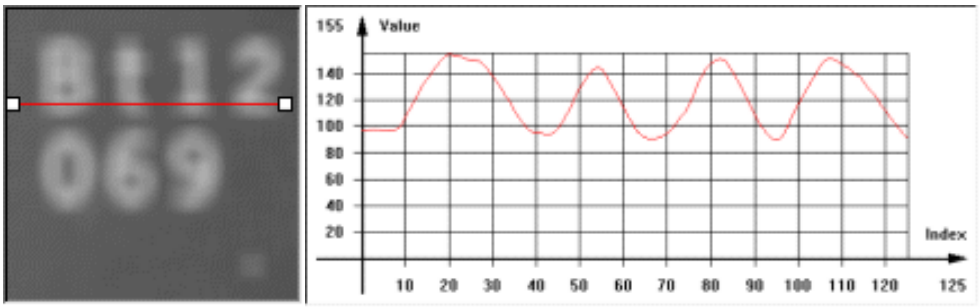
SLIDING WINDOW (CREATES NEW IMAGE OF AVG OR STD DEVIATION OF GRAY-LEVEL VALUES)

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

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

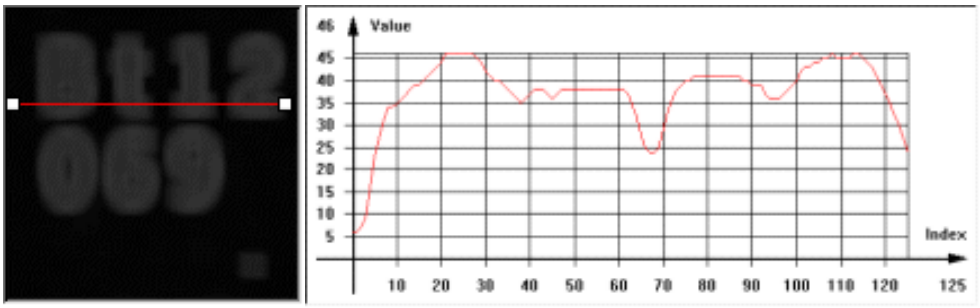
The result of the operation is another image.

The **local average**, `EasyImage::.LocalAverage`, corresponds to a strong low-pass filtering.



Sliding window average

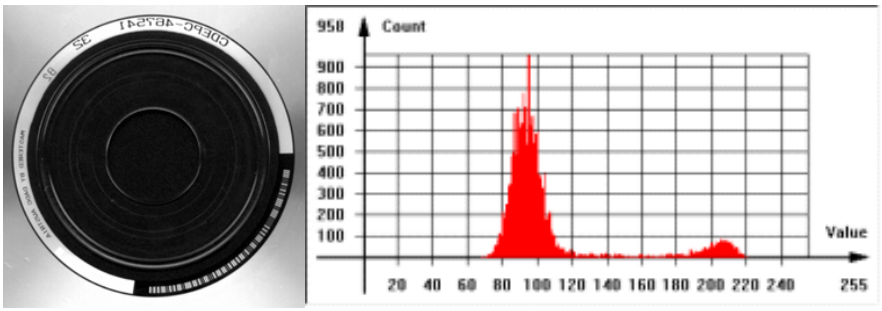
The **local standard deviation**, `EasyImage::.LocalDeviation` enhances the regions with a high frequency contents, such as noisy or textured areas.



Sliding window standard deviation

HISTOGRAM COMPUTATION AND ANALYSIS(AND LUT CREATION)

A histogram is a statistical summary of an image: it shows the number of occurrences of every gray-level value in an image, and it's shape reveals characteristics of the image. For instance, peaks in the histogram curve correspond to dominant colors in the image. If the histogram is bi-modal, a large peak for the dark values corresponding to the background, and smaller peaks in the light values.



Typical image histogram

HISTOGRAM COMPUTATION

`EasyImage::.Histogram` computes the histogram of an image. It has an input mask argument.
It supports flexible mask.
BW8, BW16 and BW32 source images are supported.

You can compute the cumulative histogram of an image, i.e. the count of pixels below a given threshold value, by calling `EasyImage::.CumulateHistogram` after `EasyImage::.Histogram`.

HISTOGRAM ANALYSIS

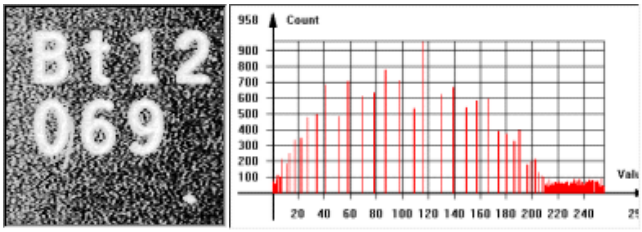
`EasyImage::.AnalyseHistogram` and `EasyImage::.AnalyseHistogramBW16` provide statistics and thresholding values:

- 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.

HISTOGRAM EQUALIZATION

`EasyImage::.Equalize` re-maps the gray levels so that the histogram fills in the whole dynamic range as uniformly as possible.

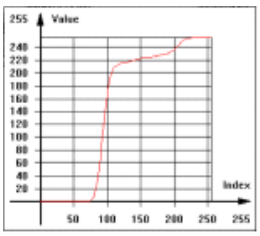
This may be useful to maximize image contrast, or reveal a lot of image details in dark areas.



Equalized image and histogram

SETUP A LOOKUP TABLE

`EasyImage::.SetupEqualize` creates a LUT so you can work explicitly with the histogram and LUT vectors. It can be more efficient to keep the image histogram for other purposes (i.e statistics) and keep the equalization LUT to apply to other images.



Equalization lookup table

IMAGE FOCUS

Sharp focusing can be achieved if the `EasyImage::.Focusing` quantity is maximum for a given image. This function must be called multiple times with multiple images with a different focus for the basis of an "auto-focus" system.

EASYIMAGE STATISTICS FUNCTIONS

AREA (NUMBER OF PIXELS WITH VALUES ABOVE/ON/BETWEEN THRESHOLDS)

- `EasyImage::.Area` counts pixels with values above (or on) a threshold.
- `EasyImage::.AreaDoubleThreshold` counts pixels whose values are comprised between (or on) two thresholds.

BINARY AND WEIGHTED MOMENTS (OBJECT POSITION AND EXTENT)

- `EasyImage::.BinaryMoments` computes the 0th, 1st or 2nd order moments on a binarized image, i.e. with a unit weight for those pixels with a value above or equal to the threshold, and zero otherwise. It provides information such as object position and extent.
- `EasyImage::.WeightedMoments` computes the 0th, 1st, 2nd, 3rd or 4th order weighted moments on a gray-level image. The weight of a pixel is its gray-level value. It provides information such as object position and extent.

GRAVITY CENTER (AVERAGE PIXEL COORDINATES ABOVE/ON THRESHOLD)

- `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.

PIXEL COUNT (BETWEEN 2 THRESHOLDS)

- `EasyImage::.PixelCount` counts the pixels in the three value classes separated by two thresholds.

MINIMUM, MAXIMUM AND AVERAGE GRAY-LEVEL VALUE

- `EasyImage::.PixelMax` computes the maximum gray-level value in an image.
- `EasyImage::.PixelMin` computes the minimum gray-level value in an image.
- `EasyImage::.PixelAverage` computes the average pixel value in a gray-level or color image. For a color image, it computes the means of the three pixel color components, the variances of the components and the covariances between pairs of components.

AVERAGE, VARIANCE AND STANDARD DEVIATION

- `EasyImage::.PixelStat` computes min, max and average gray-level values.
- `EasyImage::.PixelVariance` computes average and variance of pixel values.
- `EasyImage::.PixelStdDev` computes average and standard deviation of pixel values. For a color image, it computes the standard deviations and correlation coefficients (covariance over the product of standard deviations) of the pairs of pixel component values.

NUMBER OF DIFFERENT PIXELS BY COMPARING 2 IMAGES

- `EasyImage::.PixelCompare` counts the number of different pixels between two images.