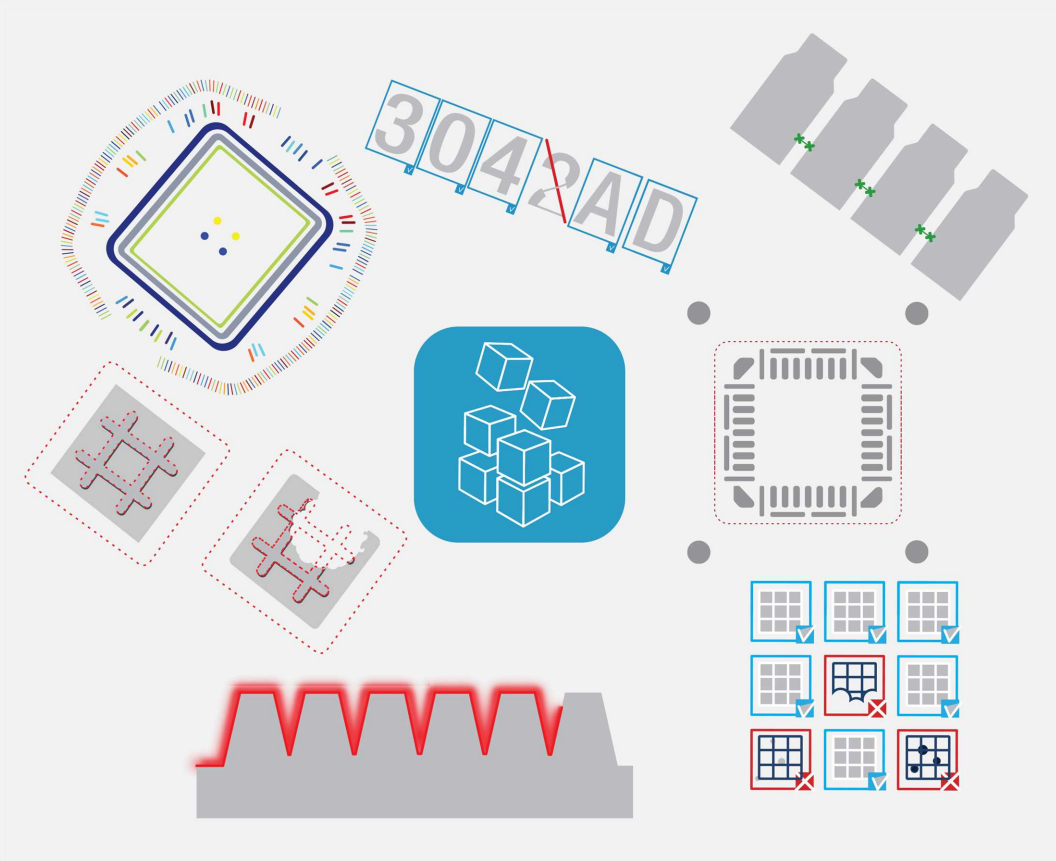


# Open eVision

## Matching and Measurement Tools



This documentation is provided with Open eVision 2.16.1 (doc build 1156).  
[www.euresys.com](http://www.euresys.com)

# Contents

1. Dealing with Pixel Containers and Files .....	6
1.1. Pixel Container Definition .....	6
1.2. Pixel Container Types .....	8
1.3. Supported Image File Types .....	9
1.4. Pixel and File Types Compatibility .....	10
1.5. Color Types .....	12
2. Manipulating Pixels Containers and Files .....	13
2.1. Pixel Container File Save .....	13
2.2. Pixel Container File Load .....	15
2.3. Memory Allocation .....	16
2.4. Image and Depth Map Buffer .....	17
2.5. Image Coordinate Systems .....	20
2.6. Image Drawing and Overlay .....	21
2.7. 3D Rendering of 2D Images .....	21
2.8. Vector Types and Main Properties .....	23
2.9. ROI Main Properties .....	27
2.10. Arbitrarily Shaped ROI (ERegion) .....	29
2.11. Flexible Masks .....	36
2.12. Profile .....	40
3. Matching and Measurement Tools .....	42
3.1. EasyObject - Analyzing Blobs .....	42
Image Segmenters .....	45
Image Encoder .....	49
Holes Construction .....	51
Normal vs. Continuous Mode .....	52
Selecting and Sorting Blobs .....	55
Object Template Matcher .....	56
Advanced Features .....	59
Computable Features .....	59
Draw Coded Elements .....	64
Flexible Masks in EasyObject .....	65
3.2. EasyGauge - Measuring down to Sub-Pixel .....	67
Workflow .....	67
Gauge Definitions .....	69
Find Transition Points Using Peak Analysis .....	72
Find Shapes Using Geometric Models .....	77
Gauge Manipulation: Draw, Drag, Plot, Group .....	79
Calibration and Transformation .....	80
Calibration Using EWorldShape .....	81
Advanced Features .....	84
Unwarp an Image .....	86
3.3. EasyFind - Matching Geometric Patterns .....	88
Workflow .....	88
Learning Process .....	91
Finding Process .....	92
Advanced Features .....	93
3.4. EasyMatch - Matching Area Patterns .....	94
Workflow .....	94
Learning Process .....	95
Matching Process .....	97
Advanced Features .....	97
3.5. EChecker2 - Validating Golden Templates .....	98
EChecker2 .....	98

Creating a Model .....	99
Inspecting an Image .....	102
4. Using Open eVision Studio .....	104
4.1. Selecting your Programming Language .....	104
4.2. Navigating the Interface .....	105
4.3. Running Tools on Images .....	106
Step 1: Selecting a Tool .....	106
Step 2: Opening an Image .....	107
Step 3: Managing ROIs .....	108
Step 4: Configuring the Tool .....	110
Step 5: Running the Tool and Checking Execution Time .....	111
Step 6: Using the Generated Code .....	113
4.4. Pre-Processing and Saving Images .....	114
5. Tutorials .....	116
5.1. EasyObject .....	116
Removing Non-Significant Objects After Image Segmentation .....	116
Detecting Differences Between Images Using Min-Max References .....	118
Detecting Printing Errors Using a Flexible Mask .....	119
5.2. EasyGauge .....	121
Measuring the Rotation Angle of an Object .....	121
Measuring the Diameter of a Circle .....	122
Measuring a Distorted Rectangle .....	124
Locating Points Regarding to a Coordinate System .....	126
Unwarping a Distorted Image .....	128
5.3. EasyFind .....	129
Detecting Highly-Degraded Occurrences of a Reference Model in Multiple Files .....	129
Improving the Score of Found Instances by Using "Don't Care Areas" .....	131
5.4. EasyMatch .....	134
Learning a Pattern and Creating an EasyMatch Model File .....	134
Matching a Pattern According to a Model File .....	134
Learning a Pattern According to an ROI .....	135
Improving the Score of Matching Instances by Using "Don't Care Areas" .....	137
6. Code Snippets .....	140
6.1. Basic Types .....	141
Loading and Saving Images .....	141
Interfacing Third-Party Images .....	141
Retrieving Pixel Values .....	142
ROI Placement .....	142
Vector Management .....	143
Exception Management .....	143
6.2. EasyObject .....	144
Constructing the Blobs .....	144
Image Encoder .....	144
Image Segmenter .....	144
Holes Extraction .....	145
Continuous Mode .....	146
Computing Blobs Features .....	146
Selecting and Sorting Blobs .....	147
Using Flexible Masks .....	147
Constructing Blobs .....	147
Generating a Flexible Mask from an Encoded Image .....	148
Generating a Flexible Mask from a Blob Selection .....	148
Using the Object Template Matcher .....	149
6.3. EasyGauge .....	151
Point Location .....	151
Line Fitting .....	151
Circle Fitting .....	152

Rectangle Fitting .....	153
Wedge Fitting .....	153
Gauge Grouping .....	154
Gauge Hierarchy .....	154
Complex Measurement .....	154
Calibration using EWorldShape .....	155
Calibration by Guesswork .....	155
Landmark-Based Calibration .....	156
Dot Grid-Based Calibration .....	156
Coordinates Transform .....	157
Image Unwarping .....	157
6.4. EasyFind .....	159
Pattern Learning .....	159
Setting Search Parameters .....	159
Pattern Finding and Retrieving Results .....	160
6.5. EasyMatch .....	161
Pattern Learning .....	161
Setting Search Parameters .....	161
Pattern Matching and Retrieving Results .....	162

# 1. Dealing with Pixel Containers and Files

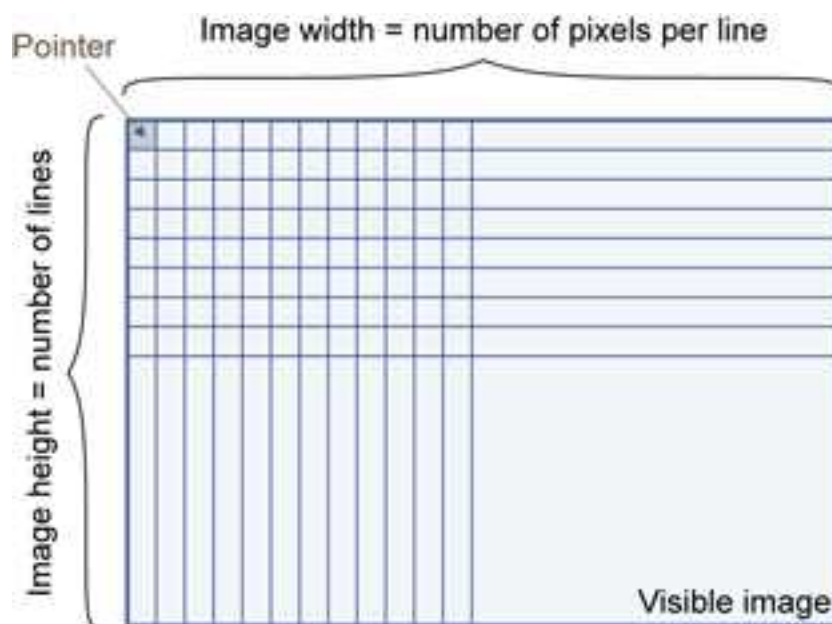
## 1.1. Pixel Container Definition

### 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 .

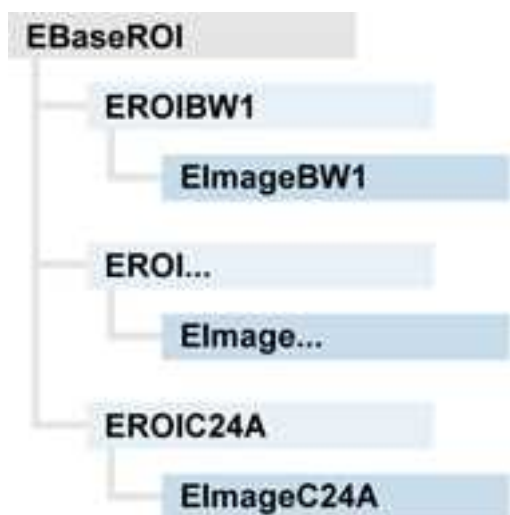
- **Width** is the number of columns (pixels) per row of the image.
- **Height** is the number of rows of the image. (Maximum width / height is 32,767 ( $2^{15}-1$ ) in Open eVision 32-bit, and 2,147,483,647 ( $2^{31}-1$ ) in Open eVision 64-bit.)
- **Size** is the width and height.

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

### Classes

---

Image and ROI classes derive from abstract class [EBaseROI](#) and inherit all its properties.



## Depth maps

---

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

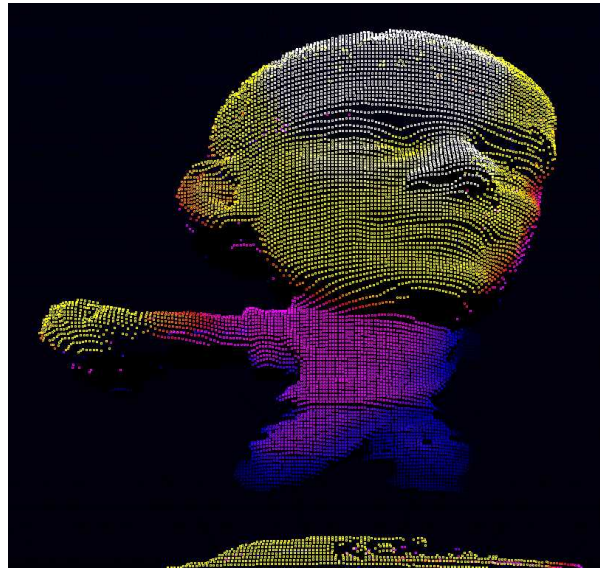


The pixel coordinates are the representation of the X and Y coordinates of the point while the grayscale value of the pixel is a representation of the Z coordinate of the point.

## Point clouds

---

A point cloud ([https://en.wikipedia.org/wiki/Point\\_cloud](https://en.wikipedia.org/wiki/Point_cloud)) is an unstructured set of 3D points representing discrete positions on 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.

## 1.2. Pixel Container Types

### Reference

### Images

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)	<a href="#">EImageBW</a>
BW8	8-bit grayscale images (each pixel is stored in 1 byte)	<a href="#">EImageBW8</a>
BW16	16-bit grayscale images (each pixel is stored in 2 bytes)	<a href="#">EImageBW16</a>
BW32	32-bit grayscale images (each pixel is stored in 4 bytes)	<a href="#">EImageBW32</a>
C15	15-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images and MultiCam RGB15 format.	<a href="#">EImageC15</a>



C16	16-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB16 color images and MultiCam RGB16 format.	<a href="#">EImageC16</a>
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.	<a href="#">EImageC24</a>
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.	<a href="#">EImageC24A</a>

## Depth Maps

8 and 16-bit depth map values are stored in buffers compatible with the 2D Open eVision images.

EDepth8	8-bit depth map (each pixel is stored in 1 byte as an integer)	<a href="#">EDepthMap8</a>
EDepth16	16-bit depth map (each pixel is stored in 2 bytes as a fixed point)	<a href="#">EDepthMap16</a>
EDepth32f	32-bit depth map (each pixel is stored in 4 bytes as a float)	<a href="#">EDepthMap32f</a>

## Point Clouds

Point Cloud	Set of points coordinates (stored as float)	<a href="#">EPointCloud</a>
-------------	---	-----------------------------

# 1.3. Supported Image File Types

### Reference

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 loses 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.

Type	Description
	<ul style="list-style-type: none"> <li>- <b>code stream</b> describes the image samples.</li> <li>- <b>file format</b> includes meta-information such as image resolution and color space.</li> </ul>
PNG	Lossless data compression method (Portable Network Graphics).
Serialized	Euresys proprietary image file format obtained from the <b>serialization</b> of Open eVision image objects.
TIFF	<p>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.</p> <p>File <b>save</b> operations are lossless and use CCITT 1D compression for 1-bit binary pixel types and LZW compression for all others.</p> <p>File <b>load</b> operations support all TIFF variants listed in the LibTIFF specification.</p>

## 1.4. Pixel and File Types Compatibility

### Depth map to image conversion

---

For 8- and 16-bit depth maps, the `AsImage()` method returns a compatible image object (respectively `EImageBW8` and `EImageBW16`) that can be used with Open eVision's 2D processing features.

### Pixel and file types compatibility

---

#### 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 images 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.

**TIP**

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

## 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`
- `EDepth8`, `EDepth16`, `EDepth32f`,

(\*) 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.

**NOTE**

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
Depth8	Ok	Ok	Ok	Ok	Ok	Ok

Type	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
Depth16	N/A	N/A	Ok	Ok	Ok (***)	Ok
Depth32f	N/A	N/A	N/A	N/A	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.

## 1.5. 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.

## 2. Manipulating Pixels Containers and Files

### 2.1. Pixel Container File Save

#### Images and depth maps

The `Save` method of an image or the `SaveImage` method of a depth map or a ZMap saves the image data of an image or of a depth map or a ZMap object into a file using two arguments:

- **Path:** path, file name 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



#### TIP

When saving a 16-bit depth map, the fixed point precision is lost and the pixels are considered as 16-bit integers.

#### Image file type arguments

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

## Saving point clouds

---

Use the following methods to save a point cloud in a specific format:

- `EPointCloud::Save`: Open eVision proprietary file format.
- `EPointCloud::SaveCSV`: CSV file.
- `EPointCloud::SaveOBJ`: OBJ file.
- `EPointCloud::SavePCD`: PCD file.
- `EPointCloud::SavePLY`: PLY file.
- `EPointCloud::SaveXYZ`: XYZ file.



### TIP

The PCD format is supported in ASCII and binary modes.

## 2.2. Pixel Container File Load

### Loading images and depth maps

---

- Use the `Load` method to load 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



### TIP

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



### TIP

When loading a BW16 image (with integer values) in a depth map, the fixed point precision set in the depth map (0 by default) is left unchanged and used.

## Loading point clouds

---

Use the following methods to load a point cloud saved in a specific format:

- `EPointCloud::Load`: Open eVision proprietary file format.
- `EPointCloud::LoadCSV`: CSV file.
- `EPointCloud::LoadOBJ`: OBJ file.
- `EPointCloud::LoadPCD`: PCD file.
- `EPointCloud::LoadPLY`: PLY file.
- `EPointCloud::LoadXYZ`: XYZ file.



### TIP

- The PCD format is supported in ASCII and binary modes.
- The PLY is supported only in ASCII mode.

## 2.3. Memory Allocation

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

### Internal memory allocation

---

The image object dynamically allocates and deallocates a buffer.

- Memory management is transparent.
- When the image size changes, reallocation occurs.
- When an image object is destroyed, the buffer is deallocated.

To declare an image with internal memory allocation:

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

- a. Declare an image object, for instance `EImageBW8`.
- b. Create a suitably sized and aligned buffer (see [Image Buffer](#)).
- c. Assign the buffer to the image with `SetImagePtr`.



#### NOTE

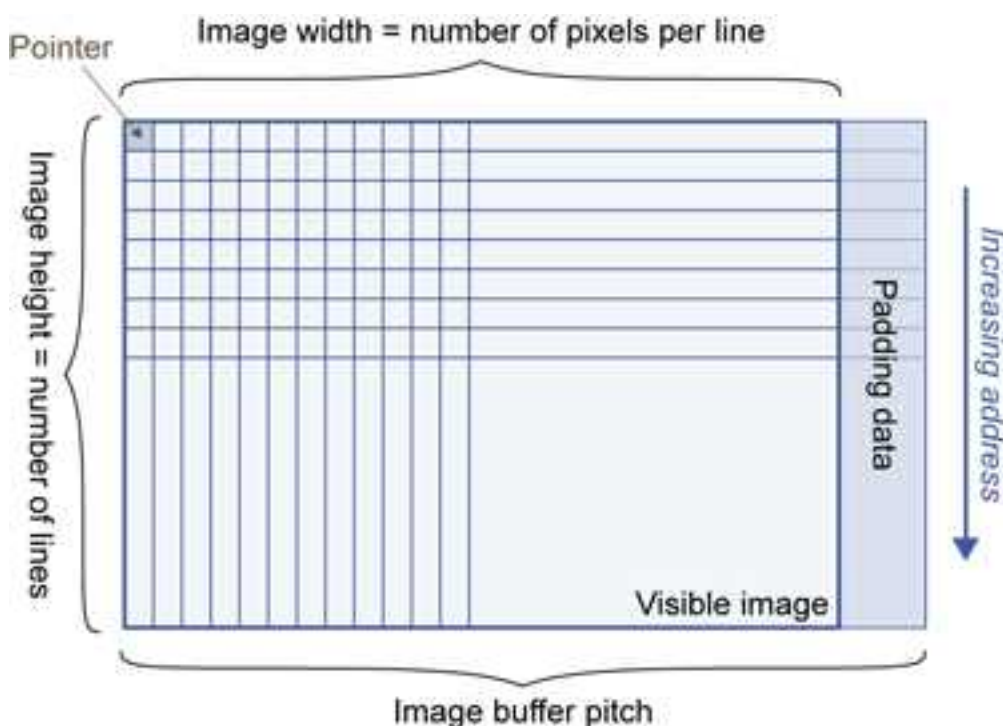
If your buffer rows are not aligned on 4 bytes, you cannot use `SetImagePtr`. In that case, use `InitializeFromUnalignedBuffer` instead.

Please note, however, that this allocates the memory internally and copies the external buffer into the internal one instead of using the external one directly.

## 2.4. Image and Depth Map Buffer

Image and depth map pixels are stored contiguously, from left to right and from top row to bottom row, in Windows bitmap format (top-down DIB -device-independent bitmap-) 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.



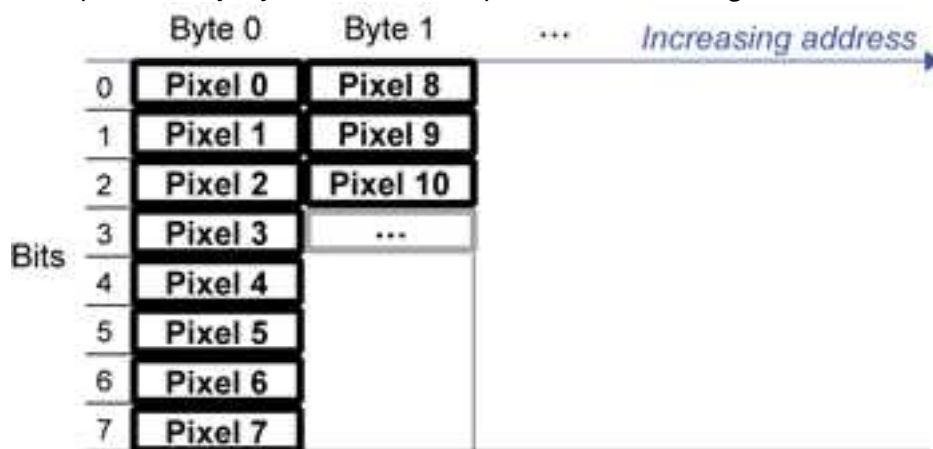
### Image buffer pitch

- 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

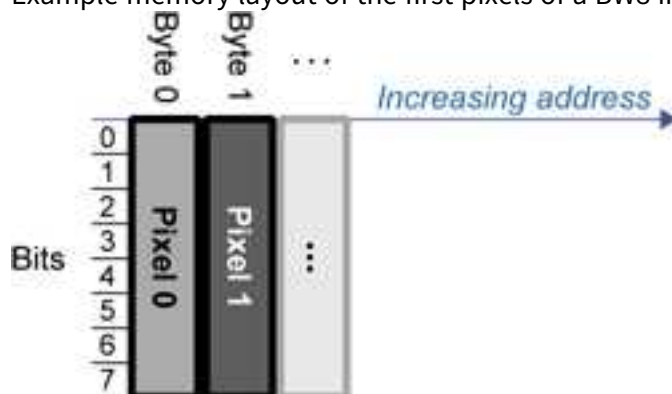
- `EImageBW_8` stores 8 pixels in one byte.

Example memory layout of the first 2 pixels of a BW1 image buffer:



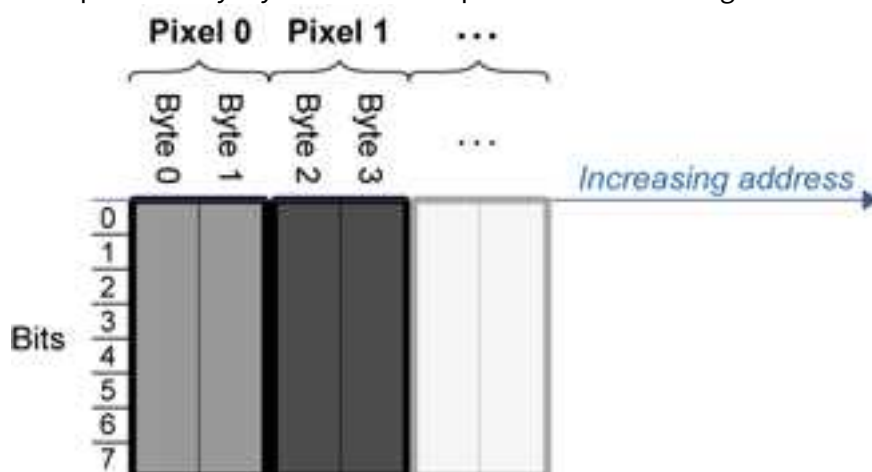
- `EImageBW_8` and `EDepthMap8` store each pixel in one byte.

Example memory layout of the first pixels of a BW8 image buffer:



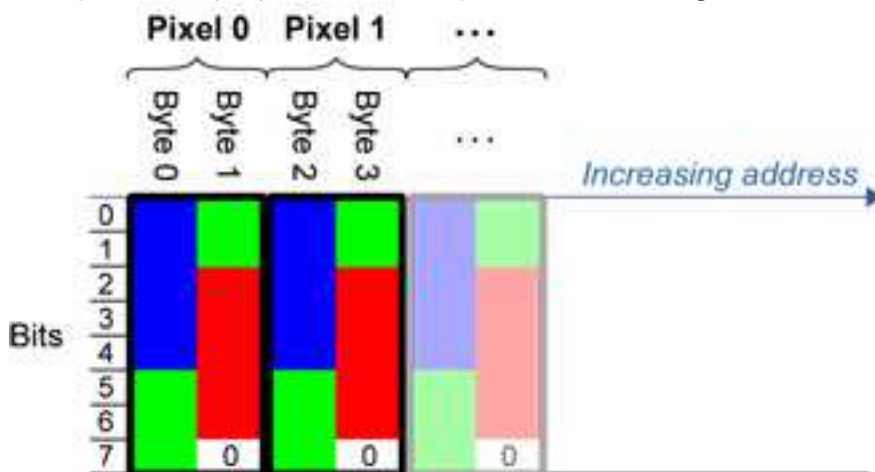
- `EImageBW_16` stores each pixel in a 16-bit word (two bytes).

Example memory layout of the first pixels of a BW16 image buffer:



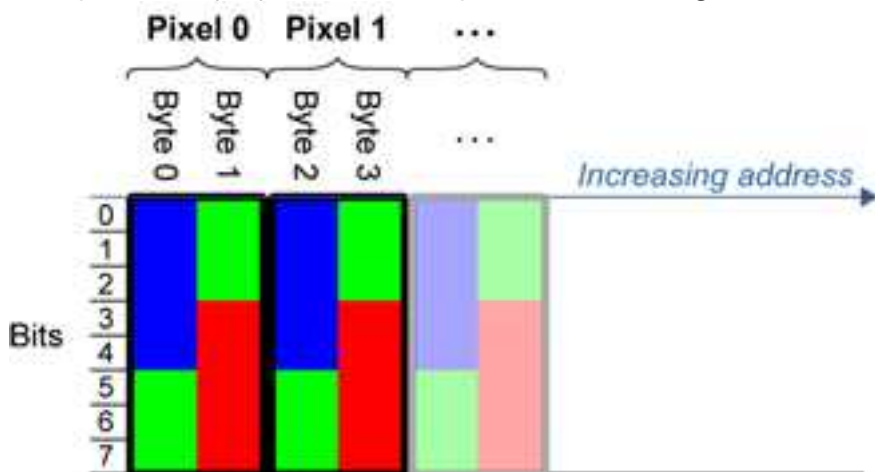
- `EImageC_5` stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

Example memory layout of the first pixels of a C15 image buffer:



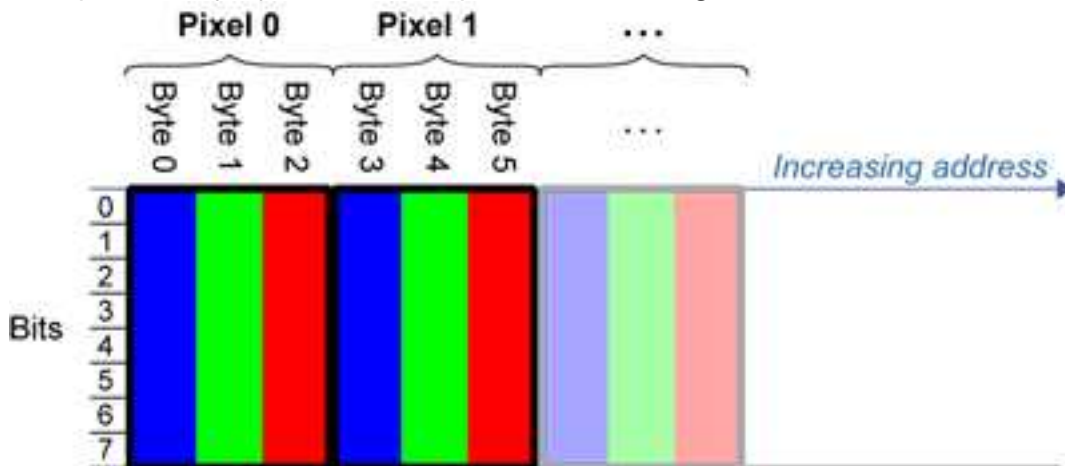
- [EImageC16](#) stores each pixel in 2 bytes. The first and third color components are coded with 5-bits. The second color component is coded with 6-bits.

Example memory layout of the first pixels of a C16 image buffer:



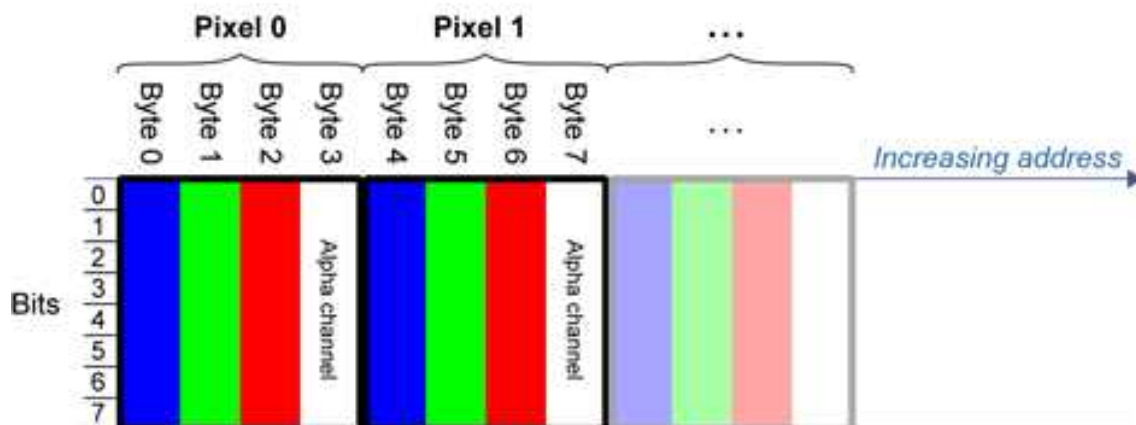
- [EDepthMap16](#) store each pixel in 2 bytes using a fixed point format.
- [EImageC24](#) stores each pixel in 3 bytes. Each color component is coded with 8-bits.

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:



- **EDepthMap32f** store each pixel in 4 bytes using a float format.

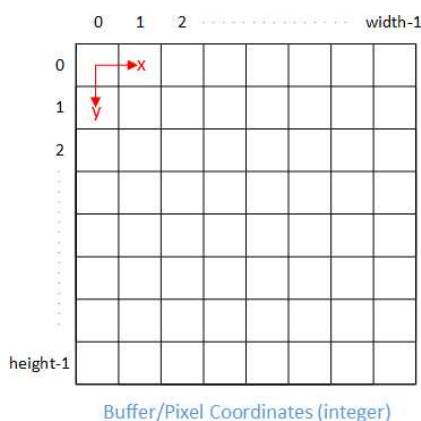
## 2.5. Image Coordinate Systems

The conventions below apply to all Open eVision functions and results.

- Pixel coordinates are usually given as integer numbers.
- Some results can use subpixel precision with real (floating point) numbers.
- Some exceptions apply and are documented per librarie.

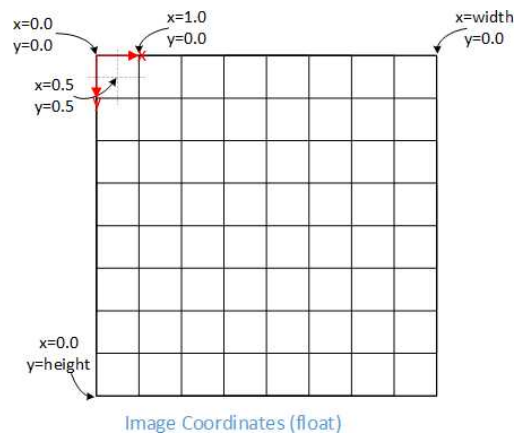
### Integer coordinates

- The origin (0,0) of the coordinate system is the upper left pixel of the image.
- The lower right pixel is (width-1, height-1).



## Real coordinates

- With floating point (x,y) coordinates, the origin is the upper left corner of the upper left pixel.
- The first pixel area ranges in  $[0,1[$  for X and Y axis.
- Coordinates greater or equal than the width or the height are outside the image.



## 2.6. Image Drawing and Overlay

- Drawing uses Windows GDI (Graphics Device Interface) system calls.
  - MFC (Microsoft Foundation Class) applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  - Borland/CodeGear 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 LUTs (LookUp Tables) (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.

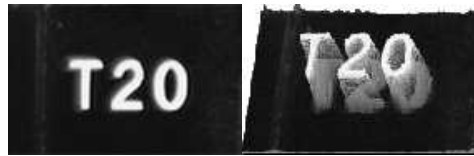
## 2.7. 3D Rendering of 2D Images

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 surface more or less opaque.



3D rendering

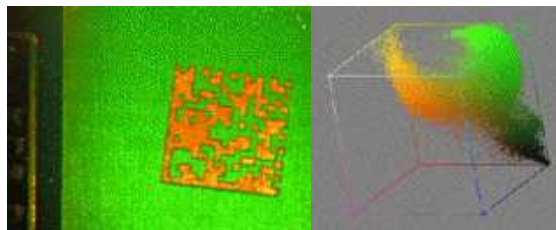
## Color histogram 3D rendering

---

**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.

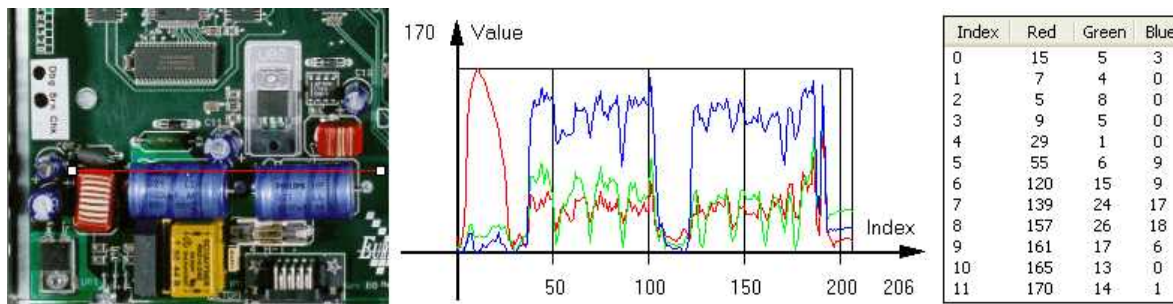


Color histogram rendering

## 2.8. 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 and 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.

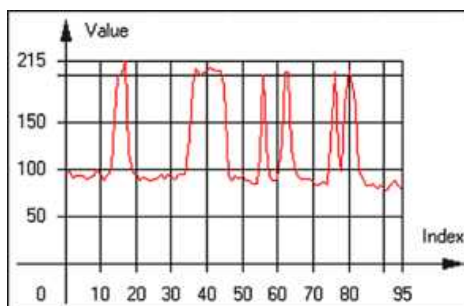
The use of vectors is quite straightforward:

- To create a vector of the appropriate type:
  - Use its constructor and preallocate elements if required.
- To fill a vector with values:
  - Call the [EVector::Empty](#) member to empty it.
  - Call the [EC24Vector::AddElement](#) member to add elements one by one.
  - Use the indexing to access any element.
- To access a vector element, either for reading or writing:
  - Use the brackets operator [EC24Vector::operator\[\]](#).
- To determine the current number of elements:
  - Use the [EVector::NumElements](#) member.
- To 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 and annotations in black. You can define: `graphicContext`, `width`, `height`, `originX`, `originY`, `color0`, `color1` and `color2`.
  - The [EC24Vector](#) has three curves drawn instead of one, each corresponding to a color component. By default the red, blue and green pens are used.



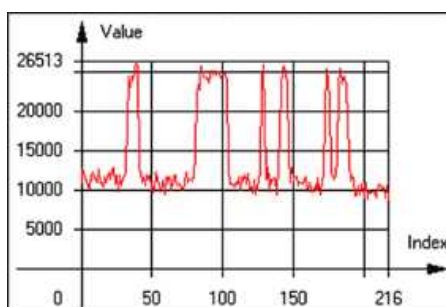
## Vector types

- **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`...).



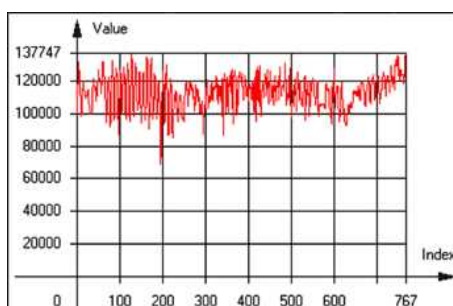
Graphical representation of an **EBW8Vector** (see [Draw](#) method)

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



Graphical representation of an **EBW16Vector**

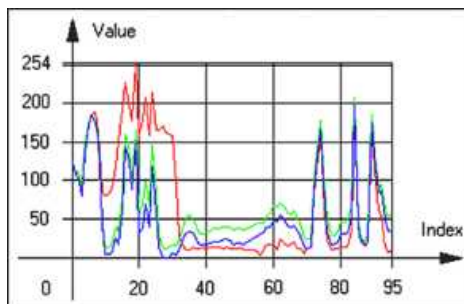
- **EBW32Vector**: a sequence of gray-level pixel values, using an extended range (32 bits), mainly for intermediate computations (used in `EasyImage::ProjectOnARow`, `EasyImage::ProjectOnAColumn`, ...).



Graphical representation of an **EBW32Vector**

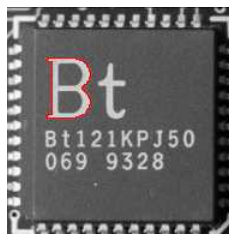


- **EC24Vector**: a sequence of color pixel values, often extracted from an image profile (used by `EasyImage::ImageToLineSegment`, `EasyImage::LineSegmentToImage`, `EasyImage::ProfileDerivative`, ...).



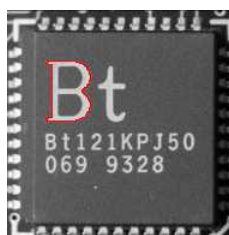
Graphical representation of an **EC24Vector**

- **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`, ...).



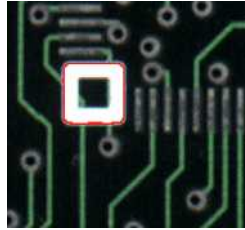
Graphical representation of an **EBW8PathVector** (see `Draw` method)

- **EBW-6PathVector**: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).



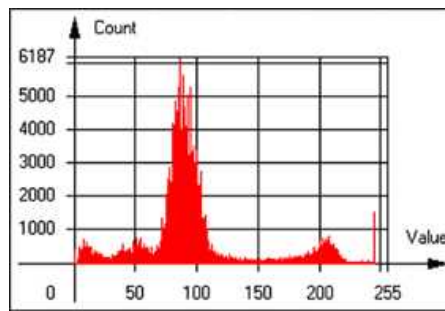
Graphical representation of an **EBW-6PathVector** (see `Draw` method)

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



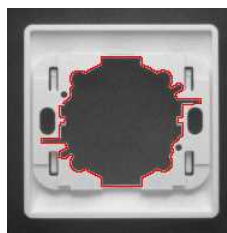
Graphical representation of an **EC24PathVector** (see [Draw method](#))

- **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`).

## 2.9. 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](#).

- [EROIBW](#)
- [EROIBW8](#)
- [EROIBW-6](#)
- [EROIBW32](#)
- [EROIC-5](#)
- [EROIC-6](#)
- [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.

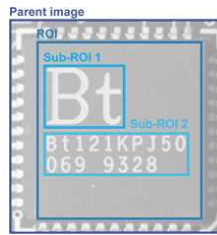
A normal image cannot be attached to another image or ROI.

### 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.



### NOTE

*(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).

## 2.10. Arbitrarily Shaped ROI (ERegion)

**See also:** [example: Inspecting Pads Using Regions](#) / [code snippets: ERegion](#)

### Regions or arbitrarily shaped ROI

You define and use regions of interest (ROI) to restrict the area processed with your vision tool and to reduce and optimize the processing time.

In Open eVision:

- An **ROI** (`EROIxxx` class) designates a rectangular region of interest.
- A **region** (`ERegion` class) designates an arbitrarily shaped ROI. With regions, you can determine precisely which part of the image, down to a single pixel, is used for your processing.

Currently, only the following Open eVision methods support `ERegions`:

Library	Method
	<code>EasyImage::Threshold</code>
	<code>EasyImage::Copy</code>
	<code>EasyImage::ConvolKernel</code>
	<code>EasyImage::ConvolSymmetricKernel</code>
	<code>EasyImage::ConvolLowpass</code>
	<code>EasyImage::ConvolLowpass2</code>
	<code>EasyImage::ConvolLowpass3</code>
	<code>EasyImage::ConvolUniform</code>
	<code>EasyImage::ConvolGaussian</code>
	<code>EasyImage::ConvolHighpass</code>
	<code>EasyImage::ConvolHighpass2</code>
	<code>EasyImage::ConvolGradientX</code>
	<code>EasyImage::ConvolGradientY</code>
	<code>EasyImage::ConvolGradient</code>
	<code>EasyImage::ConvolSobelX</code>
	<code>EasyImage::ConvolSobelY</code>
	<code>EasyImage::ConvolSobel</code>
	<code>EasyImage::ConvolPrewittX</code>
	<code>EasyImage::ConvolPrewittY</code>
	<code>EasyImage::ConvolPrewitt</code>
	<code>EasyImage::ConvolRoberts</code>
	<code>EasyImage::ConvolLaplacianX</code>
	<code>EasyImage::ConvolLaplacianY</code>
	<code>EasyImage::ConvolLaplacian8</code>
	<code>EasyImage::DilateBox</code>
	<code>EasyImage::ErodeBox</code>
	<code>EasyImage::OpenBox</code>
	<code>EasyImage::CloseBox</code>
	<code>EasyImage::WhiteTopHatBox</code>
	<code>EasyImage::BlackTopHatBox</code>
	<code>EasyImage::MorphoGradientBox</code>
	<code>EasyImage::ErodeDisk</code>
EasyImage	

Library	Method
	EasyImage::DilateDisk
	EasyImage::OpenDisk
	EasyImage::CloseDisk
	EasyImage::WhiteTopHatDisk
	EasyImage::BlackTopHatDisk
	EasyImage::MorphoGradientDisk
	EasyImage::Median
	EasyImage::ScaleRotate
	EasyImage::DoubleThreshold
	EasyImage::Histogram
	EasyImage::Area
	EasyImage::AreaDoubleThreshold
	EasyImage::BinaryMoments
	EasyImage::WeightedMoments
	EasyImage::GravityCenter
	EasyImage::PixelCount
	EasyImage::PixelMax
	EasyImage::PixelMin
	EasyImage::PixelAverage
	EasyImage::PixelStat
	EasyImage::PixelVariance
	EasyImage::PixelStdDev
	EasyImage::PixelCompare
Easy3D	EDepthMapToMeshConverter::Convert
	EDepthMapToPointCloudConverter::Convert
	EStatistics::ComputePixelStatistics
	EStatistics::ComputeStatistics
	E3DObjectExtractor::Extract
	EZMapToPointCloudConverter::Convert
EasyObject	EImageEncoder::Encode
EasyFind	EPatternFinder::Find
	EPatternFinder::Learn
EasyOCR2	EOCR2::Read
	EOCR2::Detect
EasyGauge	EPointGauge::Measure
	ELineGauge::Measure
	ERectangleGauge::Measure
	ECircleGauge::Measure
	EWedgeGauge::Measure
EasyMatch	EMatcher::LearnPattern
	EMatcher::Match
EasyQRCode	EQRCodeReader::SetSearchField
	EQRCodeReader::Read



**TIP**

In the future Open eVision releases, the support of **ERegions** will be gradually extended to all operators.

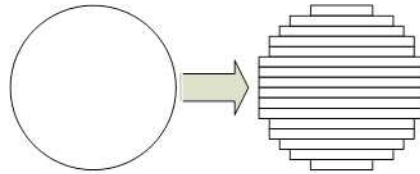
## Creating regions

---

Open eVision offers multiple ways to create regions, depending on the shape you need:

The [ERegion](#) is the base class for all regions and the most versatile. It encodes a region using a Run-Length Encoded (RLE) representation.

- The RLE representation of a region is made of runs (horizontal, 1-pixel high slices).
- The runs are stored in the form of their ordinate, starting abscissa and length.



Run-Length Encoding of a circle-shaped region

To create a region, either:

- Use one of the geometry-based region classes.
- Use the result of another tool, such as EasyFind, EasyMatch or EasyObject.
- Combine or modify other regions.
- Use a mask image.
- Directly provide the list of runs.

### Geometry-based regions

Geometry based regions are specialized classes of regions that are encompassed in simple geometries. Open eVision currently provides classes based on a rectangle, a circle, an ellipse or a polygon.

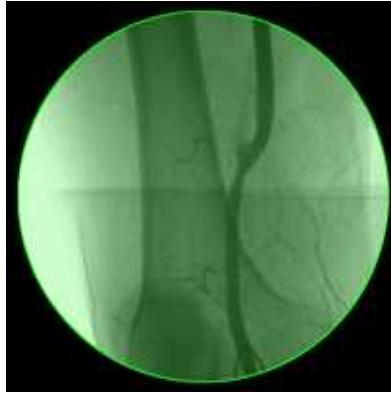
Use these classes to setup geometric regions and modify them with translation, rotation and scaling. The transformation operators return new regions, leaving the source object unchanged.

- [ERectangleRegion](#)
  - The contour of an [ERectangleRegion](#) class is a rectangle.
  - Define it using its center, width, height and angle.
  - Alternatively, use an [ERectangle](#) instance, such as one returned by an [ERectangleGauge](#) instance.



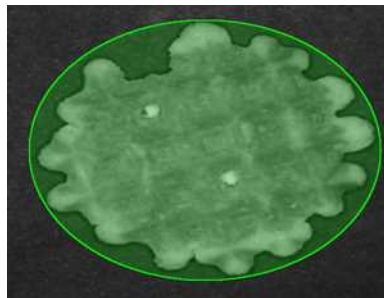
Rectangle region separating a bar code from the background

- **ECircleRegion**
  - The contour of an **ECircleRegion** class is a circle.
  - Define it using its center and radius or 3 non-aligned points.
  - Alternatively, use an **ECircle** instance, such as one returned by an **ECircleGauge** instance.



Circle region encompassing the useful part of an X-Ray image

- **EEllipseRegion**
  - The contour of an **EEllipseRegion** class is an ellipse.
  - Define it using its center, long and short radius and angle.



Ellipse region encompassing a waffle

- **EPolygonRegion**
  - The contour of an **EPolygonRegion** class is a polygon.
  - It is constructed using the list of its vertices.



Polygon region encompassing a key



## Using the result of other tools

The `ERegion` class provides a set of specialized constructors to create regions from the results of another tool.

In a tool chain, these constructors restrict the processing of a tool to the area issued from the previous tool.



Open eVision provides constructors for the following tools:

- EasyFind: `EFoundPattern`
- EasyMatch: `EMatchPosition`
- EasyGauge: `ECircle` and `ERectangle`
- EasyObject: `ECodedElement`



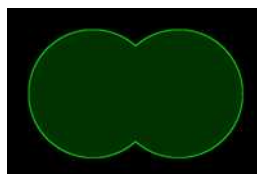
### TIP

When compatible, Open eVision also provides specialized constructors for the geometry-based regions. For instance, `ECircleRegion` provides a constructor using an `ECircle`.

## Combining regions

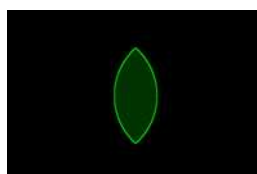
Use the following operations to create a new region by combining existing regions:

- Union
  - The `ERegion::Union(const ERegion&, const ERegion&)` method returns the region that is the addition of the two regions passed as arguments.



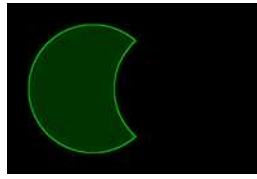
Union of 2 circles

- Intersection
  - The `ERegion::Intersection(const ERegion&, const ERegion&)` method returns the region that is the intersection of the two regions passed as argument.



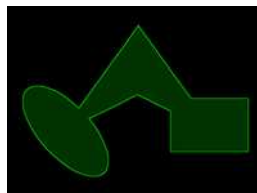
Intersection of 2 circles

- Subtraction
  - The `ERegion::Subtraction(const ERegion&, const ERegion&)` method returns the first region passed as argument after removing the second one.



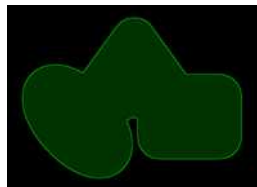
Subtraction of 2 circles

### Morphological operations on regions



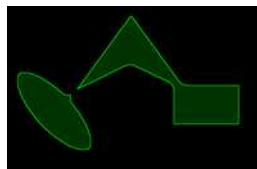
The initial arbitrary region used to illustrate the different morphological operations

- Grow
  - The `ERegion::Grow(int radius)` method returns a region that is the dilation of the region by a disk with a radius equals to the argument.



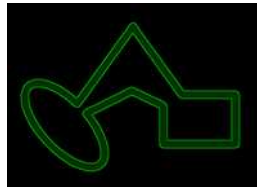
Grow of the arbitrary region

- Shrink
  - The `ERegion::Shrink(int radius)` method returns a region that is the erosion of the region by a disk with a radius equals to the argument.



Shrink of the arbitrary region

- Contour
  - The `ERegion::Contour(int thickness, bool centered = true)` method returns a region that is the contour of the region.



Contour of the arbitrary region

### Free-hand drawing a region

- The `ERegionFreeHandPainter` class provides the methods that allow you to create a region by hand, using the mouse or any other user input method.
- The `RegionFreeHand` sample, available both in C++ and C#, shows how to use this class to draw a region on an image.

## Using regions

---

The tools supporting regions provide methods that follow one of these conventions:

- `Method(const EImage& source, const ERegion& region)`
- `Method(const EImage& source, const ERegion& region, EImage& destination)`



### NOTE

The source, the region and the destination must be compatible. It means that the region must at least partly fit in the source, and that source and destination must have the same size.

### Preparing the region

- Open eVision automatically prepares the regions when it applies them to an image, but this preparation can take some time.
- If you do not want your first call to a method to take longer than the next ones, you can prepare the region in advance by using the appropriate `Prepare()` method.
- To manually prepare the regions, adapt the internal RLE description to your images.

## Drawing regions

---

The `ERegion` classes provide several methods to display the regions:

- `ERegion::Draw()` draws the region area, in a semi-transparent way, in the provided device context.
- `ERegion::DrawContour()` draws the region contour in the provided device context.

- `ERegion::ToImage()` renders the region as a mask into the provided destination image.
  - You can configure the foreground and the background colors.
  - If you initialized your image with a width and a height, Open eVision renders the region inside those bounds.
  - If not, Open eVision resizes the image to contain the whole region.
  - Use `ToImage()` to create masks for the Open eVision functions that support them.

## ERegions and EROIs

---

- The older `EROI` classes of Open eVision are compatible with the new regions.
- Some tools allow the usage of regions with source and/or destinations that are `ERoi` instead of `EImage` follow one of these conventions:
  - `Method(const ERoi& source, const ERegion& region)`
  - `Method(const ERoi& source, const ERegion& region, ERoi& destination)`



### TIP

In that case, the coordinates used for the region are relative to the reduced ROI space instead of the whole image space .

## ERegion and 3D

---

- The new regions are compatible with the 2.5D representations of Easy3D (`EDepthMap` and `EZMap`).
- You can also reduce the domain of processing when using these classes.

# 2.11. Flexible Masks

## ROIs vs flexible masks

---

ROIs and masks restrict processing to part of an image:

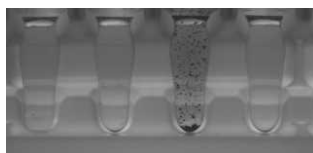
- "ROI Main Properties" on page 27 apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
- Flexible Masks are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some `EasyObject` and `EasyImage` library functions.

## Flexible Masks

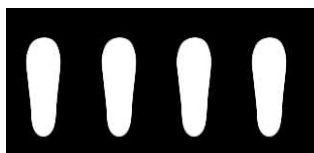
---

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):

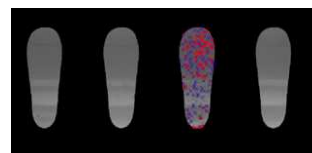
- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image



Associated mask



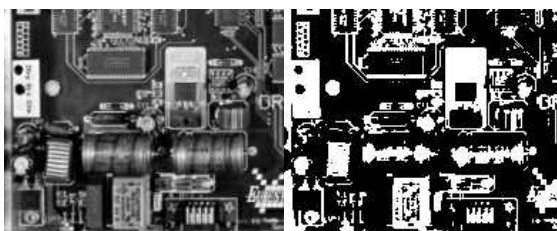
Processed masked image

A flexible mask can be generated by any application that outputs BW8 images and by some [EasyObject](#) and [EasyImage](#) functions.

## Flexible Masks in EasyImage

---

### Code Snippets

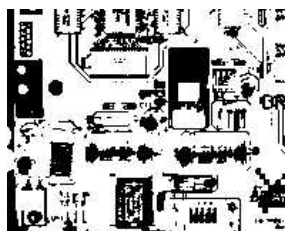


Source image (left) and mask variable (right)

## 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.

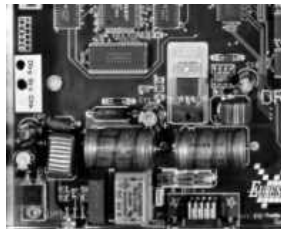
If an object of interest has the same gray level as other regions of the image, you can define "keep" and "ignore" areas using flexible masks and `Encode` functions.

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

---



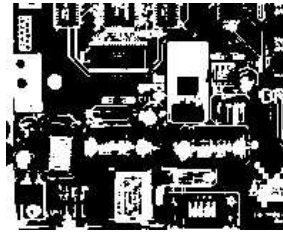
Source image

### 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`.

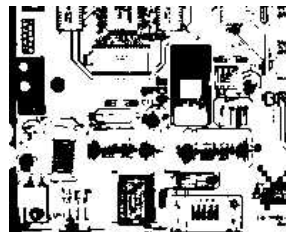


BW8 resulting image that can be used as a flexible mask

## 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.

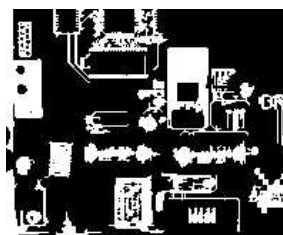


BW8 resulting image that can be used as a flexible mask

## 3) `EObjectSelection::RenderMask`: from a selection of blobs

---

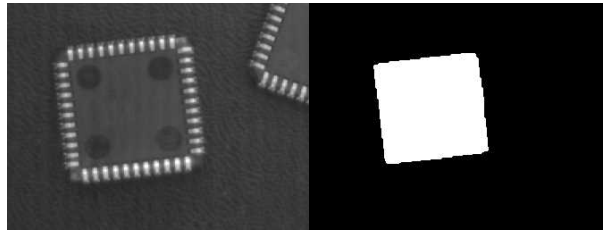
`EObjectSelection::RenderMask` can, for example, discard small objects resulting from noise.



BW8 resulting image that can be used as a flexible mask

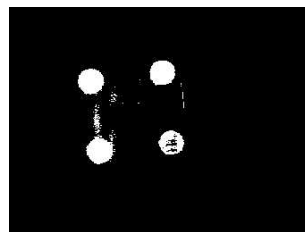
## Example: Restrict the areas encoded by EasyObject

---



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.

## 2.12. Profile

### Code Snippets

#### 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 masks.
- 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 masks.



- 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).

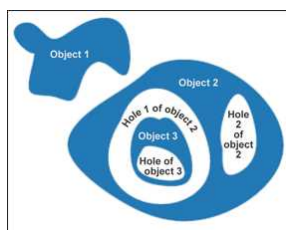
`EasyImage::PathToImage` copies the pixel values from a vector or a constant to the pixels of a given path.

## 3. Matching and Measurement Tools

### 3.1. EasyObject - Analyzing Blobs

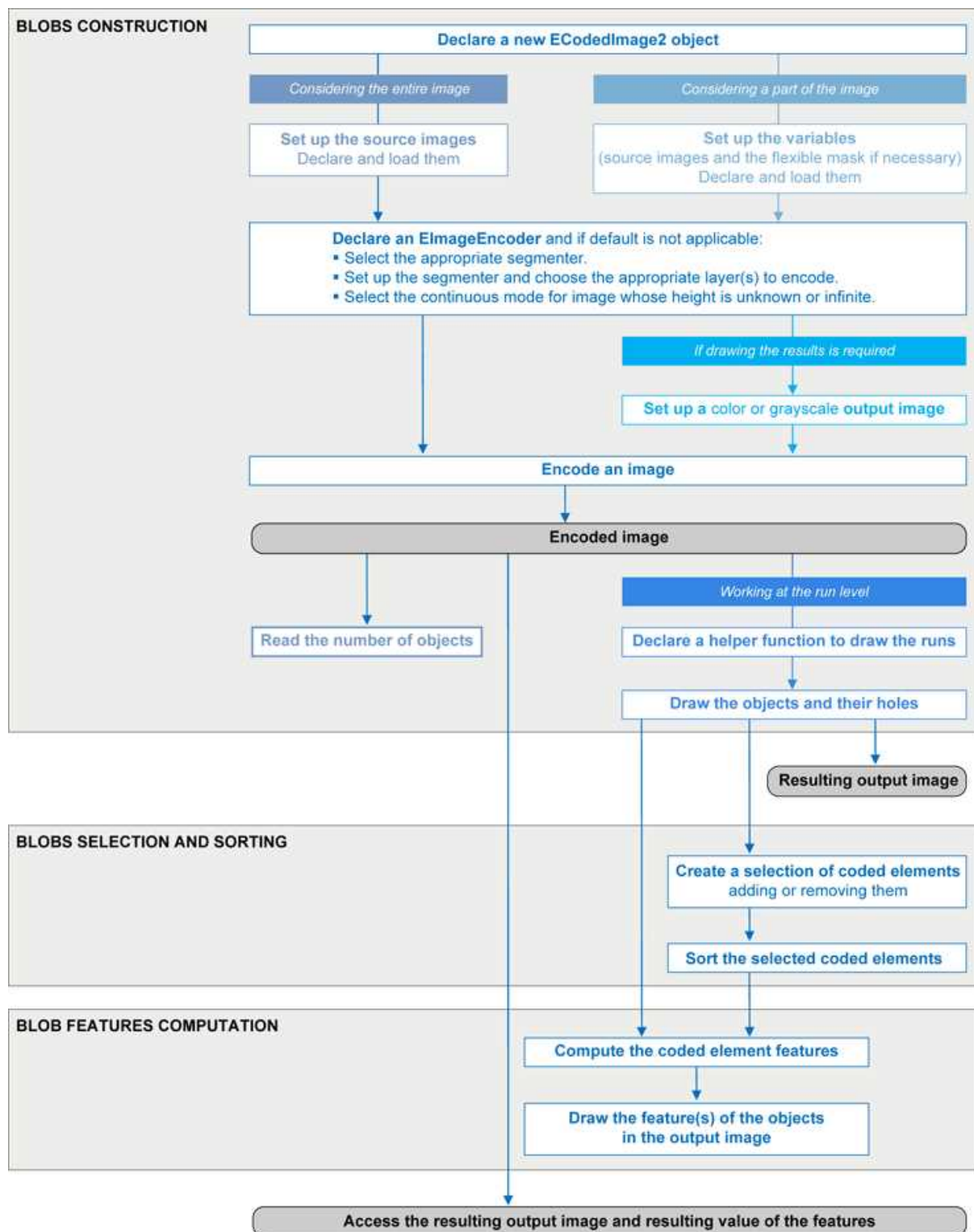
[Reference](#) | [Code Snippets](#)

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

### Code Snippets

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:

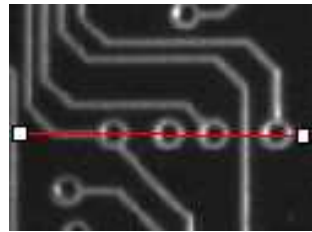
- when the background is not uniform enough,
- when the illumination is not uniform across the image,
- when only differences between the image and a reference image (ideal) are to be enhanced,

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.

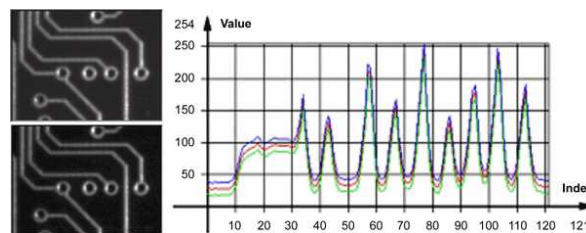
## Preparing high and low reference images

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



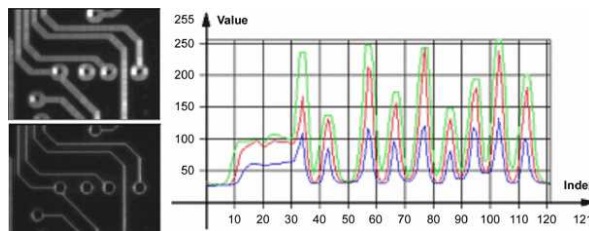
Source image

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



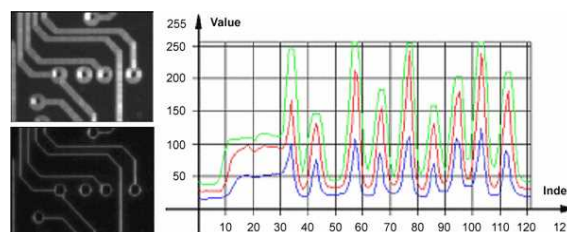
Gray-level tolerance margins

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 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: [SetLowImageBW-6](#) [GetLowImageBW-6](#)
- C24: [SetLowImageC24](#) [GetLowImageC24](#)

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

- BW8: [SetHighImageBW8](#) [GetHighImageBW8](#)
- BW16: [SetHighImageBW-6](#) [GetHighImageBW-6](#)
- C24: [SetHighImageC24](#) [GetHighImageC24](#)

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: [SetReferenceImageBW-6](#), [GetReferenceImageBW-6](#)
- 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:

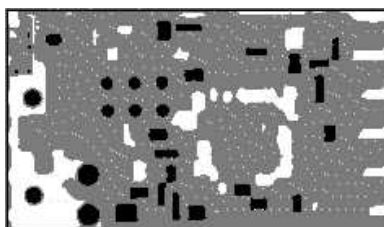
- Black layer (usually index 0) contains unmasked pixels with a binary value equal to zero.
- White layer (usually index 1) contains the remaining unmasked pixels, i.e. unmasked pixels with a binary value equal to one.



# Image Encoder

[Reference](#) | [Code Snippets](#)

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	<code>SetBlackLayerEncoded</code>	<code>IsBlackLayerEncoded</code>	<b>FALSE</b>
White layer	<code>SetWhiteLayerEncoded</code>	<code>IsWhiteLayerEncoded</code>	<b>TRUE</b>

Three-layer segmenters

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

## 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

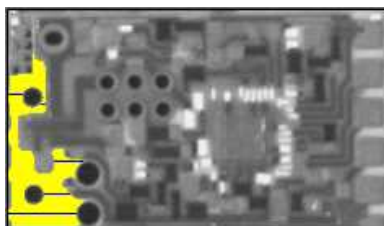
Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerIndex	IsBlackLayerIndex	0
White layer	SetWhiteLayerIndex	IsWhiteLayerIndex	1

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

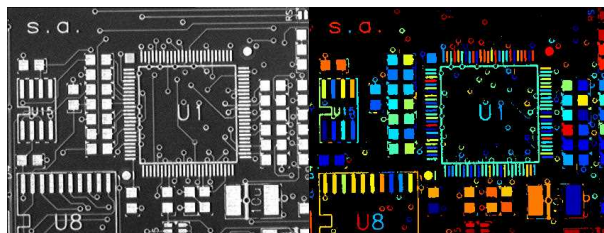
For the sake of computational efficiency, the objects are described as lists of runs. A run is a sequence of adjacent pixels that share homogeneous properties (such as being above a given threshold). These runs are merged in objects by the image encoder.



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. **Set up 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](#).

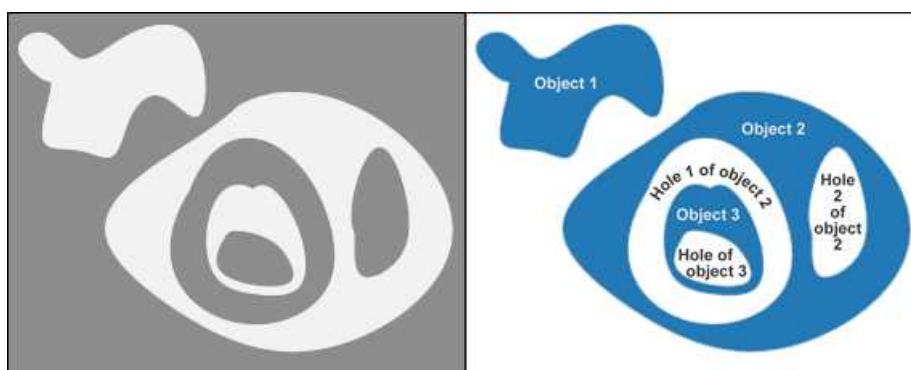
## Holes Construction

### Code Snippets

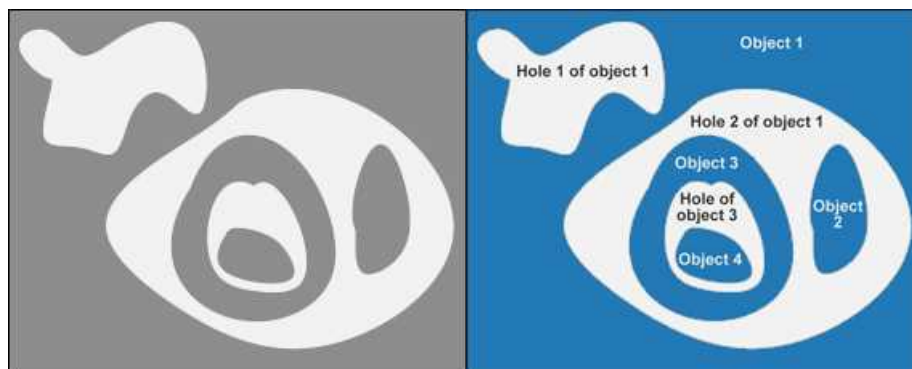
A hole is a set of connected pixels that are entirely surrounded by a parent object (4 or 8 pixels depending on the connexity mode).

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.



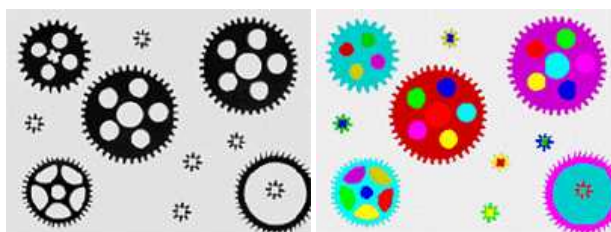
Encoding the white layer (3 objects and 3 holes)



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. **Set up 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`) that returns a color. This color 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

### Code Snippets

#### 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

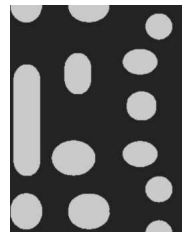
In continuous mode EasyObject can process an image whose height is unknown or infinite (e.g. coming from a line-scan camera). The image is split into several chunks that are fed into an image encoder. Objects that straddle several successive image chunks can be detected.

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 into 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

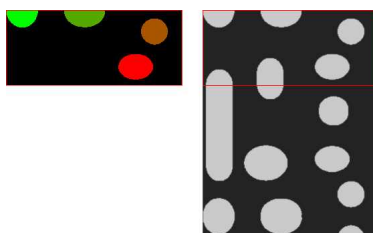


Three chunks of the image

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

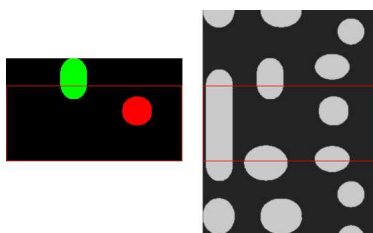
function closely follows the function `DrawRuns`, but is adapted to continuous mode by taking `GetStartY` into account.

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.

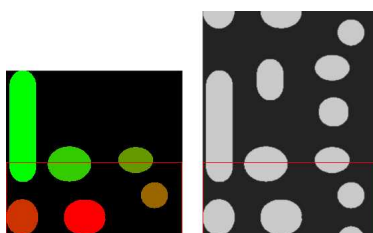
Note that two objects in the lower-left of the image chunk are not encoded, because they touch the border of the chunk.



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.

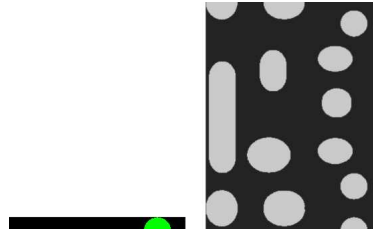
Here is the result of the encoding of the last chunk (`count = 2`).



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

### Code Snippets

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 class [EObjectSelection](#).

### Create / modify a selection

---

You can use the methods [Add](#) and [Remove](#) of the class [EObjectSelection](#) 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.
- Add or remove objects based on their specific position, or whether they lie outside, on or within a specified ROI rectangle or [ERegion](#) ([AddObjectsUsingRectangle](#) and [AddObjectsUsingRegion](#)).

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

### Clear a selection

---

You can clear a previous selection using [EObjectSelection::Clear](#).

### Sort a selection

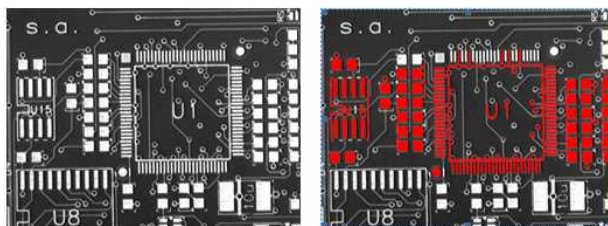
---

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.

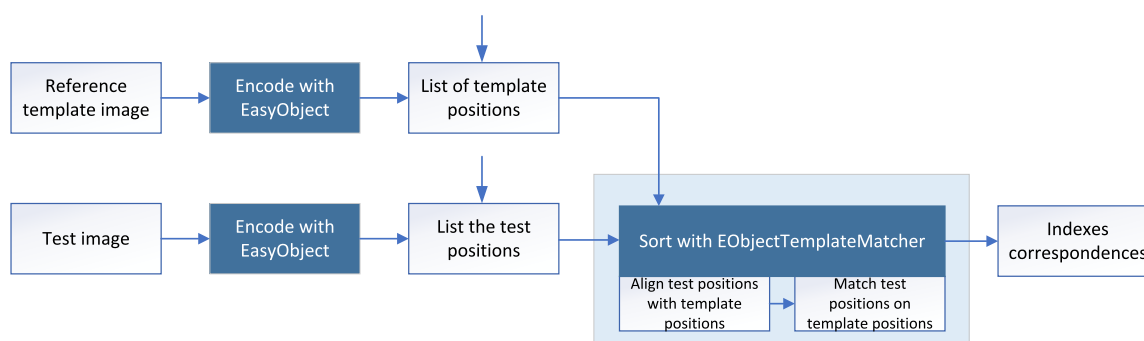


Source image, and selection of objects

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.

## Object Template Matcher

The class `EObjectTemplateMatcher` is a tool designed to align and match the output of `EasyObject` to a reference template. It is designed and developed to handle efficiently thousand of objects.

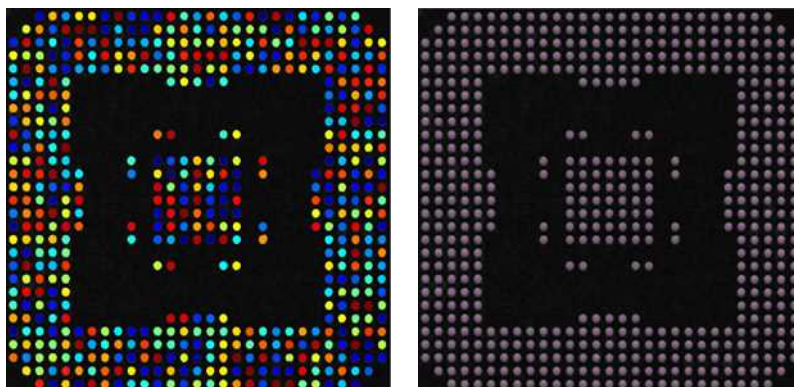




## Creating the reference template

Use the method `BuildTemplate` to create the reference template, with one of the following parameters:

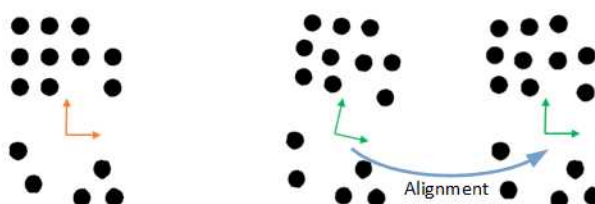
- An `ECodedImage2`, result of the method `EImageEncoder::Encode`.
- An `EObjectSelection`, a selection (subset) of `ECodedImage2` objects.
- A list of positions, given by a vector of points (`std::vector<EPoint>`).



An encoding of the reference image for use as the template.  
And the center positions of each object for use in the matching process.

## Sorting the objects

- To perform the matching after setting up the template:
  - Use the method `SortSelection` with an `EObjectSelection` as parameter.
  - Or use the more generic method `SortPositions` with a `std::vector<EPoint>` as parameter.
- When you pass the objects in a selection as the sort method parameter, the bounding box center of the objects is the position used for the matching with the template.
- Before the sorting, `EObjectTemplateMatcher` performs an optional global rigid alignment of the submitted positions with the defined template.
  - This alignment only applies the translation and rotation transformations.
  - Use the method `SetEnableAlignment` to enable the alignment process.



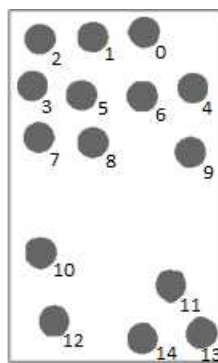
Left: the template.  
Right: the alignment of the submitted selection.

- After the optional alignment, [EObjectTemplateMatcher](#) matches the submitted positions with the reference template.
  - It uses the shortest distance criterion to pair these positions with the template.
  - You can set the maximum distance to constraint the search. This can speed up the processing.

## Retrieving the Sorting Results

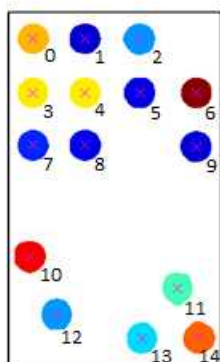
---

- Use one of these methods to retrieve the sorting results:
  - [GetSelectionIndexes](#) returns, for each position in the template, the paired index in the selection. The value `-1` is used if the object in the template has no correspondence in the selection.
  - [GetTemplateIndexes](#) returns, for each position in the selection, the paired index in the template. The value `-1` is used if the object in the selection has no correspondence in the template.
  - [GetUnpairedObjects](#) returns the positions in the template and in the selection that have not been paired.
- Use the method [GetNumberOfPairedObjects](#) to get the total number of paired objects.
- Use the methods [Save](#) and [Load](#) to store and retrieve the configuration of an [EObjectTemplateMatcher](#) object, including the template.

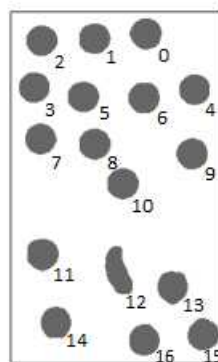


```
NumberOfPairedObjects = 15
SelectionIndexes =
2, 1, 0, 3, 5, 6, 4, 7, 8, 9, 10, 11, 12, 14, 13
TemplateIndexes =
2, 1, 0, 3, 6, 4, 5, 7, 8, 9, 10, 11, 12, 14, 13
UnpairedObjects =
  onlyInTemplate = <>
  onlyInSelection = <>
```

Selection with moved objects

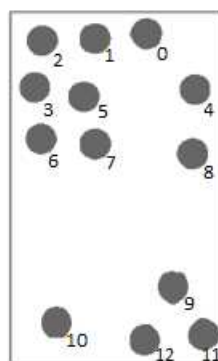


Template with object indexes



```
NumberOfPairedObjects = 15
SelectionIndexes =
2, 1, 0, 3, 5, 6, 4, 7, 8, 9, 11, 13, 14, 16, 15
TemplateIndexes =
2, 1, 0, 3, 6, 4, 5, 7, 8, 9, -1, 10, -1, 11, 13, 14, 13
UnpairedObjects =
  onlyInTemplate = <>
  onlyInSelection = 10, 12
```

Selection with extra objects



```
NumberOfPairedObjects = 13
SelectionIndexes =
2, 1, 0, 3, 5, -1, 4, 6, 7, 8, -1, 9, 10, 11, 12
TemplateIndexes =
2, 1, 0, 3, 6, 4, 7, 8, 9, 11, 12, 14, 13
UnpairedObjects =
  onlyInTemplate = 5, 10
  onlyInSelection = <>
```

Selection with missing 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)	<a href="#">ECodedElement::GetTopLimit</a> <a href="#">ECodedElement::GetBottomLimit</a> <a href="#">ECodedElement::GetLeftLimit</a> <a href="#">ECodedElement::GetRightLimit</a>
Gravity center (X and Y)	<a href="#">ECodedElement::GetGravityCenter</a> <a href="#">ECodedElement::GetGravityCenterX</a> <a href="#">ECodedElement::GetGravityCenterY</a>
Weight gravity center (X and Y)	<a href="#">ECodedElement::ComputeWeightedGravityCenter</a>

## 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)	<a href="#">ECodedElement::Area</a>
Feret box (center X and Y, height, width with distinct orientation angles at 22, 45, 68 degrees)	<a href="#">ECodedElement::ComputeFeretBox</a> <a href="#">ECodedElement::GetFeretBox22Box</a> <a href="#">ECodedElement::GetFeretBox22Center</a> <a href="#">ECodedElement::GetFeretBox22CenterX</a> <a href="#">ECodedElement::GetFeretBox22CenterY</a> <a href="#">ECodedElement::GetFeretBox22Height</a> <a href="#">ECodedElement::GetFeretBox22Width</a> <a href="#">ECodedElement::GetFeretBox45Box</a> <a href="#">ECodedElement::GetFeretBox45Center</a> <a href="#">ECodedElement::GetFeretBox45CenterX</a> <a href="#">ECodedElement::GetFeretBox45CenterY</a> <a href="#">ECodedElement::GetFeretBox45Height</a> <a href="#">ECodedElement::GetFeretBox45Width</a> <a href="#">ECodedElement::GetFeretBox68Box</a> <a href="#">ECodedElement::GetFeretBox68Center</a> <a href="#">ECodedElement::GetFeretBox68CenterX</a> <a href="#">ECodedElement::GetFeretBox68CenterY</a> <a href="#">ECodedElement::GetFeretBox68Height</a> <a href="#">ECodedElement::GetFeretBox68Width</a>
Bounding box (center X and Y, height, width)	<a href="#">ECodedElement::GetBoundingBox</a> <a href="#">ECodedElement::GetBoundingBoxCenter</a> <a href="#">ECodedElement::GetBoundingBoxCenterX</a> <a href="#">ECodedElement::GetBoundingBoxCenterY</a> <a href="#">ECodedElement::GetBoundingBoxHeight</a>

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

## 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)	<code>ECodedElement::GetContour</code> <code>ECodedElement::GetContourX</code> <code>ECodedElement::GetContourY</code>
Largest run	<code>ECodedElement::GetLargestRun</code>
Run count	<code>ECodedElement::GetRunCount</code>
Object number (index)	<code>ECodedElement::GetLayerIndex</code> <code>ECodedElement::GetElementIndex</code>
Pixel gray-level value (average, deviation, variance)	<code>ECodedElement::ComputePixelGrayAverage</code> <code>ECodedElement::ComputePixelGrayDeviation</code> <code>ECodedElement::ComputePixelGrayVariance</code>
Pixel gray-level value (min and max)	<code>ECodedElement::ComputePixelMax</code> <code>ECodedElement::ComputePixelMin</code>

## Ellipse of inertia



Eccentricity of the ellipse of inertia	<code>ECodedElement::Eccentricity</code>
Moment	<code>ECodedElement::GetCentralMoment</code> <code>ECodedElement::GetMoment</code> <code>ECodedElement::GetNormalizedCentralMoment</code>

Ellipse (angle, height, width)	<a href="#">ECodedElement::GetEllipseAngle</a> <a href="#">ECodedElement::GetEllipseHeight</a> <a href="#">ECodedElement::GetEllipseWidth</a>
Second order geometric moments (Sigma: X, XX, XY, Y, YY)	<a href="#">ECodedElement::GetSigmaX</a> <a href="#">ECodedElement::GetSigmaXX</a> <a href="#">ECodedElement::GetSigmaXY</a> <a href="#">ECodedElement::GetSigmaY</a> <a href="#">ECodedElement::GetSigmaYY</a>

**NOTE**

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**

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

$$\sigma_x = I_x = \frac{1}{N} \sum (x_i - \bar{x})^2$$

$$\sigma_y = I_y = \frac{1}{N} \sum (y_i - \bar{y})^2$$

$$\sigma_{xx} = I_x = \frac{I_x + I_y}{2} + \sqrt{\left(\frac{I_x + I_y}{2}\right)^2 + I_x I_y + I_{xy}^2}$$

$$\sigma_{yy} = I_y = \frac{1}{N} \sum (x_i - \bar{x})(y_i - \bar{y})$$

$$\sigma_{xy} = I_{xy} = \frac{I_x + I_y}{2} - \sqrt{\left(\frac{I_x + I_y}{2}\right)^2 + I_x I_y + I_{xy}^2}$$

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

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

$$\text{ANGLE} = \arccot\left(\frac{I_x - I_y}{I_{xy}}\right)$$

## Convex Hull

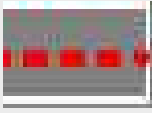


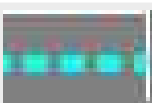

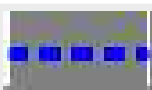


The convex hull of a shape is the convex polygon of minimum area that completely surrounds an object. The convex hull can be used to characterize the object footprint, as well as to observe concavities. Given that the number of vertices of the convex hull is variable, they are stored in a [EPathVector](#)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

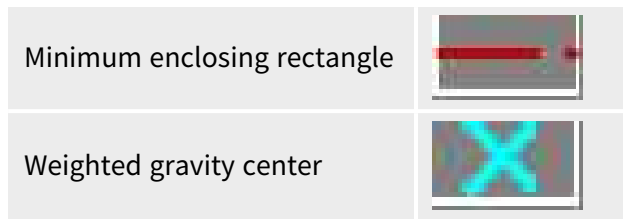
The corresponding function is [ECodedElement::ComputeConvexHull](#).



## 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	



## 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

According 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

**See also:** [using Code Snippets : Creating Code Snippets](#)

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.

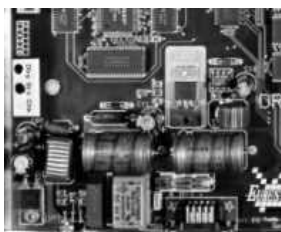
If an object of interest has the same gray level as other regions of the image, you can define "keep" and "ignore" areas using flexible masks and [Encode](#) functions.

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

---

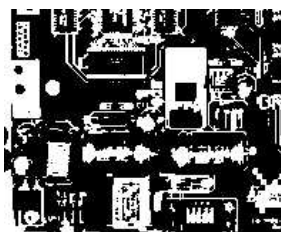


Source image

#### 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`.

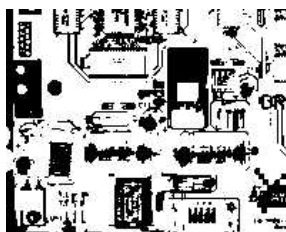


BW8 resulting image that can be used as a flexible mask

## 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.

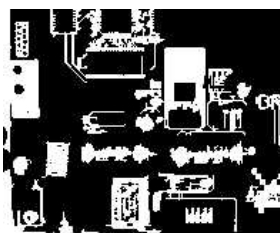


BW8 resulting image that can be used as a flexible mask

## 3. EObjectSelection::RenderMask: from a selection of blobs

---

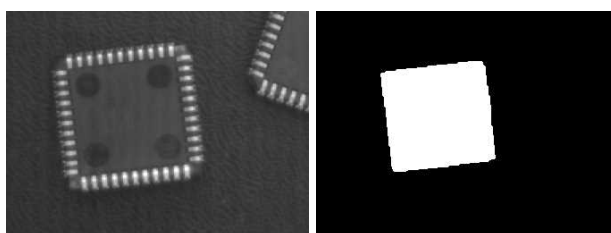
`EObjectSelection::RenderMask` can, for example, discard small objects resulting from noise.



BW8 resulting image that can be used as a flexible mask

## Example: Restrict the areas encoded by EasyObject

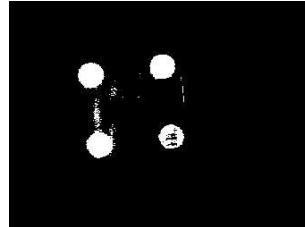
---



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](#):



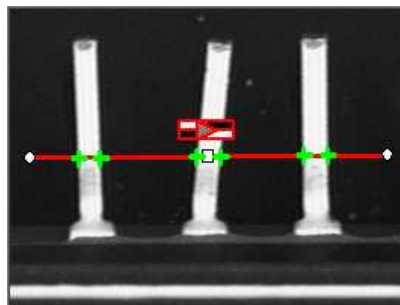
5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

## 3.2. EasyGauge - Measuring down to Sub-Pixel

### Workflow

#### 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.



#### TIP

The theoretical best-case precision is 1/64th pixel for all EasyGauge operators. In practice, you can assume a precision of 1/10th pixel.

#### Workflow

The gauge model can be built programmatically or in a graphical editor, then "played" in the final application.

Choose 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.

- a. Create a gauge object that corresponds to the required measurement.
- b. Change the parameters whose default values are not appropriate.
- c. Invoke the desired measurement function.
- d. 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.

- a. Create a calibrator object.
- b. Place it on the inspected scene.
- c. Adjust calibration parameters.
- d. 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.



### NOTE

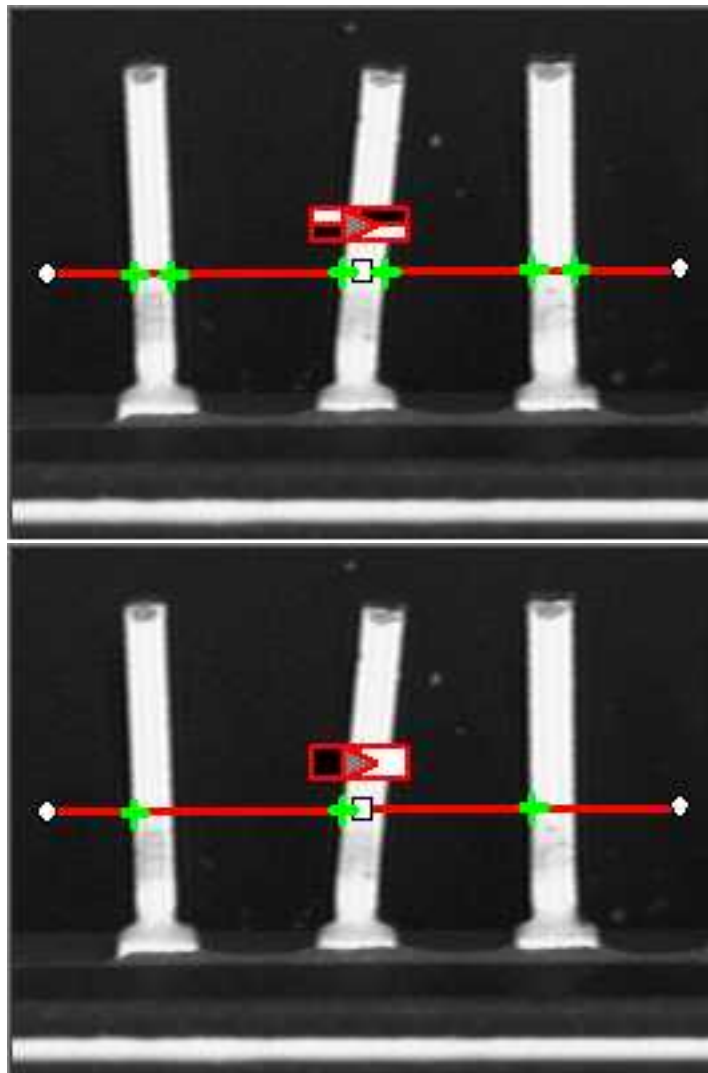
Unlike the rest of Open eVision, EasyGauge uses a pixel center origin (see "[Image Coordinate Systems](#)" on page 20). The subpixel coordinate (0, 0) is the center of the upper left pixel of the image.

# 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

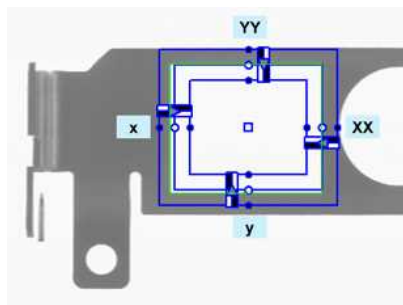
---

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

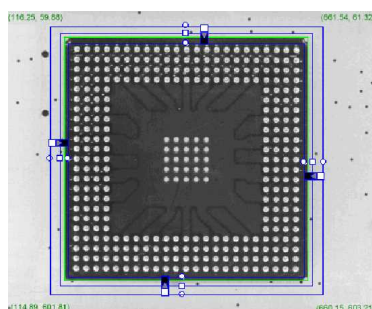
## Usage

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

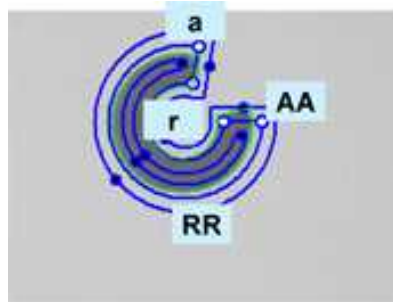
The placement of a [wedge gauge](#) is defined by its nominal position (given by the coordinates of its [center](#)), its nominal [inner](#) and [outer radius](#) ([inner](#) and [outer diameter](#)), its [breadth](#) (difference between radii), the [angular position](#) from where it extends and its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

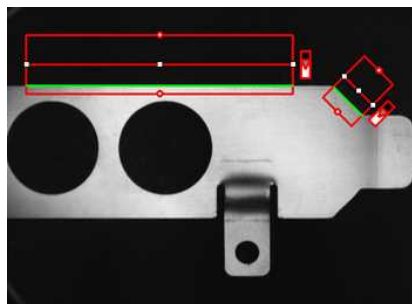
Define and position the gauge, then use [Measure](#) to fit the lines.

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).

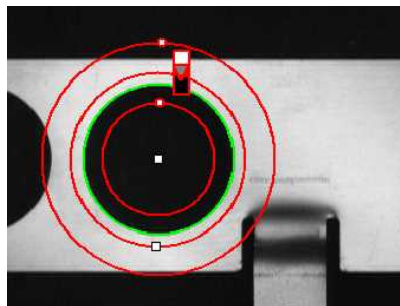
Alternatively, [MeasuredLine](#) provides the results as an [ELine](#) object.

## 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 extends 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

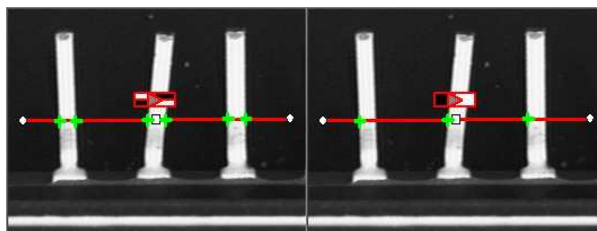
Once the gauge has been defined and positioned, use [Measure](#) to trigger the circle fitting operation. To obtain the measurement results, set the [ActualShape](#) mode to **TRUE**. The [ActualShape](#) mode determines whether an inquiry returns the fitted circle (**TRUE** value) or the nominal circle 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

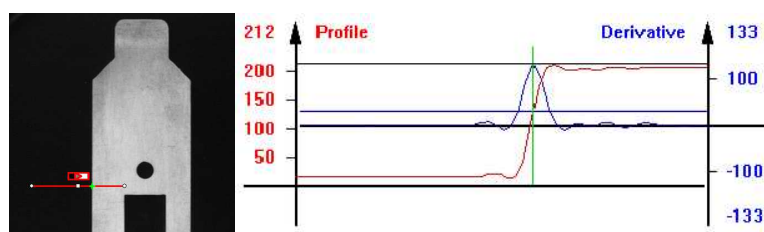
Finds the position of all transition points along a line segment probe that crosses one or several objects edges, and allows selecting the most relevant ones. Crosswise and lengthwise filtering can be activated for noise reduction.





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 [area1](#) 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.
- [BWB2](#) 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.

---

<sup>1</sup>Area between the derivative curve and a horizontal user-defined threshold level

<sup>2</sup>Black / White / Black

## 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,

**GetMeasuredPoint** returns an **EPoint** object which contains located point information.

An integer index between 0 and **GetNumMeasuredPoints-1** must be passed.

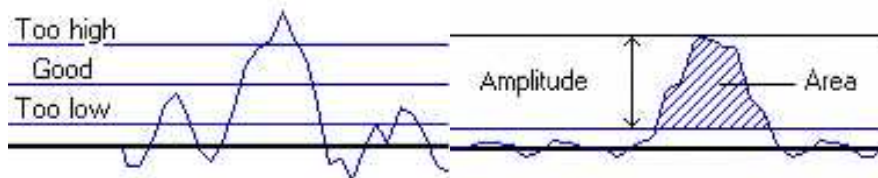
**GetMeasuredPeak:** Returns **EPeak** containing the peak's **Area** and **Amplitude**, and the delimiting coordinates along the probe segment (**Start**, **Length** and **Center** values).

## 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.

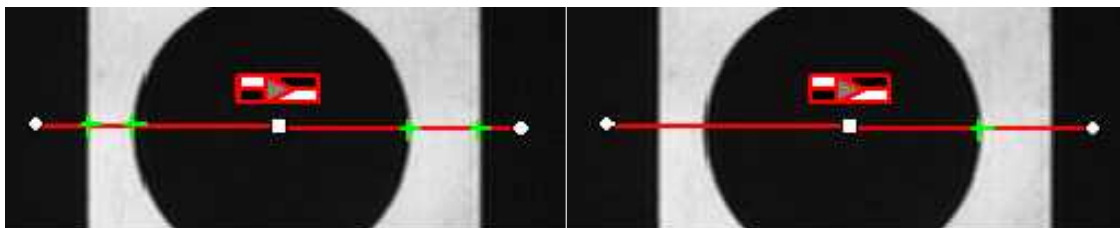
To resolve this dilemma, the EasyGauge peak selection mechanism can reject low contrast or false edges: transition strength is measured by peak *amplitude* and *area*. Every edge measurement determines peak amplitude and area. If either value falls below the **minimum amplitude** or **minimum area**,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



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

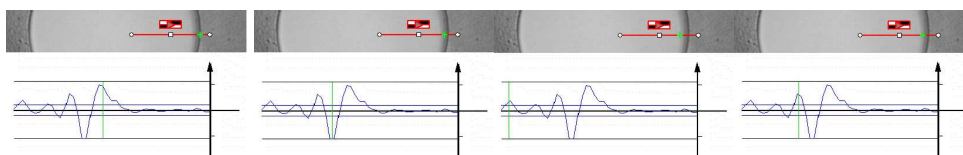
## Multiple versus single transition

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)

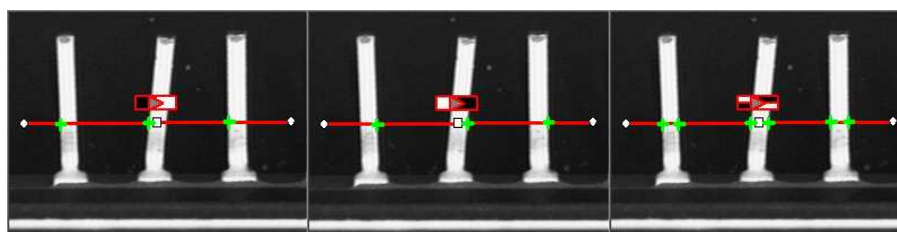
You can select the single most relevant transition based on 4 criteria: the highest peak, the peak with the largest area, the peak closest to the gauge center, or the N-th peak encountered starting from one tip of the gauge.



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

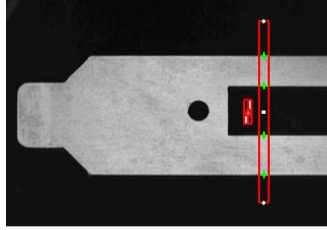
Peak selection can also be refined by choosing the transition polarity: White to Black or Black to White (i.e. positive or negative peak), or indifferent.



Black to white, white to black or indifferent polarities

### Prefiltering

Prefiltering 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

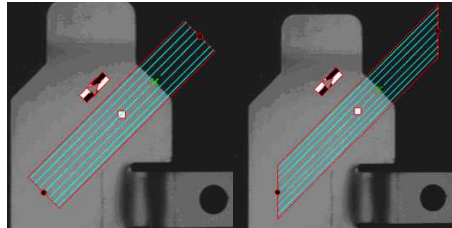
## Transverse Filtering

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  $0^\circ$  or  $90^\circ$ , 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).

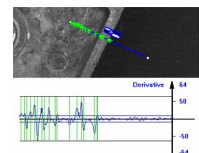
Low values indicate a weak edge, possibly corresponding to an unreliable or inaccurate measurement.

## Tuning Point Measurement Parameters for unclear edges

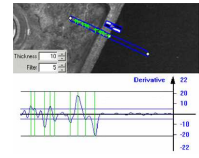
The EasyGauge default parameters and working modes are good for clear edges. More complex situations may need parameter tuning.

### 1. Set the gauge point location and tolerance.

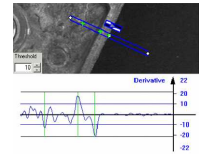
The center position and orientation are easy to decide based on a sample image or on coordinate considerations. The tolerance depends on the edge position variations. A larger tolerance increases the likelihood of hitting an edge, but it may be a false edge or extraneous feature.



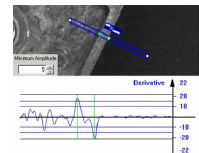
**2. Decide whether noise reduction is required.** Lay the gauge over the desired location and observe the profile curve and its derivative (play with the filtering parameters while looking at the plotted curve). The curve regularity gives an indication of the spread of the gray-level values. When these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



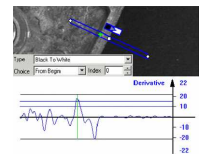
**3. Set the threshold value** to be low enough for useful parts of the peaks to cover enough pixels (to achieve better sub-pixel accuracy), but not lower than the ambient image noise.



**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.



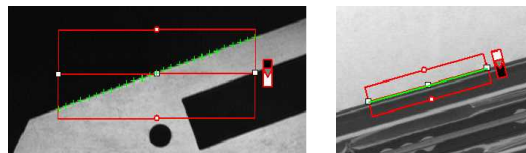
**5. Choose whether all transition points are needed or just the most relevant.** If all are required, they can be queried one after another. Otherwise, a point selection strategy should be chosen based on strength, order or transition polarity (black to white and/or conversely).



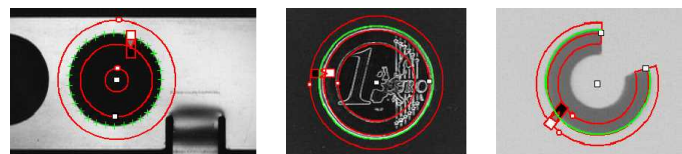
## Find Shapes Using Geometric Models

[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.

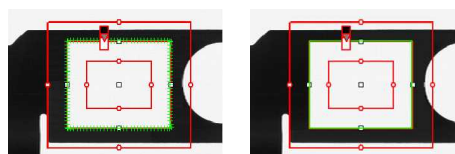
- **Line:** Measures position and orientation of straight edges.



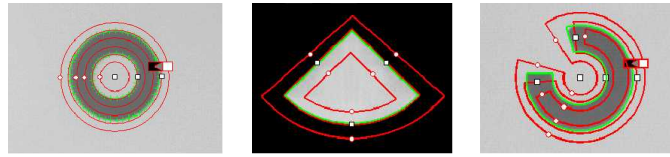
- **Circle:** Measures position and curvature of a circle or arc.



- **Rectangle:** Measures position, orientation and size of a rectangle.



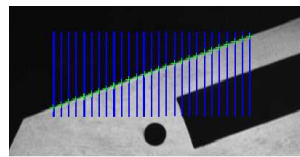
- **Wedge:** Measures position, orientation and size of a ring/ disk sector / curvilinear rectangle.



All gauge types share these common features:

- **Point sampling**

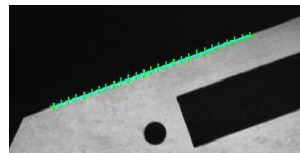
- Point gauges are placed along the edges and point measurement carried out at regularly spaced spots, which can be adjusted differently per side in rectangle and wedge gauges. All point measurement parameters and operating modes are available.
- [SamplingStep](#) sets the spacing of point location gauges along the model.
- [NumSamples](#) returns the number of points sampled during the model fitting operation.



Sampling paths and sampled points

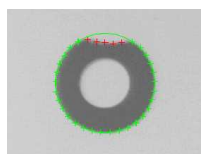
- **Model fitting**

- 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 with 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 the method **Draw** with mode **EDrawingMode\_SampledPath**.

## Group

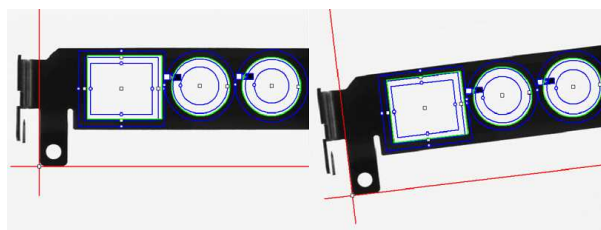
---

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.

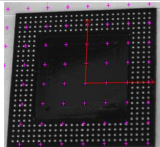
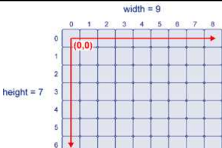
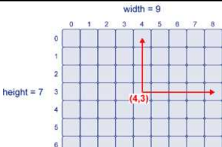
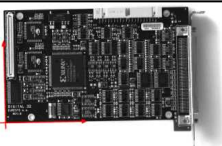
**Detach**, **DetachDaughters** dissociates the gauge or daughters from the mother.





# Calibration and Transformation

## Field-of-view calibration

<p>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.</p>	
<p><b>The Raw sensor</b> coordinate system starts from upper left and extends rightwards and downwards. The range of abscissas is <b>0</b> to <b>width-1</b> and the range of ordinates is <b>0</b> to <b>height-1</b> where integer coordinate values correspond to pixel centers.</p>	
<p><b>The Centered sensor</b> coordinate system starts at the center (<math>[(width-1)/2, (height-1)/2]</math> in the Raw system) and extends rightwards and upwards.</p>	
<p>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.</p>	

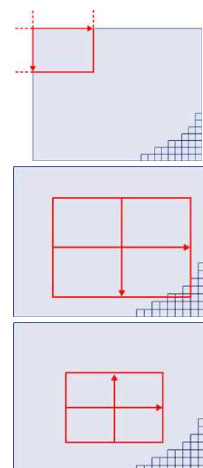
## Before World-to-Sensor Transform

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.

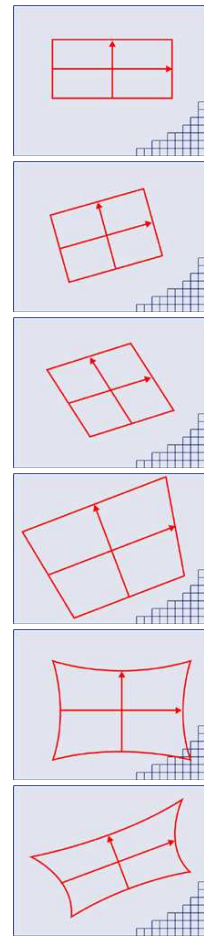
## Effects of World-to-Sensor Transform

- **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:

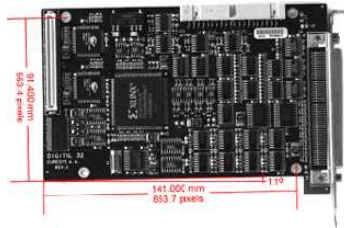
## Estimate (feasible if no distortion correction is required and accuracy requirements are low)

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  $[3/4, 4/3]$ ).

Use the method `EWorldShape::AddLandmark` to add reference points, then use `EWorldShape::AutoCalibrateLandmarks` to calculate the calibration.

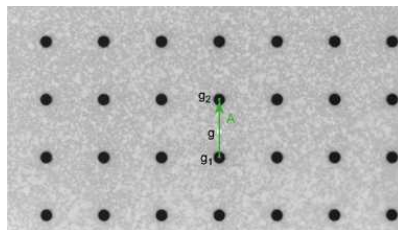
## Analyze a Calibration target

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.

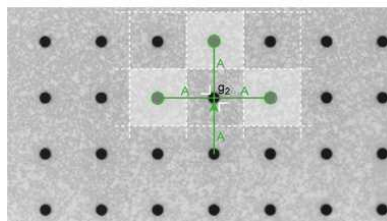
Open eVision relies on the use of a specific target holding a rectangular grid of symmetrical dots (of any shape) with no other object on the grid.

### 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 restricts 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  $A$ .



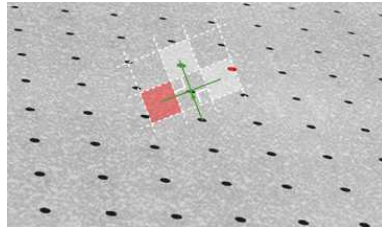
- b. Starting from the extremity of the reference segment ( $g_2$ ), the algorithm determines 3 tolerance areas (white squares in the figure), in perpendicular directions. The tolerance areas are centered at a distance  $A$  (length of the reference segment) from ( $g_2$ ). They are square, with a side-length 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](#).

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).



#### TIP

Use the method `EWorldShape::AutoCalibrateDotGrid` to automatically perform the process above.

## 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:

$$\begin{aligned} f\text{-}o\text{-}v \text{ width} &= \text{pixel width} * \text{sensor width} \\ f\text{-}o\text{-}v \text{ height} &= \text{pixel height} * \text{sensor height} \end{aligned}$$

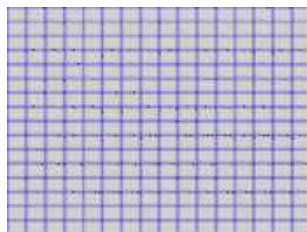
or

$$\begin{aligned} \text{sensor width} &= f\text{-}o\text{-}v \text{ width} * \text{horizontal resolution} \\ \text{sensor height} &= f\text{-}o\text{-}v \text{ height} * \text{vertical resolution} \end{aligned}$$

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

## Ratio

---



Anisotropic aspect ratio

## Center abscissa and ordinate

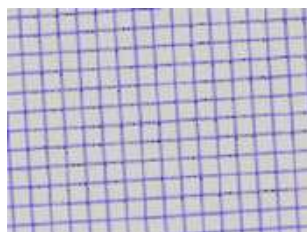
---

The **center abscissa** (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



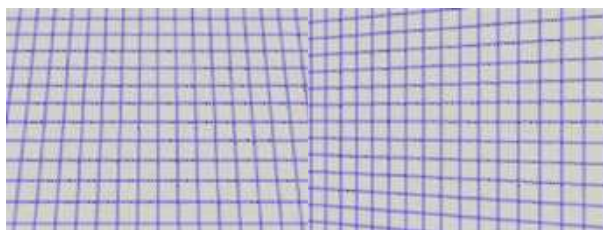
### NOTE

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.

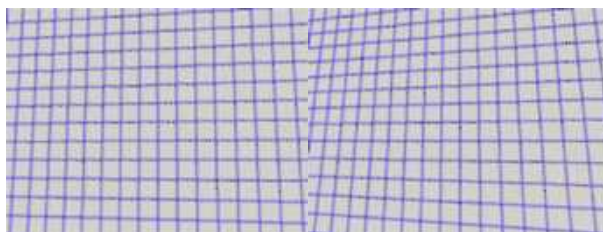


Tilt X and tilt Y angles

## Perspective strength

---

The [perspective strength](#) gives a relative measure of the perspective effect. The shorter the focal length, the larger the value.

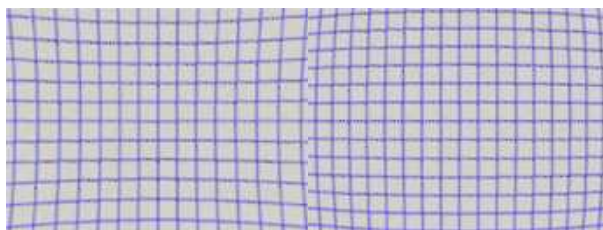


Weak and strong perspective

## Distortion strength

---

[Distortion strength](#) gives a relative measure of radial distortion in the image corners, that is the ratio of image diagonal length with and without distortion.

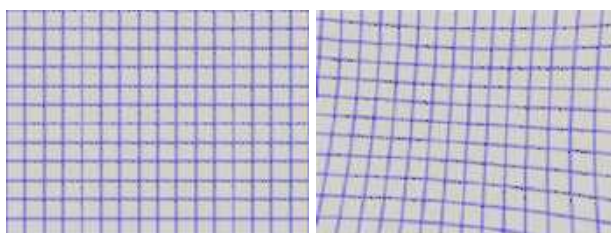


Positive and negative distortion

[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

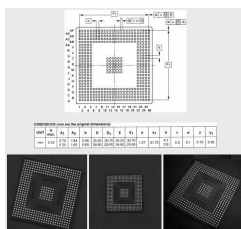
An **EWorldShape** object manages a field-of-view calibration context. Such an object is able to represent the relationship between world coordinates (physical units) and sensor coordinates (pixels), and account for the distortions inherent in the image formation process.

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.

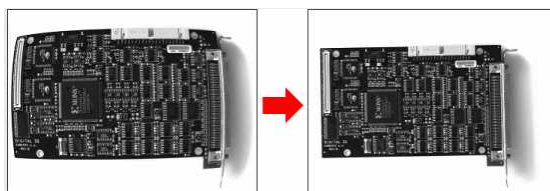
The goal of calibration is twofold:

- 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.



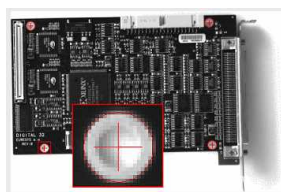
### Single model versus multiple viewing conditions

- 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.



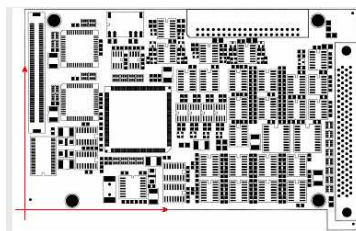
### Feature point in sensor space

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.

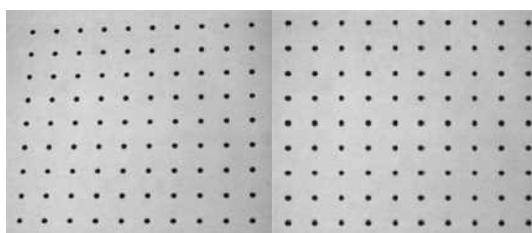




Reference frame in world space

## Unwarp

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

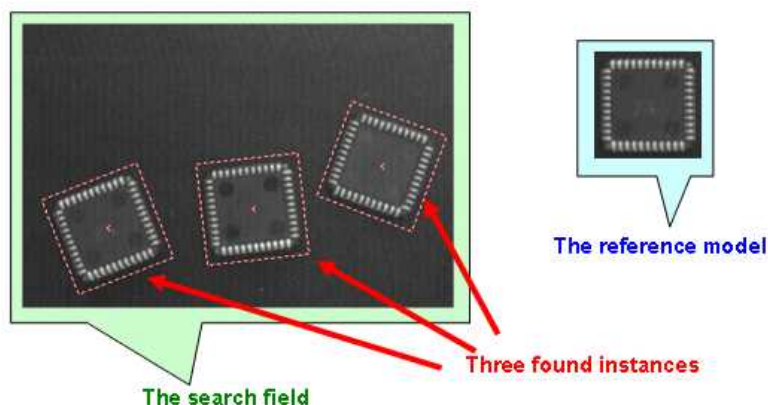


Distorted vs. Unwarped image

## 3.3. EasyFind - Matching Geometric Patterns

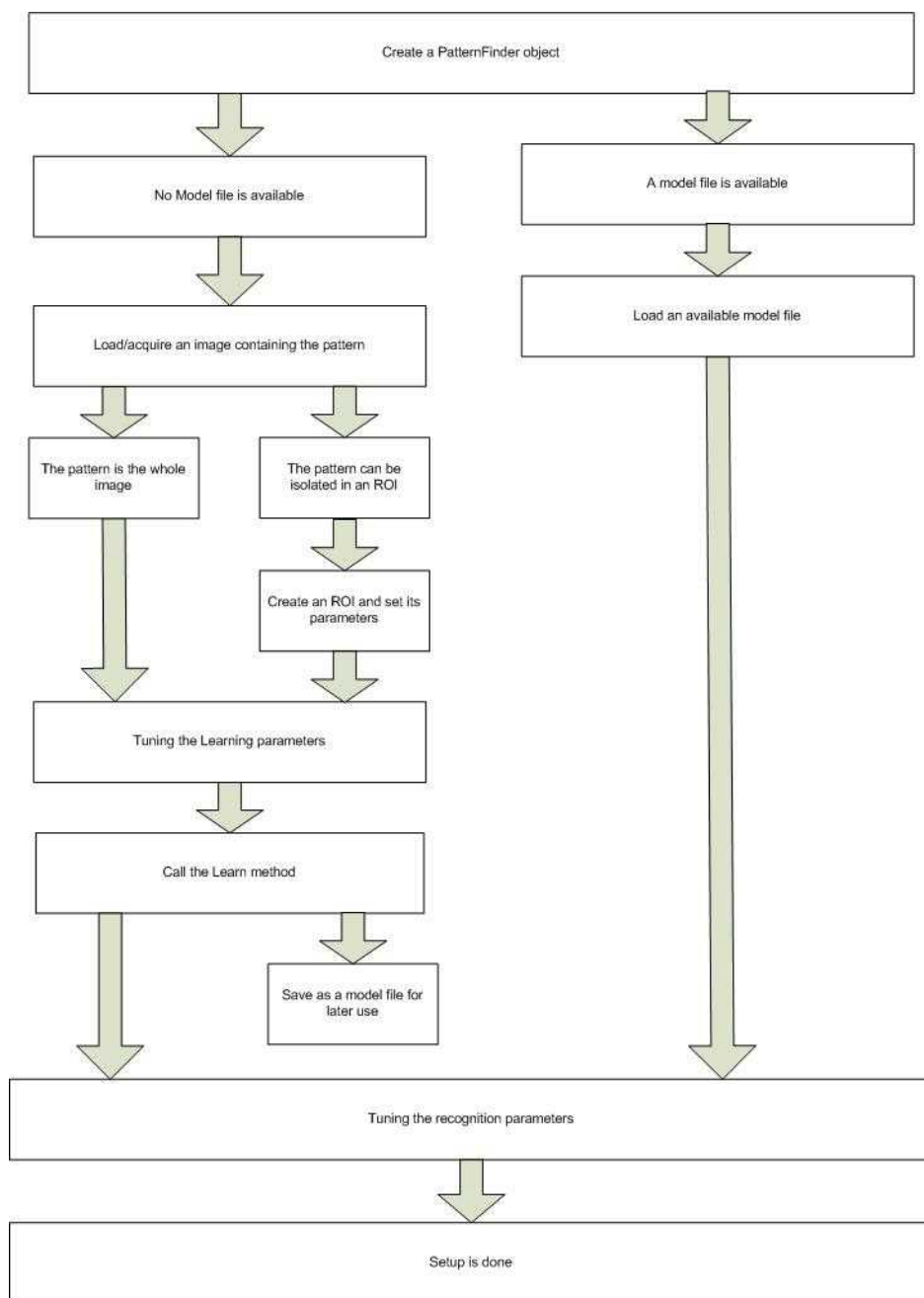
### Workflow

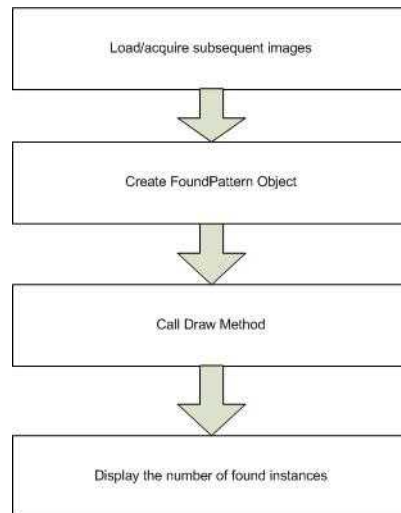
`EasyFind` learns a reference model from a pattern, which is used to find similar patterns in other images and retrieve information about these instances. It is quick and robust, and very tolerant of noise, blur, occlusion, missing parts and changes in illumination.





## Workflow

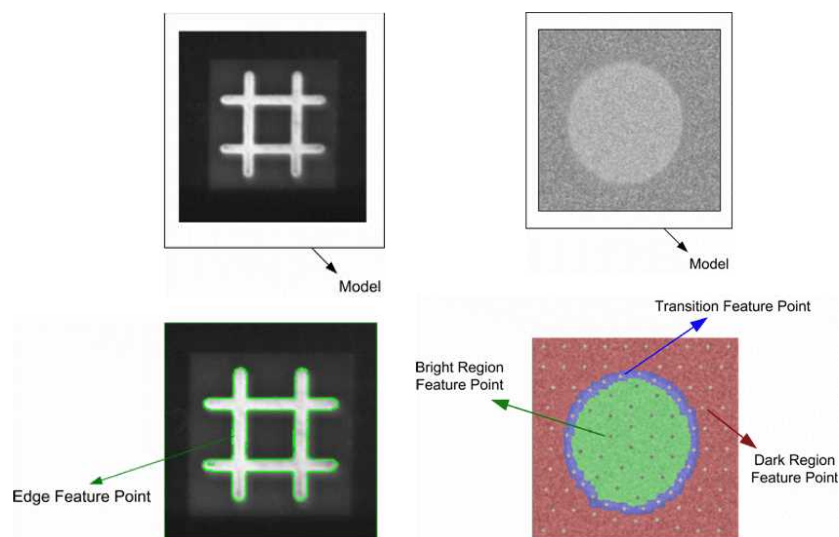




## Feature points definition

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.

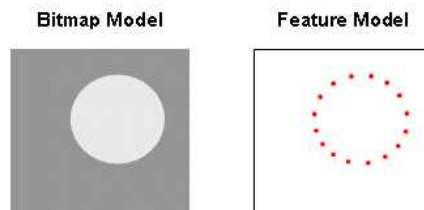


- **Edge feature points:** an abrupt change of gray level between two regions indicates an edge at this location in the search fields.
- **Transition feature points:** a smooth change of gray level between two regions indicates a transition area in their neighborhood (represented by dots in the blue 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 or Thin structures). 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.



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

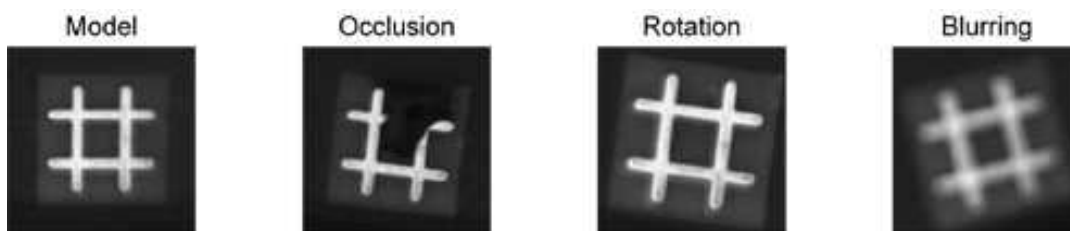
### Consistent Edges

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:

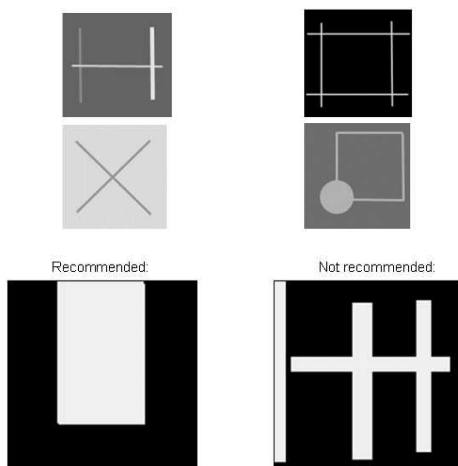
- `PointByPointNormal`: if points share the same contrast polarity.
- `PointbyPointInverse`: if points exhibit opposite contrast polarity.
- `PointByPointAny`: regardless of their respective contrast polarity.



### 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.

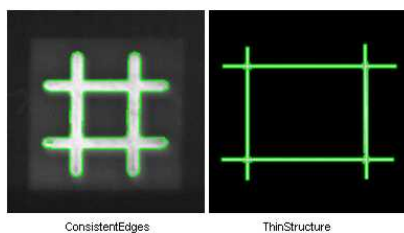
**Good for** models containing thin elements.



### 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.

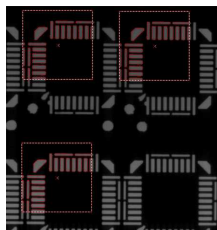


## Finding Process

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

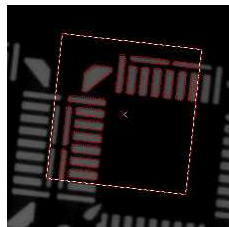
Set the maximum number of instances that EasyFind should return. In this example the number was three.



## Angles and scales (thin structure and consistent edges pattern types)

---

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

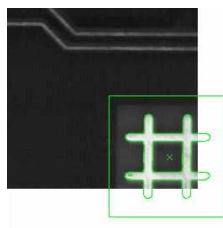


## Advanced Features

### 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 the search field is set to  $> 0$  pixels.

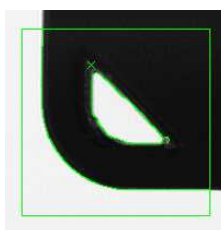


### Tune Parameters

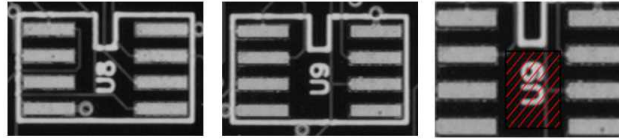
---

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.
- 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).



- **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.



- **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

## 3.4. EasyMatch - Matching Area Patterns

### Workflow

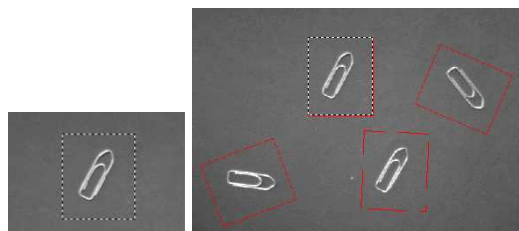
#### EasyMatch

---

##### Reference

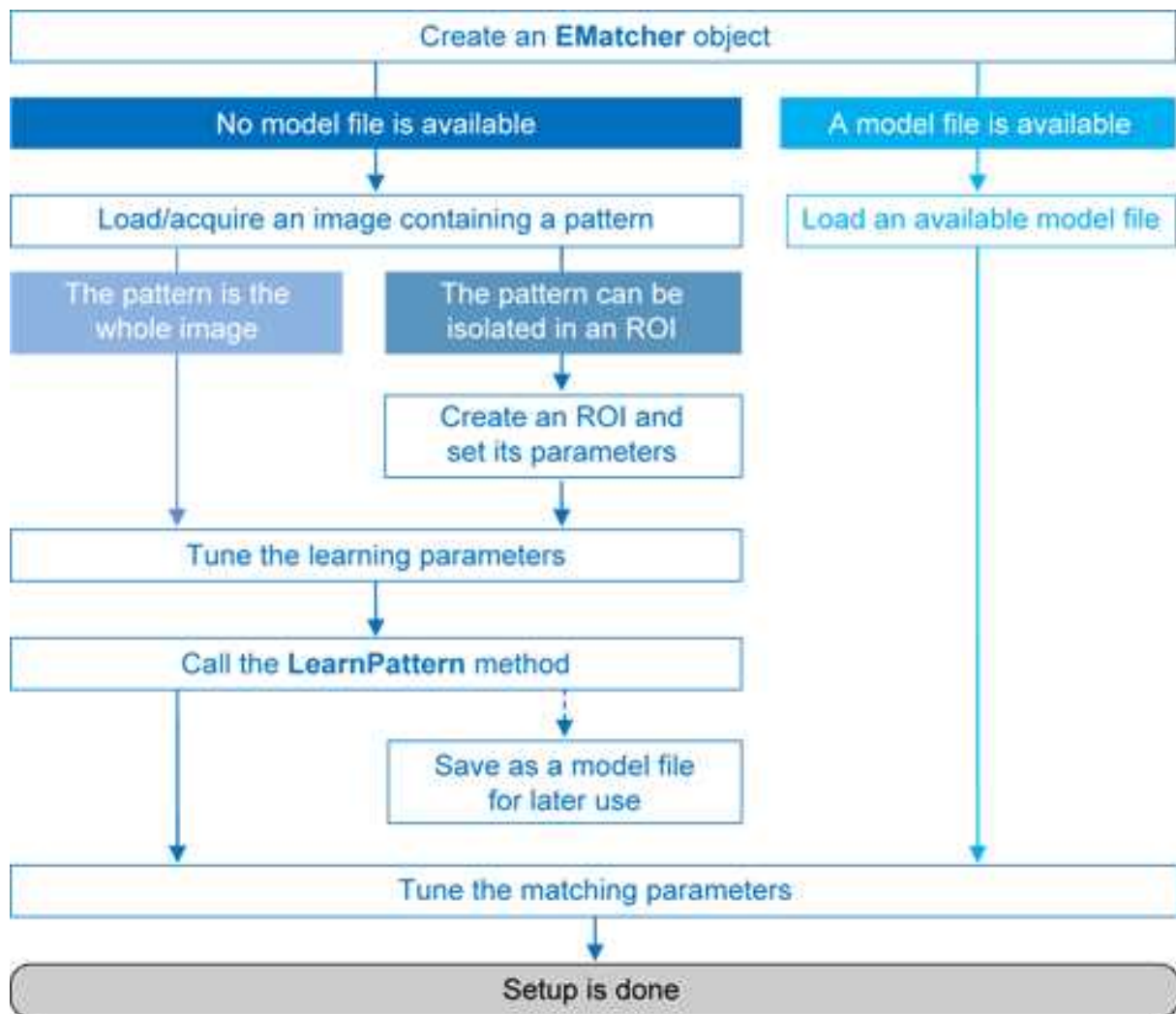
EasyMatch learns a pattern and finds exact matches:

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.

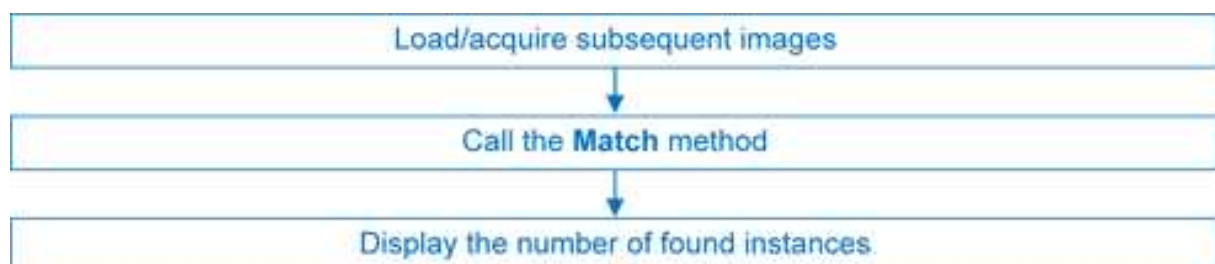


Learning and Matching a pattern

## Learning workflow



## Matching workflow



## Learning Process

Select an image containing the pattern/ROI to be searched for and call **LearnPattern**. Pass an arbitrary shaped region of interest (**ERegion**) to ignore the pixels outside of this region.

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.

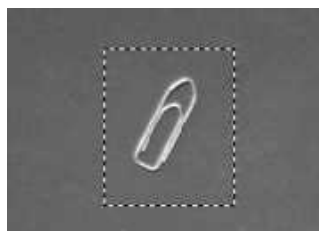
### Customize Parameters

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).  
  
Alternatively, pass an arbitrary shaped region of interest (**ERegion**) to ignore the pixels outside of this region. It is equivalent to setting all pixels outside of the region to black and having a **DontCareThreshold** set to 1.
- **MinReducedArea:** To improve time performance, EasyMatch sub-samples the pattern. This parameter stipulates the minimum size of the pattern (as its area in pixels) during sub-sampling. The smaller the value, the faster the matching process, but, set too low, it decreases the matching process reliability.  
The value of **MinReducedArea** is computed automatically if **AdvancedLearning** is enabled (default behavior). Setting explicitly **MinReducedArea** will disable **AdvancedLearning**. A value of 64 is usually a good compromise.
- **AdvancedLearning:** If the pattern is defined as a ROI of an image, **AdvancedLearning** optimizes learning parameters, such as **MinReducedArea**, by using the whole image context.  
**AdvancedLearning** is enabled by default, as it leads to better results in case of tiled or periodic images. If **MinReducedArea** is set explicitly, **AdvancedLearning** is disabled. Please note that as **AdvancedLearning** changes the number of pixels in the pattern, it can have a significant impact on the matching process duration.



- **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 pass an arbitrary shaped region of interest (**ERegion**) to ignore the pixels located outside of this region.

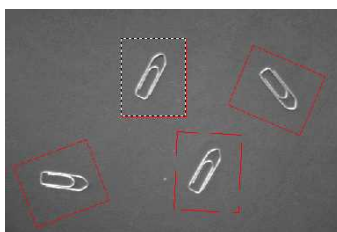
You can set these parameters:

- 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  $2K$ , 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  $[0 \dots \text{NumReductions}-1]$ .
  - 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.

## 3.5. EChecker2 - Validating Golden Templates

### EChecker2

- EChecker2 is a tool that allows to inspect an image using a Golden Template validation. It works in 2 steps:
  - a. The *Model Creation* involves pre-processing a set of reference images to compute a model. You can create the model once and archive the results in a Golden Template model for later use.
  - b. The *Inspection* involves processing an image and checking its quality using the previously computed model.

These 2 operations are totally independent and can even be programmed in separate applications.

- EChecker2 is part of the EasyObject library.
- The following sections present the relevant API functions for use in the training and inspection steps.

## EChecker2 vs. EChecker

---

- EChecker2 supersedes the original EChecker:
  - It expands EChecker with an up-to-date API.
  - It adds the possibility to use geometrical pattern matching and a flexible number of fiducials.
  - It works with the newer `ECodedImage2`.
  - It only requires the EasyObject license.
- For all these reasons, the original EChecker is now considered legacy and deprecated.

## Creating a Model

During the model creation phase, the good images are processed to build the model that is used in the inspection phase. The model includes the pixel acceptance ranges, in the form of 2 threshold images, as well as the information needed to realign and normalize the images.

To create a model, 2 operations are performed: initialization and training.

### Initialization

---

- The initialization of the model creation process is done on a specific image, called the reference.
- On this reference, the following information are defined or computed:
  - The global gray level metrics are computed as reference for the normalization process. It is thus very important that the reference image is well lit and contrasted.
  - One or more `ERegions` are placed to define the location patterns (fiducial marks or landmarks).  
The location of these patterns is used as reference in the realignment process.

**TIP**

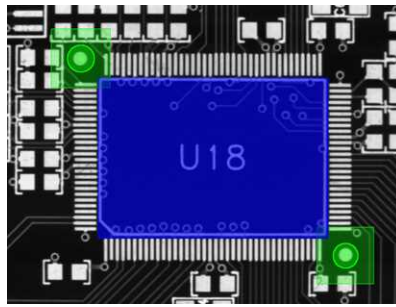
- If you use only 1 pattern, the only transformation handled is the translation.
- With 2 or more patterns, the scaling and the rotation are also processed.

**NOTE**

- As per Open eVision 2.15, you can only use either 1 or 2 regions.
- The possibility to use more regions will be added in the future.

- An `ERegion` is defined to delimit the area to be inspected. This area should only include pixels of the rigid part (that moves with the fiducial marks), and not the background.

- To perform the initialization, use the method `EChecker2::Initialize`.



Initialization: the fiducial regions and tolerances (green) and the inspection region (blue)

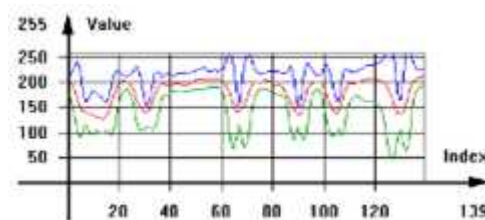
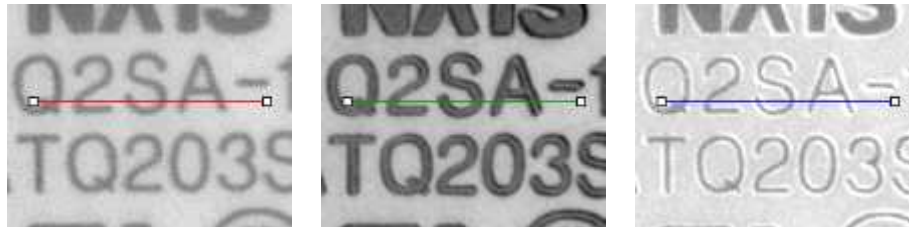
## Training

---

- After the initialization, the main training phase begins.
  - All the training images are processed and are averaged using statistical training (see below).
  - The training uses realignment to deal with displacement of the inspected part in the field of view.
  - The training uses gray-level normalization to deal with global illumination changes.
- Ideally, use 16 images or more in training to create the low and high threshold images that serve as the basis of the inspection process.
- To perform the training:
  - Use the method `EChecker2::Train` with class instances.
  - Use the method `EChecker2::TrainFromImageFiles` with a list of image files.

## Statistical training

- Use several training images to optimize the assessment of normal gray-level variations and acceptance intervals:
  - Consecutive images of the same part without any change (static test) generates a gray-level distribution that corresponds to the noise distribution.
  - Consecutive images of different defect-free parts reveal variations due to the parts themselves (as opposed to defects).



Accepted gray-level ranges

## Model creation parameters

- Choose the **TrainingMode** to fit your needs:
  - **Quick** for a quick training process and simpler cases (well defined defects and stable illumination).
  - **Precise** for more difficult cases.
  - **Default: Precise mode.**
- Set **NormalizationMode** to select the type of normalization used by EChecker2:
  - **Moments: linear.**
  - **Threshold: non-linear.**
  - **NoNormalization:** if your acquisition process already produces consistently lit images.
  - **Default: Moments.**
- Choose the **FiducialMatchingMode** to define the search of the fiducials inside the processed images:
  - **Geometric** for well-defined fiducials that can potentially suffer from occlusion.
  - **Area** for less-well defined fiducials.
  - **Default: Geometric.**
- Set **FiducialHorizontalTolerance** and **FiducialVerticalTolerance** to adjust the search distance for the fiducials from the reference position (after realignment).
  - **Default: 30 pixels.**

- Set `InspectionTolerance` to adjust the acceptance ranges during the inspection.
  - Use higher values to make the inspection process more tolerant to noise and/or texture.

**NOTE**

All these parameters have an influence on how the model is built, and, as such, if any of these parameters is changed, you must restart the model creation.

## Model serialization

- After the training, use the methods `EChecker2::Save` and `EChecker2::Load` you can save the created model in a single file including all the relevant information and to retrieve it.

## Inspecting an Image

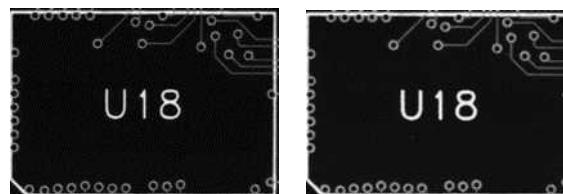
- Use the inspection on an image to assess it towards the trained model.

The process is straightforward:

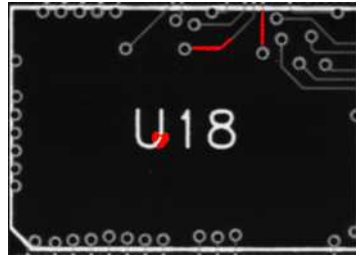
- a. The sample image is realigned with the model.
- b. The gray-level is normalized.
- c. This gray-level is combined with the high and low threshold images to populate an `ECodedImage2`.
- d. The computed blobs are made of pixels that fall out of the range defined by the threshold images and thus represent potential defects.



The realigned Image



The low and high threshold images



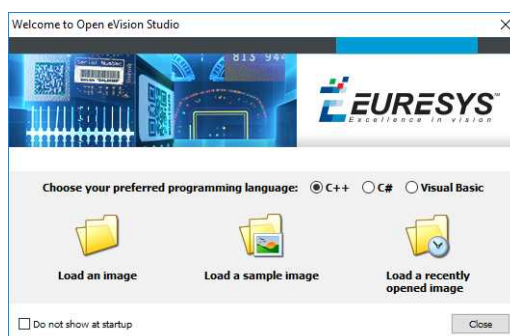
### Detected defects after inspection

- When the inspection is done, you can discard the smaller defects (usually noise), as well as measure the geometric characteristics (location, size, orientation...) using the standard EasyObject2 processes.
- To perform the inspection, use the method [EChecker2::Inspect](#).

## 4. Using Open eVision Studio

### 4.1. Selecting your Programming Language

When you start Open eVision Studio for the first time, the following welcome screen is displayed:



1. Select your programming language.

**TIP**

Your selection is saved and your programming language will be automatically selected next time you start Open eVision Studio.

**NOTE**

When you change your programming language, any script present in the scripting window is automatically deleted and the window content is reset.

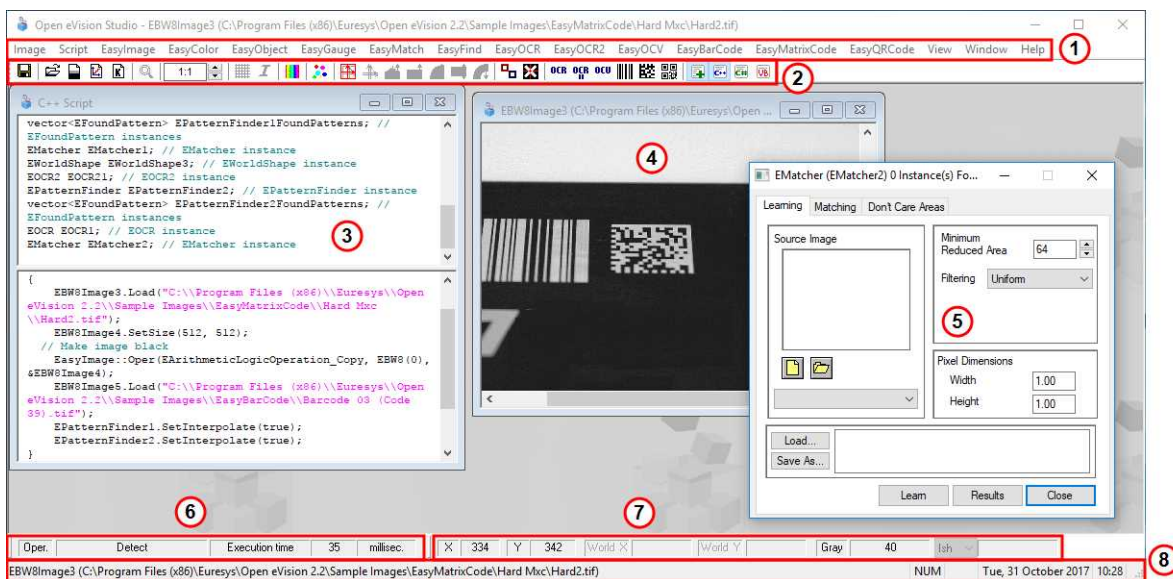
2. Click on one of the Load buttons to already load one or several images for later processing.
3. Check the Do not show at startup box to hide this welcome screen next time you start Open eVision Studio.

**TIP**

To access this welcome screen at any time, and change this setting, go to the Help > Welcome Screen menu.



## 4.2. Navigating the Interface



Open eVision Studio graphical user interface (GUI) is organized as follows:

1. The *main menu bar* gives you access to the functions and tools of all libraries.



### TIP

Open eVision Studio does not require any license and allows you to test all libraries. Of course, if you copy code from Open eVision Studio in your own application but you do not have the required license, you will receive a "missing license" error at run-time.

2. The *main toolbar* gives you quick access to main Open eVision objects such as images, shapes, gauges, bar codes, matrix codes...
3. The *script window* displays the code, in the programming language you selected, corresponding to the actions you perform in Open eVision Studio. You can save or copy this code in your own application at any time.
4. The *image windows* display the open images that you can process using the libraries and tools.
5. The *tool windows* enable you to easily configure all the available tools. The corresponding settings are automatically added in the script window for easy reuse.



### TIP

Most tool windows are floating and you can easily move them outside the Open eVision Studio main window to make better use of your screen size.

6. The *execution time bar* displays the precise time taken for the execution of the selected functions (measured in milliseconds or microseconds) on your computer. This accurate measurement helps you to evaluate the performance of your application.

7. The *color toolbar* displays current information such as the X and Y coordinates of the cursor on an image and the corresponding pixel value.
8. The *status bar* displays general information about the application such as the active image file path...

## 4.3. Running Tools on Images

### Step 1: Selecting a Tool

When you use Open eVision Studio, the first step is to select the library and the tool you want to use on your image.

To do so:

1. In the main menu bar, click on the library you want to use.
2. Click on the tool you want to use.

**TIP**

All libraries (except EasyImage, EasyColor and EasyGauge) expose only one tool named **New Xxx Tool**. Some of these libraries also expose additional functions.

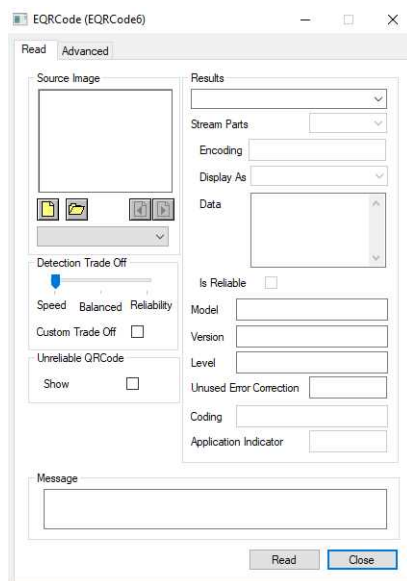
3. In the dialog box, enter a **Variable name** for the variable that is automatically created and that will contain the result of the processing.



Example of variable creation dialog box for EasyQRCode

4. Click OK.

The selected tool dialog box opens.



Example of variable creation dialog box for EasyQRCode


The next step is ["Step 2: Opening an Image" on page 107](#).

## Step 2: Opening an Image

Once you have selected your library and your tool, you need to open an image to apply this tool.

In the **Source Image** area of the selected tool dialog box:

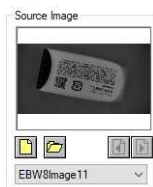
### 1. Open an image:



- Click on the  **Open an Image** button and select one or several (using SHIFT and CTRL) images on your computer.
- Or select one of the images (or one of the ROIs, if any) already open in the drop-down list.



### NOTE

You can select only images with an appropriate file format (JPG, PNG, TIFF or BMP) and in 8- and/or 24-bit depending on the library.



2. If you selected several images, activate one with the  **Load Previous** or  **Load Next** buttons.

The tool is automatically applied on any loaded image and, at this stage, the result is displayed based on the tool default settings.

The next step is "[Step 3: Managing ROIs](#)" on page 108.

## Step 3: Managing ROIs

In some cases, most often to decrease the processing time or to single-out the object you want to read, you do not want to process the whole image but only one or several well defined rectangular parts of this image, or ROIs (Regions Of Interest).



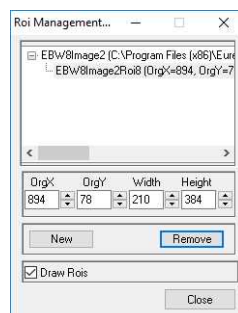
### TIP

In Open eVision, ROIs are attached to an image and exist only as long as the parent image is available.

## Creating a ROI

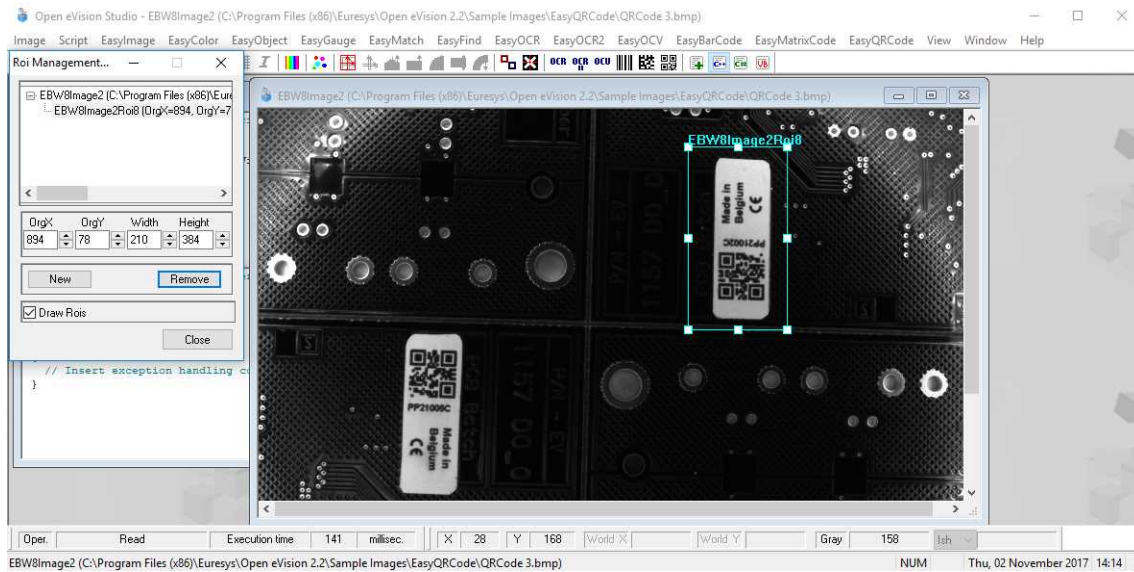
1. Open the image:
  - If the image is already open, activate the corresponding image window.
  - If the image is not open yet, go to the main menu: **Image > Open...** to open one.
2. To create an ROI, go to the main menu: **Image > ROI Management...**

The **ROI Management** window is displayed as illustrated below.



3. Select the image in the tree.
4. Click on the **New** button.
5. In the dialog box, enter a **Variable** name for the new ROI.

The ROI is represented as a color rectangle on your image as illustrated below.



6. Drag the ROI corner and side handles to move it to the required position.

7. Click on the Close button to close the ROI Management window.

The next step is "[Step 4: Configuring the Tool](#)" on page 110.

## Managing ROIs

You can add, change and remove ROIs.



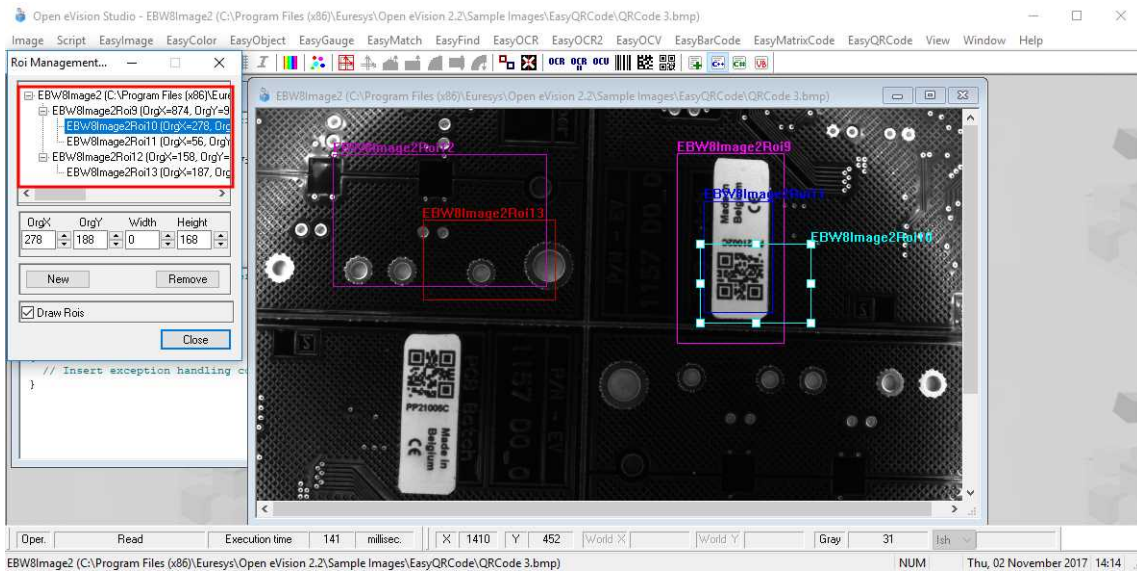
### TIP

An image can have several ROIs. Each ROI can be attached directly to the image (meaning that its position is relative to the image) or to another ROI (meaning that its position is relative to this 'parent' ROI).

1. To manage ROIs, go to the main menu: **Image > ROI Management**...

The ROI Management window is displayed with the ROI relation tree as illustrated below.

If the **Draw Rois** box is checked, all ROIs are displayed on the image with a different color.



2. Select an ROI in the ROI relation tree.
3. Drag the ROI corner and side handles to change the position and size of the selected ROI (as well as the position of all ROIs attached to it if any).
4. Click on the **New** button to add a new ROI attached to the selected ROI.

**TIP**

Select the image at the top of the ROI relation tree to attach the ROI directly to the image.

5. Click on the **Remove** button to delete the selected ROI (and all ROIs attached to it if any).
6. Click on the **Close** button to close the **ROI Management** window.

## Step 4: Configuring the Tool

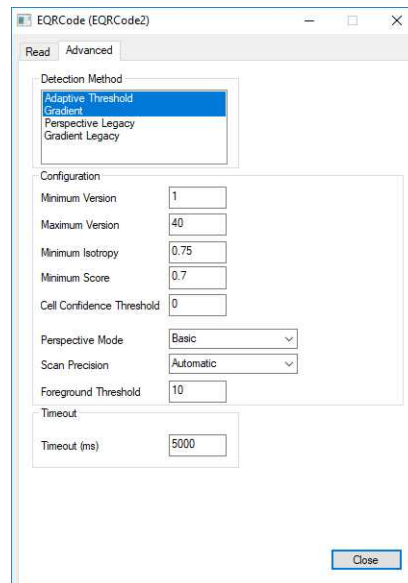
Once your image, including its ROIs if you created some, is ready, you need to configure your tool.

In the tool window:

1. Open the various tabs.

**TIP**

When you create a new tool, all parameters are set with their default value.



Example of the parameter tab of an EasyQRCode tool

2. In each tab, set the value of the parameters as desired.

Please refer to the "Functional Guide" and to the "Reference Manual" for detailed information about the parameters, their function and their default value.

For specific actions such as learning or using gauges, please refer to the "Functional Guide".

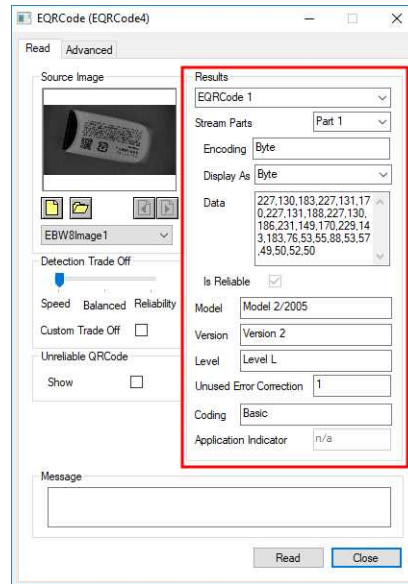
3. Run the tool and analyze the results as described in the next step "[Step 5: Running the Tool and Checking Execution Time](#)" on page 111.

## Step 5: Running the Tool and Checking Execution Time

Once your tool parameters are set, run your tool and, if desired, check the execution time on your computer.

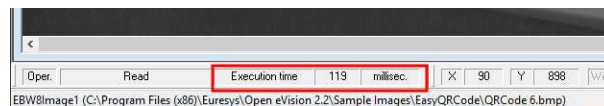
In the tool window:

1. Click on the **Read, Detect, Results** or **Execute** button (depending on the library function), to run the tool on the selected image.
2. Check the results on the image and in the Results field or area as illustrated below.



Example of results after reading a QRCode

3. If you do not have the expected results:
  - Try to change your parameters (start with default values then change one parameter at a time).
  - If your image is not good enough, try to enhance it as described in .
4. Check the execution time in the execution time bar at the bottom left of the main Open eVision Studio window.



The execution time



#### TIP

The execution time is the actual time that the processing took as measured on your computer. It depends your computer processor, memory, operating system... and, of course, on the processor load at the time of execution. Thus this execution time slightly varies from execution to execution.

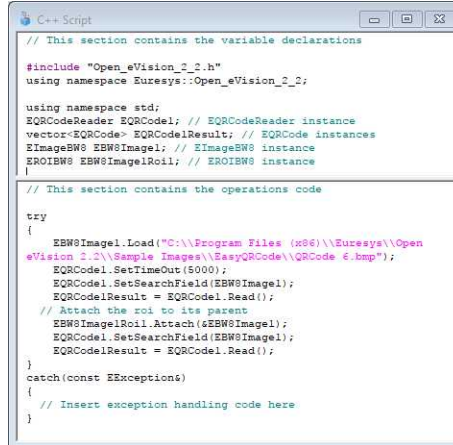
5. To get a more representative execution time, click on the **Read**, **Detect**, **Results** or **Execute** button several times and calculate the mean execution time.
6. If your application requires that you reduce the execution time, try:
  - To change the tool parameters,
  - To add one or several ROIs on your image,
  - To enhance your image.

The next step is "[Step 6: Using the Generated Code](#)" on page 113.



## Step 6: Using the Generated Code

By default, Open eVision Studio translates all the operations you perform in the interface into code in the language you selected as illustrated below.



```

C++ Script
// This section contains the variable declarations
#include "Open_eVision_2_2.h"
using namespace Euresys::Open_eVision_2_2;

using namespace std;
EQRCoderReader EQRCoder; // EQRCoderReader instance
vector<EQRCoder> EQRCoderResult; // EQRCoder instances
EImageBWS EBWSImage1; // EImageBWS instance
EROIENB EBWSImage1Roll; // EROIENB instance
|

// This section contains the operations code

try
{
    EBWSImage1.Load("C:\\Program Files (x86)\\Euresys\\Open
eVision 2.2\\Sample Images\\EasyQRCode\\QRCode 8.bmp");
    EQRCoder.SetTimeOut(5000);
    EQRCoder.SetSearchField(EBWSImage1);
    EQRCoderResult = EQRCoder.Read();
    // Attach the Roll to its parent
    EBWSImage1Roll.Attach(EBWSImage1);
    EQRCoder.SetSearchField(EBWSImage1);
    EQRCoderResult = EQRCoder.Read();
}
catch(const EException&)
{
    // Insert exception handling code here
}

```

Once your tool results suit you, you can save or copy this generated code to use it in your own application.

### Copy and paste the code in your application

In the script window:

1. Select the code section you want to copy.
2. Right click on this code and click **Copy** in the menu.
3. Go to your development environment tool and paste the code in place.

### Save the code

1. Go to the **Script** menu.
2. Click on **Save Script As...**
3. Enter a file name and path to save the code as a text file.

### Manage the generated code

In the **Script** menu, you can:

- Select the programming language (please note that if you change the language, the script window content is automatically deleted).
- Activate or deactivate the **Script Code Generation**. Deactivate this option if you want to perform some operations without saving them as code.

## 4.4. Pre-Processing and Saving Images

### When should you pre-process your images?

Of course, the best situation is to set up your image acquisition system to have good and easy to process images so the Open eVision tools run smoothly and efficiently.

If this is not possible or easy to achieve, you can pre-process your images or your ROIs to enhance and prepare them for the Open eVision tool you want to run.

Using the various available functions, you can adjust the gain and offset of your image, apply a convolution, threshold, scale, rotate and white balance your image, enhance contours... using EasyImage and EasyColor functions.

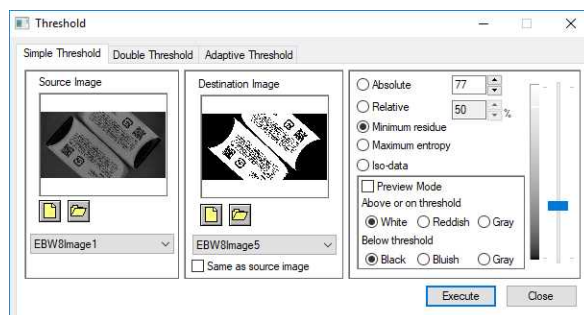
### Pre-processing images

The difference between pre-processing an image and running tools is that the pre-processing generates a new image while the tools mainly extract and retrieve information from the image without changing it.

To pre-process an image or an ROI:

1. In the main menu bar, click on the library you want to use (EasyImage or EasyColor).
2. Click on the function you want to use.

Most function dialog boxes are similar to the one illustrated below with 2 image selection areas and a parameter setting area.



Example of a pre-processing dialog box (Threshold with EasyImage)

3. If there are multiple versions for your selected function, open the corresponding tab.
4. In the Source Image area, open the source image (as described in "Step 2: Opening an Image" on page 107).
5. In the Destination Image area, open or create a new destination image.
6. Set your parameters.
7. Click on the Execute button.

The pre-processed image is available in the destination image as illustrated below.



Source and destinations images (Threshold with EasyImage)

8. If you want to use the destination image outside of Open eVision Studio, save it as described below.

## Saving an image

1. Click on the image you want to save to makes its window active.
2. To open the save menu either:
  - Right-click in the image
  - Or open the main menu > **Image**
3. Click on **Save as...**
4. Select the file format (JPEG, JPEG2000, PNG, TIFF or Bitmap).
5. Enter a name and select a path.
6. Click on the **Save** button.

## 5. Tutorials

### 5.1. EasyObject

#### Removing Non-Significant Objects After Image Segmentation

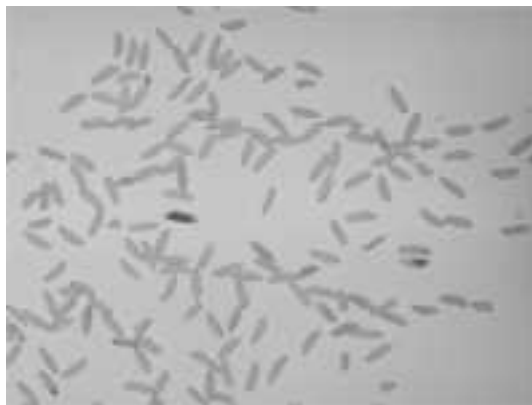
["Image Segmenter" on page 144](#)

##### **Objective**

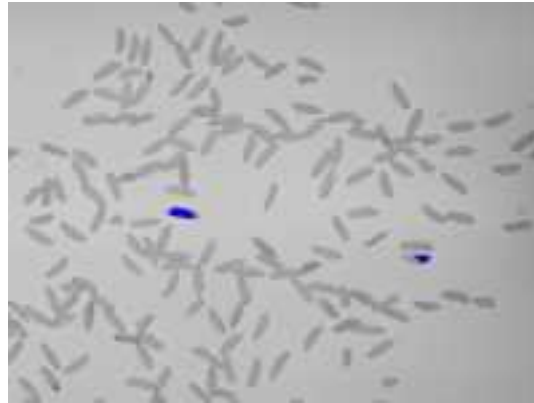
---

Following this tutorial, you will learn how to use EasyObject to detect bad rice grains (largely dark) among many normal rice grains (largely light).

You'll need first to load the source image (step 1). Then you'll perform the image segmentation, based on a threshold value (step 2). All the detected objects are dark, but some are too small to be significant. So, you'll set a minimum object area (number of pixels), and remove the smallest objects (step 3). Finally, you'll get only the objects that really represent bad rice grains.



Source image



Bad rice grains are detected

### Step 1: Load the source image

---

1. From the main menu, click **EasyObject**, then **New EasyObject Tool**.
2. Keep the default variable name for the new object, and click **OK**.
3. In the **Encoder** tab, click the **Open** icon of the **Source Image** area, and load the image file `EasyObject\Rice.jpg`.
4. Keep the default variable name for the new image, and click **OK**.

### Step 2: Perform image segmentation

---

1. In the **Encoder** tab, select the **Black Layer** check-box, and unselect the **White Layer** check-box.
2. Click the **...** button around the **Threshold** field. In the **Threshold** dialog box, select **Absolute**, enter '115', and click **OK**.
3. Click **Encode** to detect the dark objects. In the image, each object is drawn using a different color.
4. Click **Results** to display the list of all the detected objects. Clicking an object in the image highlights it in the list, and vice versa.

### Step 3: Remove the smallest objects

---

1. In the objects list, click **Columns**.
2. Tick the **Area** check-box, and click **OK**. In the list view, a new **Area** column appears, displaying each object area.
3. Click the **Area** column header to sort the objects. There are many small objects (area < 100) that may be considered as noise.
4. In the **Selection** tab, select **Area** from the **Feature** drop-down list. Select **'Less'** from the **Mode** drop-down list. In the **Threshold** field, enter '100'.
5. Click **Remove**. All the objects with an area smaller than 100 pixels have been removed from the list. The two remaining objects are the bad rice grains.

# Detecting Differences Between Images Using Min-Max References

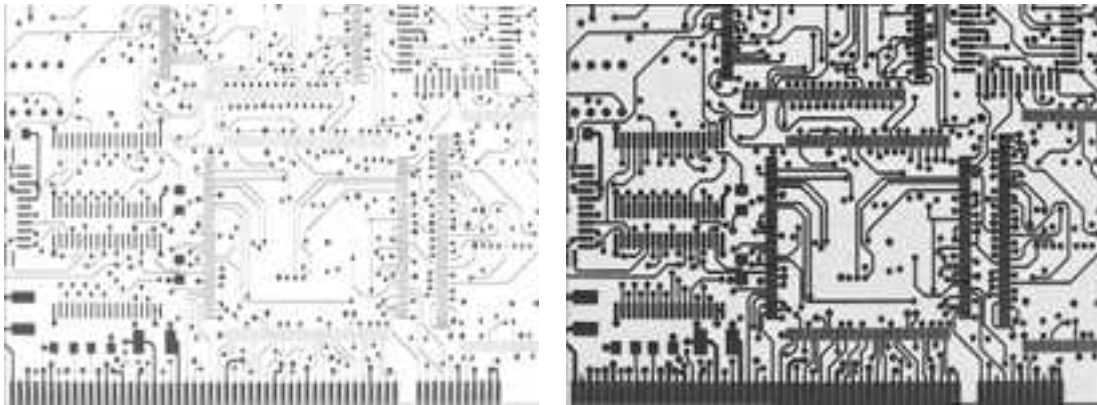
"Selecting and Sorting Blobs" on page 147

## Objective

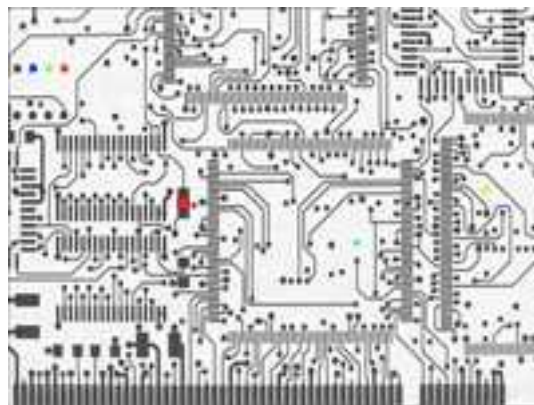
---

Following this tutorial, you will learn how to use EasyObject to compare images. In this example, we will check the quality of a PCB film.

You'll need first to load a reliable source image (step 1), from which two reference images (min and max) will be built (step 2). Then you'll load another image to be inspected (step 3), and perform the comparison with the min and max reference images (step 4). The differences will be detected.



High (left) and low (right) threshold reference images



In another image, differences are detected

## Step 1: Load the source image

---

1. From the main menu, click EasyObject, then Make Min Max.
2. Click the Open icon of the Source Image area, and load the image file `EasyObject\Film0k.png`.

3. Enter 'filmOk' for the name of the new image, and click OK.

## Step 2: Build min and max reference images

---

1. Click Execute. Min and Max reference images are created, based on the source image.
  - `filmOkMax` is computed by dilating `filmOk` a given number of times ('geometric margin') and adding a constant ('gray level margin') to every pixel. `filmOkMin` is computed by eroding `filmOk` the same number of times and subtracting the same constant to every pixel.
  - The geometric margin can be seen as a position tolerance between the image to be inspected and the reference.
  - The gray level margin introduces a tolerance to lighting variations.

## Step 3: Load an image to be inspected

---

1. From the main menu, click EasyObject, then New EasyObject Tool.
2. Keep the default variable name for the new object, and click OK.
3. In the Encoder tab, click the Open icon of the Source Image area, and load the image file `EasyObject\FilmBad.png`.
4. Enter 'filmBad' for the name of the new image, and click OK

## Step 4: Compare the image with the reference images

---

1. In the Encoder tab, select ImageRange in the segmentation method drop-down list.
2. Disable the White Layer check-box, and enable the Black Layer check-box.
3. Click the ... button around the High Image field. Select `filmOkMax` in the drop-down list, and click OK.
4. Click the ... button around the Low Image field. Select `filmOkMin` in the drop-down list, and click OK.
5. Click Encode. Eight differences are highlighted in the image.
6. Click Results to get further information about the detected objects. They may be further filtered and analyzed using object features selection and sorting capabilities of EasyObject.

# Detecting Printing Errors Using a Flexible Mask

["Generating a Flexible Mask from an Encoded Image" on page 148](#)

## Objective

---

Following this tutorial, you will learn how to use a flexible mask to target and search specific areas in the image.

You'll need first to load a source image (step 1), and a flexible mask image (step 2), that can be automatically applied on the source image to separate do-care areas (that must be considered) and don't-care areas (that should not be considered). Then, you'll perform the inspection only on do-care areas (step 3).



Source image



Mask image



Results

### Step 1: Load the source image

1. From the main menu, click EasyObject, then New EasyObject Tool.
2. Keep the default variable name for the new object, and click OK.
3. In the Encoder tab, click the Open icon, and load the image file EasyObject\Mobile3.jpg.



4. Keep the default variable name for the new image, and click OK.

## Step 2: Load the flexible mask image

---

1. In the Encoder tab, click the Open icon, and load the flexible mask image file `EasyObject\MobileMask.bmp`.
2. Enter 'mask' for the name of the new image, and click OK. In the Mask area of the Encoder tab, notice that the mask image is selected from the drop-down list: the mask is automatically applied on the source image, because its name contains 'mask', and because it has been loaded from the coded image dialog box. The source image preview in the dialog box shows (in red diagonal lines) the don't-care area, that is the area that will be not be considered when encoding the source image.

## Step 3: Inspect the image

---

1. In the Segmentation area of the Encoder tab, click the ... button to display the threshold dialog box.
2. Select Absolute and enter 202 for the threshold. Click OK to close the dialog box.
3. Click Encode to locate the objects (in the do-care areas only). In the source image, each object is drawn using a different color. Three printing errors can be observed:
  - The digit '7' is partially printed.
  - The '+' sign is missing.
  - The handset is printed on a lighter tone.
4. Click Results to display the statistics on each object.
  - Selecting an object in the list highlights it in the image.
  - Selecting Columns and Drawing will display more options.

## 5.2. EasyGauge

### Measuring the Rotation Angle of an Object

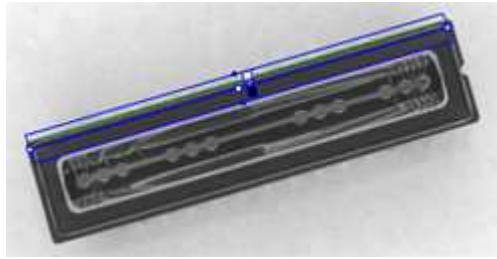
["Line Fitting" on page 151](#)

#### Objective

---

Following this tutorial, you will learn how to use EasyGauge to measure the rotation angle of a CCD sensor package. As we only need to retrieve an angle value, it's not required to work in a calibrated field of view. All geometrical parameters and results will be express as numbers of pixels.

You'll need to load the source image (step 1), and attach a line fitting gauge (step 2). The inspection is automatically performed (step 3).



Line fitting gauge

### Step 1: Load the source image

---

1. From the main menu, click EasyGauge, then New World Shape.
2. Keep the default variable name, and click OK.
3. From the Gauges tab, click the Open icon, and load the image file `EasyImage\CCD.tif`.
4. Keep the default variable name for the new image object, and click OK.

### Step 2: Attach a line gauge to the image

---

1. In the Gauges tab of the world shape dialog box, right-click the world shape icon, select **New > Line Gauge** from the contextual menu.
2. Keep the default variable name, and click OK. The line location gauge appears on the image. It consists of the following elements:
  - A blue line segment along which the transitions search is carried out.
  - Five white handles, allowing the user to move and rotate the segment.
  - A gray arrow, indicating in which direction the segment is traversed.
  - Black and white rectangles, indicating which kind of transition is searched for.
  - Green line, indicating the transition points found (if any).
3. Using the handles, move, rotate and extend the line gauge, so that it is positioned on the upper edge of the CCD package, with the gray arrow pointing downwards.
4. In the Measurement tab of the line gauge dialog box, choose 'White to Black' from the Type dropdown list and 'From Begin' from the Choice dropdown list.

### Step 3: Perform the inspection

---

1. The image is automatically inspected. However, clicking **Process** in the world shape dialog box will insert the corresponding code into the script window.
2. Click the **Results** tab to retrieve the measured angle value.
3. To see the individual fitting points, select the **Points** checkbox under the Draw Samples area.

## Measuring the Diameter of a Circle

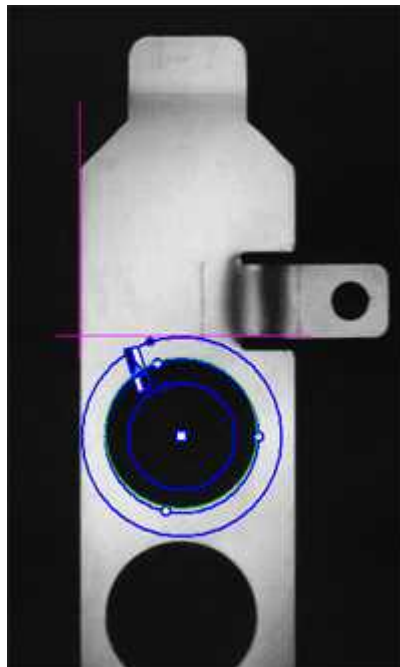
["Circle Fitting" on page 152](#)

## Objective

---

Following this tutorial, you will learn how to use EasyGauge to measure the diameter of a circle in an image.

You'll first load an image for calibration—a dot grid—(step 1), and calibrate the field of view (step 2). Then you'll load the source image for inspection (step 3), and attach a circle fitting gauge (step 4). The inspection is automatically performed (step 5). All measurement results are expressed in physical units..



Measuring the diameter of a circle

### Step 1: Load the calibration image

---

1. From the main menu, click EasyGauge, then New World Shape.
2. Keep the default variable name, and click OK.
3. In the Dot Grid Calibration tab, click the Open icon, and load the image file `EasyGauge\Dot Grid.tif`.
4. Keep the default variable name for the new image object, and click OK.

### Step 2: Calibrate the field of view

---

- Click Calibrate. From now on, the field of view is calibrated, and all results will be expressed in physical units.

### Step 3: Load the source image

---

1. From the Gauges tab, click the Open icon, and load the image file `EasyGauge\Bracket 6.tif`.
2. Keep the default variable name, and click OK.

## Step 4: Attach a circle gauge to the image

---

1. In the Gauges tab of the world shape dialog box, right-click the frame shape icon, select **New > Circle Gauge** from the contextual menu.
2. Keep the default variable name, and click **OK**.
3. The circle location gauge appears on the image. It consists of the following elements:
  - A blue ring in which the circle is searched for.
  - A blue nominal circle.
  - Six white handles, allowing the user to move and rotate the segment.
  - A gray arrow, indicating in which direction the segment is traversed.
  - Black and white rectangles, indicating which kind of transition are searched for.
  - A green arc of circle, indicating the circle found (if any).
4. Using the handles, drag the circle fitting gauge around the upper bracket hole. Adjust the nominal circle on the hole edge and extend the searching area if necessary.
5. In the **Measurement** tab of the circle gauge dialog box, select 'From Begin' from the Choice dropdown list.

## Step 5: Perform the inspection

---

1. The image is automatically inspected. However, clicking **Process** in the world shape dialog box will insert the corresponding code into the script window.
2. Click the **Results** tab to retrieve the measured diameter. All measurement results are expressed in physical units.

# Measuring a Distorted Rectangle

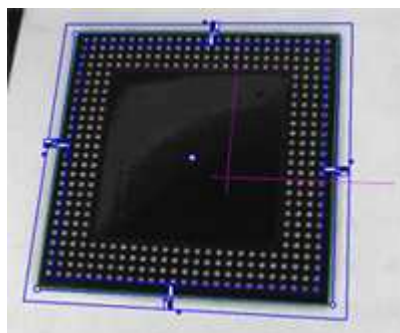
["Rectangle Fitting" on page 153](#)

## Objective

---

Following this tutorial, you will learn how to use EasyGauge to perform measurements on a distorted rectangle component.

To obtain measurement results in physical units, we need to work in a calibrated field of view. You'll need first to load an image for calibration —a dot grid— (step 1), and calibrate the field of view (step 2). Then you'll load the distorted image (step 3), and attach a rectangle fitting gauge (step 4). The inspection is automatically performed (step 5). All measurement results are expressed in physical units.



## Measuring a distorted rectangle

### Step 1: Load the calibration image

---

1. From the main menu, click EasyGauge, then New World Shape.
2. Keep the default variable name, and click OK.
3. In the Dot Grid Calibration tab, click the Open icon, and load the image file EasyGauge\Dot Grid 5.tif. This dot grid has been acquired in the same conditions and has the same distortion as the image we want to inspect.
4. Keep the default variable name for the new image object, and click OK.

### Step 2: Calibrate the field of view

---

- Click Calibrate. From now on, the field of view is calibrated, and all results will be expressed in physical units.

### Step 3: Load the distorted image

---

1. From the Gauges tab, click the Open icon, and load the image file EasyGauge\Distorted Component.tif.
2. Keep the default variable name, and click OK.

### Step 4: Attach a rectangle gauge to the image

---

1. In the Gauges tab, right-click the world shape icon, and select New > Rectangle Gauge from the contextual menu.
2. Keep the default variable name, and click OK. The rectangle gauge dialog box is opened, and the rectangle gauge is drawn on the image. It consists of the following elements:
  - A blue rectangular ring in which the rectangle is searched for.
  - A blue nominal rectangle.
  - Eleven white handles, allowing the user to move and extend the search area.
  - Gray arrows, indicating in which direction segments are examined.
  - Black and white rectangles, indicating which kind of transition are searched for.
  - A green rectangle, indicating the rectangle found (if any).
3. Due to the perspective effect, the rectangle gauge doesn't look like a rectangle. Using the central handle, move the rectangle gauge in the image and observe the rectangle deformation. Due to the calibration, the rectangle gauge shape adapts to the field of view deformation. Extend the searching area, and adjust the nominal rectangle on the component edges.
4. In the Measurement tab of the rectangle gauge dialog box, select 'White To Black' from the Type dropdown list and 'From Begin' from the Choice dropdown list.

### Step 5: Perform the inspection

---

1. The image is automatically inspected. However, clicking Process in the world shape dialog box will insert the corresponding code into the script window.
2. Click the Results tab to retrieve the measured X and Y sizes. All measurement results are expressed in physical units.

# Locating Points Regarding to a Coordinate System

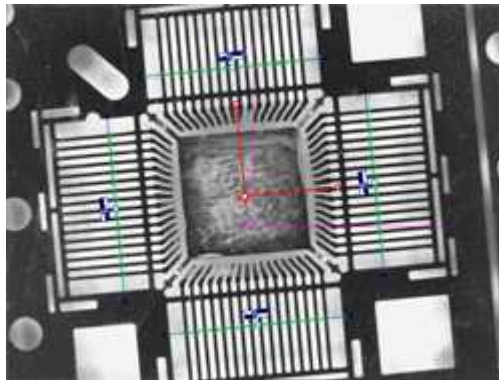
"Point Location" on page 151

## Objective

---

Following this tutorial, you will learn how to use EasyGauge to perform lead frames inspection. This operation determines the dimension, position, curvature, size, angle or diameter of the lead frames with an excellent accuracy. Robustness is ensured by powerful edge-point selection mechanisms that are intuitive and easy to tune, allowing measurement in cluttered images.

You'll first load an image for calibration —a dot grid— (step 1), and calibrate the field of view (step 2). Then you'll load the lead frame image (step 3), and set a coordinate system (a frame shape). Regarding to this coordinate system, you can define point gauges (steps 5-6). Finally, you'll load another lead frame image, that has a slight angle deviation, so the coordinate system has to be rotated (steps 7-8). The inspection is then automatically performed (step 9). All measurement results are expressed in physical units.



Four point gauges over four sets of leads

## Step 1: Load the calibration image

---

1. From the main menu, click EasyGauge, then New World Shape.
2. Keep the default variable name, and click OK.
3. In the Dot Grid Calibration tab, click the Open icon, and load the image file EasyGauge\Dot Grid 2.tif.
4. Keep the default variable name for the new image object, and click OK.

## Step 2: Calibrating the field of view

---

1. Click Calibrate. From now on, the field of view is calibrated, and all results will be expressed in physical units.

## Step 3: Loading a lead frame image

---

1. From the Gauges tab, click the Open icon, and load the image file EasyGauge\Lead Frame 1.tif.
2. Keep the default variable name, and click OK.

## Step 4: Setting a coordinate system

---

1. In the **Gauges** tab of the world shape dialog box, right-click the world shape icon, select **New > Frame Shape** from the contextual menu.
2. Keep the default variable name, and click **OK**. The frame shape icon appears in the world shape dialog box.
3. Drag the frame shape center approximately to the square center of the image.

## Step 5: Attaching a point gauge to the frame shape

---

1. In the **Gauges** tab of the world shape dialog box, right-click the frame shape icon, select **New > Point Gauge** from the contextual menu.
2. Keep the default variable name, and click **OK**. The point location gauge appears on the image. It consists of the following elements:
  - A blue line segment along which the transitions search is carried out.
  - Three white handles, allowing the user to move and rotate the segment.
  - A gray arrow, indicating in which direction the segment is traversed.
  - Black and white rectangles, indicating which kind of transition is searched for.
  - Green crosses, indicating the transition points found (if any).
3. Place the point location gauge over a set of leads: in the **Position** tab of the point gauge dialog box, set the **Center Y** property to 7, and the **Tolerance** property to 5.

## Step 6: Attaching other point gauges to the frame shape

---

1. From the **Gauges** tab of the world shape dialog box, create three other point gauges (refer to step 5).
2. Place these point location gauges over the remaining sets of leads :
  - Center Y = -7, Tolerance = 5
  - Center X = 7, Tolerance = 5, Angle = 90
  - Center X = -7, Tolerance = 5, Angle = 90

## Step 7: Loading another lead frame image

---

1. From the **Gauges** tab, click the **Open** icon, and load the image file `EasyGauge\Lead Frame 2.tif`.
2. Keep the default variable name for the new image, and click **OK**.

## Step 8: Tuning the coordinate system

---

1. In the frame shape dialog box, set the **Angle** property to 5.8.
2. Drag the frame shape center approximately to the square center of the image. All point location gauges automatically follow.

## Step 9: Performing the inspection

---

1. The image is automatically inspected. However, clicking **Process** in the world shape dialog box will insert the corresponding code into the script window.

2. From the point gauge dialog boxes, click the Results tab to retrieve the located points coordinates. All measurement results are expressed in physical units. The values refer to the frame shape system.

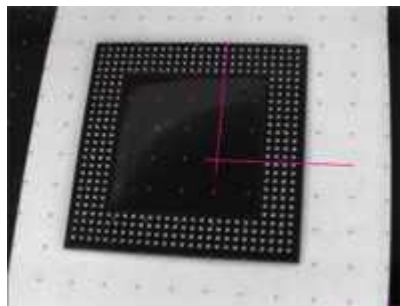
## Unwarping a Distorted Image

### Objective

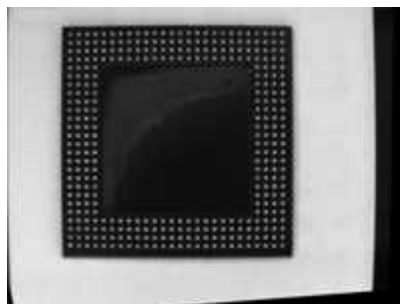
---

Following this tutorial, you will learn how to use EasyGauge to perform grid calibration, and unwrap a distorted image.

You'll first load an image for calibration—a dot grid— (step 1), and calibrate the field of view (step 2). Then you'll load the distorted image (step 3), and perform the unwarping operation (step 4).



Distorted image



Unwarped image

### Step 1: Load the calibration image

---

1. From the main menu, click EasyGauge, then New World Shape.
2. Keep the default variable name, and click OK.
3. In the Dot Grid Calibration tab, click the Open icon, and load the image file EasyGauge\Dot Grid 5.tif. This dot grid has been acquired in the same conditions and has the same distortion as the image we want to unwrap.
4. Keep the default variable name for the new image object, and click OK.



## Step 2: Calibrate the field of view

---

- Click Calibrate.

## Step 3: Load the distorted image

---

1. From the Unwarping tab, click the Open icon, and load the image file `EasyGauge\Distorted component.tif`.
2. Keep the default variable name, and click OK.

## Step 4: Unwarp the distorted image

---

1. In the Destination Image area, click New icon to create a new image.
2. Keep the default image settings, and click OK.
3. Select Interpolate checkbox to improve resulting image quality.
4. Click Unwarp. In the destination image, all distortions are corrected.

## 5.3. EasyFind

### Detecting Highly-Degraded Occurrences of a Reference Model in Multiple Files

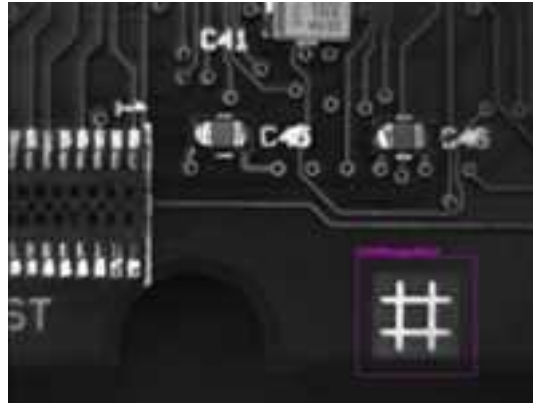
["Pattern Finding and Retrieving Results" on page 160](#)

#### Objective

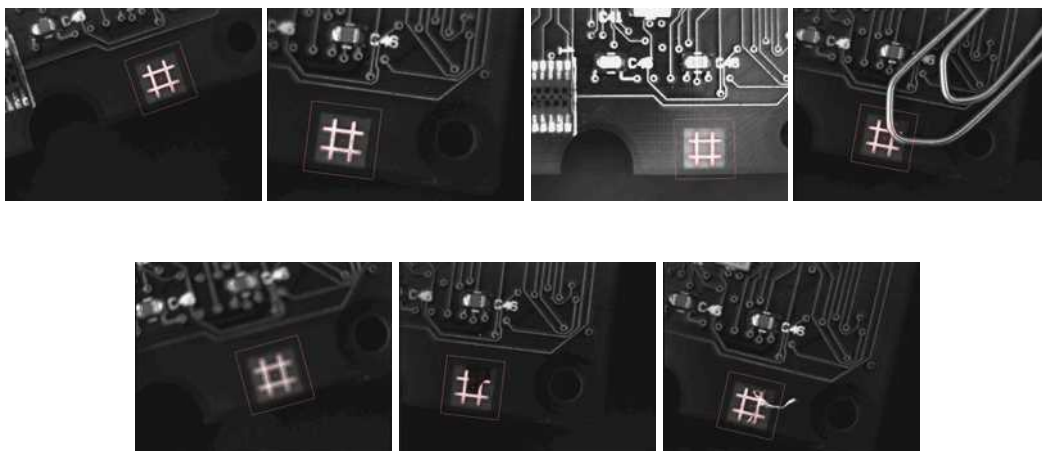
---

Following this tutorial, you will learn how to use EasyFind to detect in multiple images highly-degraded occurrences of a reference model. The degradation can be due to noise, blur, occlusion, missing parts or unstable illumination conditions.

You'll need first to load a reference image, define an ROI where EasyFind will learn the reference model, and set rotation and scaling tolerances for the expected occurrences to search for (steps 1-4). Then you're ready to open multiple files, and perform automatic detection of occurrences (even highly-degraded) of the reference model (steps 5-6).



Reference model



Occurrences of the reference model are found, even if highly-degraded

### Step 1: Load the reference image

---

1. From the main menu, click EasyFind, then New EasyFind Tool.
2. Keep the default variable name for the new PatternFinder object, and click OK. The PatternFinder management dialog box is opened.
3. In the Model tab, click the Open icon, and load the image file EasyFind\Fiducial \*.tif.
4. Keep the default variable name for the new image object, and click OK.

### Step 2: Create an ROI to define the reference model on the reference image

---

1. In the image, right-click and select New ROI... item from the menu.
2. Keep the default variable name for the new ROI object, and click OK. A default ROI is placed over the image (blue rectangle with handles). The ROI management dialog box is opened.
3. Drag the ROI over the reference model and resize it using its handles. Alternatively, enter the following coordinates in the ROI dialog box: 500, 365, 170, 170 for OrgX, OrgY, Width, and Height respectively, and click Close.

### Step 3: Learn the reference model

---

1. In the PatternFinder **Model** tab, select the ROI object from the source image drop-down list, and click **Learn**. The reference model is perfectly detected (green edges).
2. In the PatternFinder **Search Field** tab, select the Image object from the source image drop-down list. Tick the **Draw Features** check-box.

The model location and feature points are highlighted in the source image.

### Step 4: Set rotation and scaling tolerances

---

- In the PatternFinder **Allowances** tab, set both angle tolerance and scale tolerance to 25.0.

### Step 5: Select multiple images

---

1. In the PatternFinder **Search Field** tab, click the **Open** icon. Select the images files `EasyFind\Fiducial 2.tif` to `Fiducial 8.tif` (use the shift key to select multiple files), and click **Open**.
2. Keep the default variable name for the new Image object, and click **OK**. The last image appears. The reference model is found, even if highly-degraded.
3. Detection of the reference model is automatically performed. It is not necessary to click **Find** once a new image appears, as inspection is automatically performed. However, clicking **Find** will insert the corresponding code into the script windows.
4. Click **Results** to find more information about the found instance.

### Step 6: Browse multiple images

---

- In the PatternFinder **Search Field** tab, click the **Load Next** or **Load Previous** icons.

The image files appear, and the reference model is automatically detected.

## Improving the Score of Found Instances by Using "Don't Care Areas"

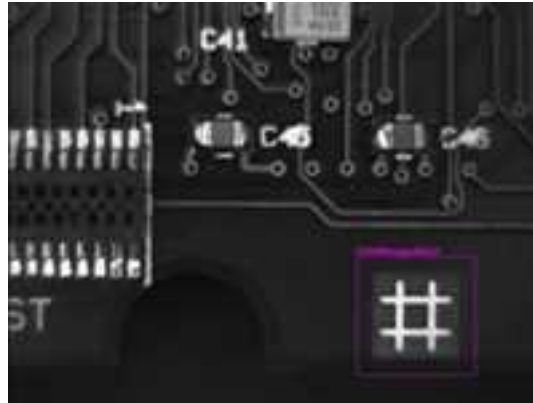
["Setting Search Parameters" on page 159](#)

### Objective

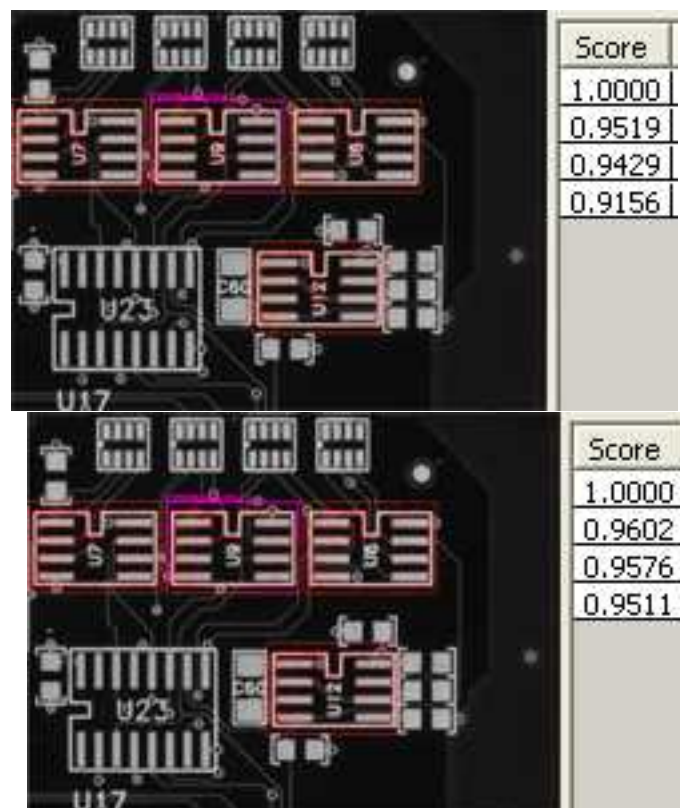
---

Following this tutorial, you will learn how to use EasyFind to handle "don't care areas" in geometric pattern matching. "Don't care areas" help to define in the image the meaningful features only, by masking the areas that might change from image to image, such as text and numbers.

You'll need first to load a reference image, define an ROI where EasyFind will learn the reference model, and set a rotation tolerance for the expected instances to search for (steps 1-4). Then you're ready to perform automatic detection of instances of the reference model, without using "don't care areas" (step 5). As the matching scores of the found instances are not high enough, you'll define a "don't care area" on the reference model, and restart the detection. The matching scores are slightly better (steps 6-7).



Reference model



Found instances and matching scores, without (left) and with (right) using "don't care areas"

## Step 1: Loading the reference image

---

1. From the main menu, click EasyFind, then New EasyFind Tool.
2. Keep the default variable name for the new PatternFinder object, and click OK. The PatternFinder management dialog box is opened.
3. In the Model tab, click the Open icon, and load the image file EasyFind\Solder Pad ▶.tif.
4. Keep the default variable name for the new Image object, and click OK.

## Step 2: Creating an ROI to define the reference model on the reference image

---

1. In the image, right-click and select **New ROI...** item from the menu.
2. Keep the default variable name for the new ROI object, and click **OK**. A default ROI is placed over the image (blue rectangle with handles). The ROI management dialog box is opened.
3. Drag the ROI over the reference model and resize it using its handles. Alternatively, enter the following coordinates in the ROI dialog box: 200, 130, 190, 130 for **OrgX**, **OrgY**, **Width**, and **Height** respectively, and click **Close**.

## Step 3: Learning the reference model

---

1. In the PatternFinder **Model** tab, select the ROI object from the source image drop-down list, and click **Learn**. The reference model is detected.
2. In the PatternFinder **Search Field** tab, select the **Image** object from the source image drop-down list. Tick the **Draw Features** check-box. The model location and feature points are highlighted in the source image.

## Step 4: Setting a rotation tolerance

---

1. In the PatternFinder **Allowances** tab, set the angle tolerance to 5.0.

## Step 5: Detecting instances of the reference model without "don't care areas"

---

1. In the PatternFinder **Search Field** tab, set **Max Instances** to 4, and click **Find**. The instances matching the reference model are highlighted.
2. Click **Results** to see the matching score of each found instance. Even though the scores are good, we can still improve them slightly by using a "don't care area" to mask the text appearing in the learned pattern.

## Step 6: Defining the "don't care area"

---

1. In the PatternFinder **Don't Care Areas** tab, select the **Rectangle** radio button from the **Blacken Inside** group.
2. In the ROI defining the reference model, move your mouse pointer on the top-left corner of the text "U9", left-click and drag the don't care area (rectangle with red stripes) to mask out the text.

## Step 7: Detecting instances of the reference model with "don't care areas"

---

1. In the PatternFinder **Don't Care Areas** tab, click **Learn**, and then click **Find**.

The instances matching the reference model are still highlighted, but the text is not found anymore.

2. Click **Results** to compare the new matching scores.

They are slightly better.

## 5.4. EasyMatch

### Learning a Pattern and Creating an EasyMatch Model File

["Pattern Learning" on page 161](#)

#### Objective

---

Following this tutorial, you will learn how to use EasyMatch to learn a model from a reference image, and save it as an EasyMatch model file.

You'll need first to load the reference image (step 1). Then, you'll learn it as the reference model (step 2). And you'll save the model as an EasyMatch model file (step 3).



Reference image

#### Step 1: Load the reference image

---

1. From the main menu, click **EasyMatch**, then **New Match Tool**.
2. Keep the default variable name for the new matcher object, and click **OK**.
3. In the **Learning** tab, click the **Open** icon, and load the image file `EasyMatch\FrameModel.tif`.
4. Keep the default variable name for the new image object, and click **OK**.

#### Step 2: Learn the reference image

---

- In the **Learning** tab, click **Learn** to acquire the model pattern.

#### Step 3: Save the model file

---

1. In the **Learning** tab, click the **Save As...** button.
2. Type a file name for the new EasyMatch model file. Its extension will be `.mch`.
3. Click **Save**.

### Matching a Pattern According to a Model File

["Pattern Matching and Retrieving Results" on page 162](#)

#### Objective

---

Following this tutorial, you will learn how to use EasyMatch to load an EasyMatch model file, and search for occurrences of the pattern in an image.

You'll need first to load the model file, and a source image where the model will be searched for (steps 1-2). Then the pattern matching is fully automatic (step 3).



Occurrences of the model are automatically highlighted

### Step 1: Load the model file

---

1. From the main menu, click **EasyMatch**, then **New Match Tool**.
2. Keep the default variable name for the new matcher object, and click **OK**.
3. In the **Learning** tab, click **Load...** to open the model file `EasyMatch\Switch.MCH`. The model contains all necessary information about the pattern we are searching for.

### Step 2: Load a source image

---

1. In the **Matching** tab, click the **Open** icon, and load the image file `EasyMatch\Switch.tif`.
2. Keep the default variable name for the new image object, and click **OK**.

### Step 3: Perform the pattern matching

---

1. The pattern matching is automatically performed on the source image, and the matching occurrences are highlighted. Clicking **Execute** will insert the corresponding code into the script windows.
2. Further information about each occurrence can be found by clicking **Results**.
3. Click in a row to see the corresponding occurrence highlighted in the image.

## Learning a Pattern According to an ROI

"Pattern Learning" on page 161



## Objective

---

Following this tutorial, you will learn how to use EasyMatch to learn a model from an ROI in a source image, and to perform pattern matching on the same image.

You'll need first to load the source image, and define an ROI inside (steps 1-2). Then, you'll have to learn the model, using this ROI (step 3). Finally, you'll perform pattern matching in the source image (step 4), and will find additional occurrences of the model.



ROI that will be learned



Occurrences matching the model ROI

### Step 1: Load the source image

---

1. From the main menu, click **Image**, then **Open**.
2. Load the image file `EasyMatch\BOARD.JPG`.
3. Keep the default variable name for the new image object, and click **OK**.

### Step 2: Define an ROI

---

1. Right-click in the image, and select **New ROI...** from the contextual menu.



2. Keep the default variable name for the new ROI object, and click **OK**. A default ROI is placed over the image (blue rectangle with handles).

The ROI management dialog box is opened.

3. Resize the ROI and move it around one of the blue capacitors at the lower left part of the image.

### Step 3: Learn a model from the ROI

---

1. From the main menu, click **EasyMatch**, then **New Match Tool**.
2. Keep the default variable name for the new matcher object, and click **OK**.
3. In the **Learning** tab of the matcher dialog box, select the ROI object from the **Source Image** drop-down list, and click **Learn** to acquire the model pattern.

### Step 4: Match the pattern

---

1. In the **Matching** tab, increase the **Max Occurrences** field to 2.
2. Select the image object from the **Source Image** drop-down list.
3. Click **Execute**. The occurrences of the learned model are highlighted in the source image.
4. Further information about each occurrence can be found by clicking **Results**.
5. Click in a row to see the corresponding occurrence highlighted in the image.

## Improving the Score of Matching Instances by Using "Don't Care Areas"

["Setting Search Parameters" on page 161](#)

### Objective

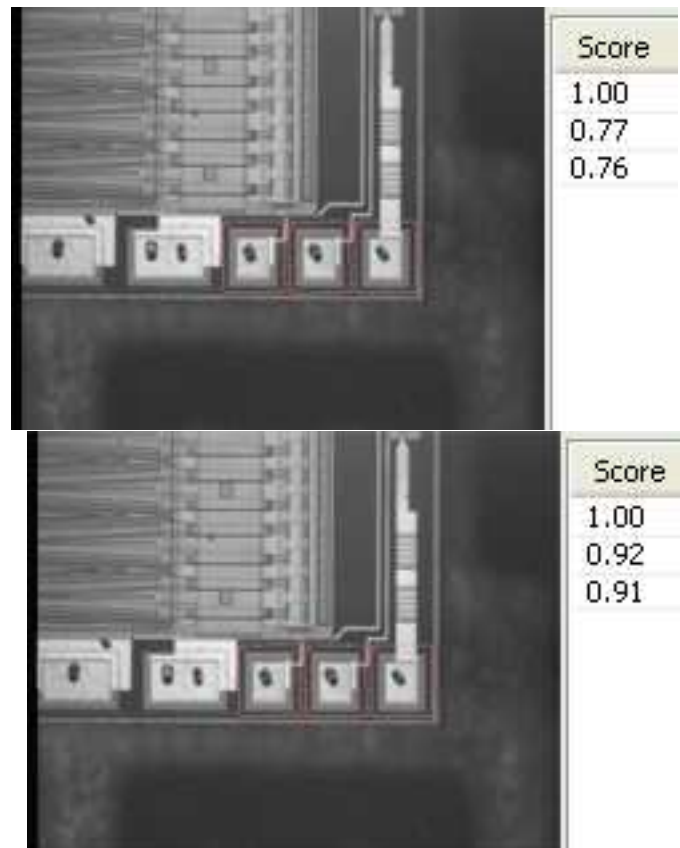
---

Following this tutorial, you will learn how to use EasyMatch to handle "don't care areas". "Don't care areas" help to define in the image the meaningful features only, by masking the areas that might change from image to image.

You'll need first to load a reference image and learn the reference model (steps 1-2). Then you'll perform automatic pattern matching of instances of the reference model, without using "don't care areas" (step 3). As the matching scores of the found instances are not high enough, you'll define a "don't care area" on the reference model, and restart the detection. The matching scores are much better (steps 4-5).



Reference model



Found instances and matching scores, without (left) and with (right) using "don't care areas"

### Step 1: Load the reference image

---

1. From the main menu, click EasyMatch, then New Match Tool.
2. Keep the default variable name for the new Matcher object, and click OK. The Matcher management dialog box is opened.
3. In the Learning tab, click the Open icon, and load the image file EasyMatch\Die Pad Model ↵.bmp.
4. Keep the default variable name for the new Image object, and click OK.

### Step 2: Learn the reference model

---

- Click Learn to acquire the model pattern.

### Step 3: Detect instances of the reference model without "don't care areas"

---

1. In the Matching tab, click the Open icon, and load the image file EasyMatch\Die Pad ↵.bmp.
2. Keep the default variable name for the new Image object, and click OK. An instance matching the reference model is highlighted.
3. Increase the Max Occurrences field to 3. Enable the Sub-Pixel Interpolate check-box to increase the matching precision.

4. Enter '-10.0' as the Minimum Angle (Deg). (Check that the angle is displayed in degrees. If not, select the angles unit from **View > Option** menu.)
5. Enter '10.0' as the Maximum Angle (Deg).
6. Click **Execute**. The pattern locations are highlighted in the source image.
7. Click **Results** to see the matching score of each found instance. The last two scores are rather bad. This is mainly due to the bright rectangle on the upper part of the reference image we have learned. We can improve the scores by using a "don't care area" to mask this bright area.

#### **Step 4: Define the "don't care area"**

---

1. In the **Don't Care Areas** tab, select the **Rectangle** radio button from the **Blacken Inside** group.
2. In the reference image, move your mouse pointer on the lower left corner of the bright rectangle, left-click and drag the don't care area (rectangle with red stripes) to the upper right corner of the bright rectangle to mask out this area.
3. In the **Don't Care Areas** tab, click **Learn**.

#### **Step 5: Detect instances of the reference model with "don't care areas"**

---

1. In the **Matching** tab, click **Execute**. The instances matching the reference model are still highlighted.
2. Click **Results** to compare the new matching scores. They are much better.

## 6. Code Snippets

## 6.1. Basic Types

### Loading and Saving Images

Functional Guide | Reference: [Load](#), [Save](#), [SaveJpeg](#)

```

////////////////////////////////////
// This code snippet shows how to load and save an image. //
////////////////////////////////////

// Images constructor
EImageBW8 srcImage;
EImageBW8 dstImage;

// Load an image file
srcImage.Load("mySourceImage.bmp");

// ...

// Save the destination image into a file
dstImage.Save("myDestImage.bmp");

// Save the destination image into a jpeg file
// The default compression quality is 75
dstImage.Save("myDestImage.jpg");

// Save the destination image into a jpeg file
// set the compression quality to 50
dstImage.SaveJpeg("myDestImage50.jpg", 50);

```

### Interfacing Third-Party Images

Functional Guide | Reference: [SetImagePtr](#)

```

////////////////////////////////////
// This code snippet shows how to link an Open eVision image //
// to an externally allocated buffer. //
////////////////////////////////////

// Images constructor
EImageBW8 srcImage;

// Size of the third-party image
int sizeX;
int sizeY;

//Pointer to the third-party image buffer
EBW8* imgPtr;

// ...

// Link the Open eVision image to the third-party image
// Assuming the corresponding buffer is aligned on 4 bytes
srcImage.SetImagePtr(sizeX, sizeY, imgPtr);

```

## Retrieving Pixel Values

Functional Guide | Reference: [GetImagePtr](#)

```

////////////////////////////////////
// This code snippet shows the recommended method (fastest) //
// to access the pixel values in a BW8 image //
////////////////////////////////////

EImageBW8 img;

OEV_UINT8* pixelPtr;
OEV_UINT8* rowPtr;
OEV_UINT8 pixelValue;
OEV_UINT32 rowPitch;
int x, y;

rowPtr = reinterpret_cast <OEV_UINT8*>(img.GetImagePtr());
rowPitch = img.GetRowPitch();

for (y = 0; y < height; y++)
{
    pixelPtr = rowPtr;

    for (x = 0; x < width; x++)
    {
        pixelValue = *pixelPtr;

        // Add your pixel computation code here

        *pixelPtr = pixelValue;
        pixelPtr++;
    }

    rowPtr += rowPitch;
}

```

## ROI Placement

Functional Guide | Reference: [Attach](#), [SetPlacement](#)

```

////////////////////////////////////
// This code snippet shows how to attach an ROI to an image //
// and set its placement. //
////////////////////////////////////

// Image constructor
EImageBW8 parentImage;

// ROI constructor
EROIBW8 myROI;

// ...

// Attach the ROI to the image
myROI.Attach(&parentImage);

//Set the ROI position
myROI.SetPlacement(50, 50, 200, 100);

```

# Vector Management

Functional Guide | Reference: [Empty](#), [AddElement](#)

```

////////////////////////////////////
// This code snippet shows how to create a vector, fill it //
// and retrieve the value of a given element. //
////////////////////////////////////

// EBW8Vector constructor
EBW8Vector ramp;

// Clear the vector
ramp.Empty();

// Fill the vector with increasing values
for(int i= 0; i < 28; i++)
{
    ramp.AddElement((EBW8)i);
}

// Retrieve the 9th element value
EBW8 value= ramp[9];

```

# Exception Management

Functional Guide | Reference: [GetPixel](#), [What](#)

```

////////////////////////////////////
// This code snippet shows how to manage //
// Open eVision exceptions. //
////////////////////////////////////

try
{
    // Image constructor
    EImageC24 srcImage;

    // ...

    // Retrieve the pixel value at coordinates (56, 73)
    EC24 value= srcImage.GetPixel(56, 73);
}

catch(Euresys::Open_eVision_::EException exc)
{
    // Retrieve the exception description
    std::string error = exc.What();
}

```

## 6.2. EasyObject

### Constructing the Blobs

#### Image Encoder

Functional Guide | Reference: [Encode](#), [SetBlackLayerEncoded](#), [SetWhiteLayerEncoded](#), [SetMode](#), [SetAbsoluteThreshold](#), [GetGrayscaleSingleThresholdSegmenter](#)

```

////////////////////////////////////
// This code snippet shows how to build blobs belonging to //
// the white layer according to the minimum residue method //
// and how to build blobs belonging to the black layer     //
// according to an absolute threshold.                     //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Build the blobs belonging to the white layer,
// the segmentation is based on the Minimum Residue method
encoder.Encode(srcImage, codedImage);

// Build the blobs belonging to the black layer,
// the segmentation is based on an absolute threshold (↔0)
Segmenters::EGrayscaleSingleThresholdSegmenter& segmenter= encoder.GetGrayscaleSingleThresholdSegmenter();
segmenter.SetBlackLayerEncoded(true);
segmenter.SetWhiteLayerEncoded(false);

segmenter.SetMode(EGrayscaleSingleThreshold_Absolute);
segmenter.SetAbsoluteThreshold(↔0);

encoder.Encode(srcImage, codedImage);

```

#### Image Segmenter

Functional Guide | Reference: [SetSegmentationMethod](#), [GetGrayscaleDoubleThresholdSegmenter](#), [SetHighThreshold](#), [SetLowThreshold](#)

```

////////////////////////////////////
// This code snippet shows how to build blobs according to //
// a user-defined image segmenter.                         //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Image encoder

```



```

EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Set the segmentation method to GrayscaleDoubleThreshold
encoder.SetSegmentationMethod(ESegmentationMethod_GrayscaleDoubleThreshold);

// Retrieve the segmenter object
Segmenters::EGrayscaleDoubleThresholdSegmenter& segmenter= encoder.GetGrayscaleDoubleThresholdSegmenter();

// Set the high and low threshold values
segmenter.SetHighThreshold(150);
segmenter.SetLowThreshold(50);

// Specify the layers to be encoded (neutral layer only)
segmenter.SetBlackLayerEncoded(false);
segmenter.SetNeutralLayerEncoded(true);
segmenter.SetWhiteLayerEncoded(false);

// Encode the image
encoder.Encode(srcImage, codedImage);

```

## Holes Extraction

Functional Guide | Reference: [GetHoleCount](#), [GetHole](#), [GetObjCount](#), [GetObj](#)

```

////////////////////////////////////
// This code snippet shows how to retrieve blobs' holes. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Encode the image
encoder.Encode(srcImage, codedImage);

// Retrieve holes for all the blobs
for (unsigned int blobIndex = 0; blobIndex < codedImage.GetObjCount(); blobIndex++)
{
    EObject& blob = codedImage.GetObj(blobIndex);

    // Browse the holes of the current object
    for (unsigned int holeIndex = 0; holeIndex < blob.GetHoleCount(); holeIndex++)
    {
        // Retrieve a given hole
        EHole& hole = blob.GetHole(holeIndex);
    }
}

```

## Continuous Mode

Functional Guide | Reference: [SetContinuousModeEnabled](#), [FlushContinuousMode](#)

```

////////////////////////////////////
// This code snippet shows how to build blobs //
// in the continuous mode context.           //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Enable the continuous mode
encoder.SetContinuousModeEnabled(true);

// Loop to acquire the different chunks
for (int count = 0; count < MAX_COUNT ; count++)
{
    // Store the new chunk into srcImage
    // ...

    // Encode the current chunk
    encoder.Encode(srcImage, codedImage);
}

// Flush the continuous mode
encoder.FlushContinuousMode(codedImage);

```

## Computing Blobs Features

Functional Guide | Reference: [GetGravityCenter](#), [GetObj](#)

```

////////////////////////////////////
// This code snippet shows how to retrieve blobs' features. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Encode the source image
encoder.Encode(srcImage, codedImage);

for (unsigned int index = 0; index < codedImage.GetObjCount(); index++)
{

```

```

// Retrieve the selected blob gravity center
EObject& blob = codedImage.GetObject(index);
float centerX = blob.GetGravityCenter().GetX();
float centerY = blob.GetGravityCenter().GetY();
}

```

## Selecting and Sorting Blobs

Functional Guide | Reference: [AddObjects](#), [ElementCount](#), [RemoveUsingUnsignedIntegerFeature](#), [Sort](#)

```

////////////////////////////////////
// This code snippet shows how to build blobs, select //
// some of them and sort the selected ones.          //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Encode the source image
encoder.Encode(srcImage, codedImage);

// Create a blob selection
EObjectSelection selection;
selection.AddObjects(codedImage);

// Remove the Small blobs
selection.RemoveUsingUnsignedIntegerFeature(EFeature_Area, 0, ESingleThresholdMode_Less);

// Retrieve the number of remaining blobs
unsigned int numBlobs= selection.GetElementCount();

// Sort the remaining blobs based on their area
selection.Sort(EFeature_Area, ESortDirection_Ascending);

// Retrieve the selected blobs
for (unsigned int index = 0; index < numBlobs; index++)
{
    float centerX= selection.GetElement(index).GetGravityCenterX();
    float centerY= selection.GetElement(index).GetGravityCenterY();
}

```

## Using Flexible Masks

## Constructing Blobs

Functional Guide | Reference: [Encode](#)

```

////////////////////////////////////
// This code snippet shows how to build blobs inside //

```

```
// a region defined by a flexible mask.          //
////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcImage;
EImageBW8 mask;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Encode the source image regions
// corresponding to the mask do care areas
encoder.Encode(srcImage, mask, codedImage);
```

## Generating a Flexible Mask from an Encoded Image

Functional Guide | Reference: [RenderMask](#)

```
//////////////////////////////////////
// This code snippet shows how to generate a flexible //
// mask from an encoded image.                       //
////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcImage;
EImageBW8 mask;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Encode the source image
encoder.Encode(srcImage, codedImage);

// The source image and the mask must have the same size
mask.SetSize(&srcImage);

// Create the mask based on the white layer
// of the coded image
codedImage.RenderMask(mask, ↵);
```

## Generating a Flexible Mask from a Blob Selection

Functional Guide | Reference: [RenderMask](#)

```
//////////////////////////////////////
// This code snippet shows how to generate a flexible //
// mask from a selection of blobs.                   //
////////////////////////////////////////////////////

// Images constructor
```

```

EImageBW8 srcImage;
EImageBW8 mask;

// Image encoder
EImageEncoder encoder;

// Coded image
ECodedImage2 codedImage;

// ...

// Encode the source image
encoder.Encode(srcImage, codedImage);

// The source image and the mask must have the same size
mask.SetSize(&srcImage);

// Create a blob selection
EObjectSelection selection;
selection.AddObjects(codedImage);

// Remove the Small blobs
selection.RemoveUsingUnsignedIntegerFeature(EFeature_Area, 100, ESingleThresholdMode_Less);

// Create the mask based on the blob selection
selection.RenderMask(mask);

// Sort the remaining blobs based on their area
selection.Sort(EFeature_Area, ESortDirection_Descending);

// Create the mask corresponding to the largest blob
selection.GetElement(0).RenderMask(mask);

```

## Using the Object Template Matcher

Functional Guide | Reference: [EObjectTemplateMatcher](#)

```

////////////////////////////////////
// This code snippet shows how to use EObjectTemplateMatcher//
// for alignment and template matching //
////////////////////////////////////

// Encode the template image
EImageEncoder encoder;
ECodedImage2 coded_img;

EImageBW8 template_img;
encoder.Encode(template_img, coded_img);

// Initialize EObjectTemplateMatcher
EObjectTemplateMatcher object_matcher;
object_matcher.SetEnableAlignment(true); // optional
object_matcher.SetMaximumDistance(60); // optional

// set the template
object_matcher.BuildTemplate(coded_img);

// Encode the test image
EImageBW8 test_img;
encoder.Encode(test_img, coded_img);

// Build a selection of test objects

```

```
EObjectSelection object_select;
object_select.AddObjects(coded_img);
object_select.RemoveUsingUnsignedIntegerFeature(EFeature_Area, 0, ESingleThresholdMode_Less); // optional
filter

// Perform the alignment and the matching
object_matcher.SortSelection(object_select);

// Display the number of matches
std::cout << object_matcher.GetNumberOfPairedObjects() << " paired objects" << std::endl;
// Retrieve the template indexes for each selection object
std::vector<int> template_indexes = object_matcher.GetTemplateIndexes();
```

## 6.3. EasyGauge

### Point Location

Functional Guide | Reference: [SetTransitionType](#), [SetTransitionChoice](#), [SetCenterXY](#), [SetTolerance](#), [Measure](#), [GetMeasuredPoint](#), [GetX](#), [GetY](#)

```

////////////////////////////////////
// This code snippet shows how to create a point location tool, //
// adjust the transition parameters, set the nominal gauge      //
// position, perform the measurement and retrieve the result.   //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// EPointGauge constructor
EPointGauge pointGauge;

// Adjust the transition parameters
pointGauge.SetTransitionType(ETransitionType_Wb);
pointGauge.SetTransitionChoice(ETransitionChoice_Closest);

// Set the gauge nominal position
pointGauge.SetCenterXY(256.f, 256.f);

// Set the gauge length to 100 units and the angle to 45°
pointGauge.SetTolerances(100.f, 45.f);

// Measure
pointGauge.Measure(&srcImage);

// Get the measured point coordinates
float measuredX = pointGauge.GetMeasuredPoint().GetX();
float measuredY = pointGauge.GetMeasuredPoint().GetY();

// Save the point gauge measurement context
pointGauge.Save("myPointGauge.gge");

```

### Line Fitting

Functional Guide | Reference: [SetTransitionType](#), [SetTransitionChoice](#), [SetTransitionIndex](#), [SetLine](#), [Measure](#), [GetMeasuredLine](#), [GetOrg](#), [GetEnd](#)

```

////////////////////////////////////
// This code snippet shows how to create a line measurement tool, //
// adjust the transition parameters, set the nominal gauge      //
// position, perform the measurement and retrieve the result.   //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// ELineGauge constructor
ELineGauge lineGauge;

// Adjust the transition parameters

```

```

lineGauge.SetTransitionType(ETransitionType_Bw);
lineGauge.SetTransitionChoice(ETransitionChoice_NthFromEnd);
lineGauge.SetTransitionIndex(2);

// Set the line fitting gauge position,
// length (50 units) and orientation (20°)
EPoint center(256.f, 256.f);
ELine line(center, 50.f, 20.f);
lineGauge.SetLine(line);

// Measure
lineGauge.Measure(&srcImage);

// Get the origin and end point coordinates of the fitted line
EPoint originPoint = lineGauge.GetMeasuredLine().GetOrg();
EPoint endPoint = lineGauge.GetMeasuredLine().GetEnd();

// Save the point gauge measurement context
lineGauge.Save("myLineGauge.gge");

```

## Circle Fitting

Functional Guide | Reference: [SetTransitionType](#), [SetTransitionChoice](#), [SetCircle](#), [Measure](#), [GetMeasuredCircle](#), [GetCenter](#), [GetRadius](#)

```

////////////////////////////////////
// This code snippet shows how to create a circle measurement tool, //
// adjust the transition parameters, set the nominal gauge          //
// position, perform the measurement and retrieve the result.      //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// ECircleGauge constructor
ECircleGauge circleGauge;

// Adjust the transition parameters
circleGauge.SetTransitionType(ETransitionType_Bw);
circleGauge.SetTransitionChoice(ETransitionChoice_LargestAmplitude);

// Set the Circle fitting gauge position, diameter (50 units),
// starting angle (↺0°), and amplitude (270°)
EPoint center(256.f, 256.f);
ECircle circle(center, 50.f, ↺0.f, 270.f);
circleGauge.SetCircle(circle);

// Measure
circleGauge.Measure(&srcImage);

// Get the center point coordinates and the radius of the fitted circle
float centerX = circleGauge.GetMeasuredCircle().GetCenter().GetX();
float centerY = circleGauge.GetMeasuredCircle().GetCenter().GetY();
float radius = circleGauge.GetMeasuredCircle().GetRadius();

// Save the point gauge measurement context
circleGauge.Save("myCircleGauge.gge");

```



## Rectangle Fitting

Functional Guide | Reference: [SetTransitionType](#), [SetTransitionChoice](#), [SetRectangle](#), [Measure](#), [GetMeasuredRectangle](#), [GetSizeX](#), [GetSizeY](#), [GetAngle](#)

```

////////////////////////////////////
// This code snippet shows how to create a rectangle measurement tool, //
// adjust the transition parameters, set the nominal gauge position, //
// perform the measurement and retrieve the result. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// ERectangleGauge constructor
ERectangleGauge rectangleGauge;

// Adjust the transition parameters
rectangleGauge.SetTransitionType(ETransitionType_Bw);
rectangleGauge.SetTransitionChoice(ETransitionChoice_LargestAmplitude);

// Set the rectangle fitting gauge position,
// size (50x30 units) and orientation (↙5°)
EPoint center(256.f, 256.f);
ERectangle rectangle(center, 50.f, 30.f, ↙5.f);
rectangleGauge.SetRectangle(rectangle);

// Measure
rectangleGauge.Measure(&srcImage);

// Get the size and the rotation angle of the fitted rectangle
float sizeX = rectangleGauge.GetMeasuredRectangle().GetSizeX();
float sizeY = rectangleGauge.GetMeasuredRectangle().GetSizeY();
float angle = rectangleGauge.GetMeasuredRectangle().GetAngle();

// Save the point gauge measurement context
rectangleGauge.Save("myRectangleGauge.gge");

```

## Wedge Fitting

Functional Guide | Reference: [SetTransitionType](#), [SetTransitionChoice](#), [SetWedge](#), [Measure](#), [GetMeasuredWedge](#), [GetInnerRadius](#), [GetOuterRadius](#)

```

////////////////////////////////////
// This code snippet shows how to create a wedge measurement tool, //
// adjust the transition parameters, set the nominal gauge //
// position, perform the measurement and retrieve the result. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// EWedgeGauge constructor
EWedgeGauge wedgeGauge;

// Adjust the transition parameters
wedgeGauge.SetTransitionType(ETransitionType_Bw);
wedgeGauge.SetTransitionChoice(ETransitionChoice_NthFromBegin);
wedgeGauge.SetTransitionIndex(0);

// Set the wedge fitting gauge position, diameter (50 units),

```

```
// breadth (-25 units), starting angle (0°) and amplitude (270°)
EPoint center(256.f, 256.f);
EWedge wedge(center, 50.f, -25.f, 0.f, 270.f);
wedgeGauge.SetWedge(wedge);

// Measure
wedgeGauge.Measure(&srcImage);

// Get the inner and outer radius of the fitted wedge
float innerRadius = wedgeGauge.GetMeasuredWedge().GetInnerRadius();
float outerRadius = wedgeGauge.GetMeasuredWedge().GetOuterRadius();

// Save the point gauge measurement context
wedgeGauge.Save("myWedgeGauge.gge");
```

## Gauge Grouping

### Gauge Hierarchy

Functional Guide | Reference: [Attach](#), [SetName](#), [Save](#)

```
////////////////////////////////////
// This code snippet shows how to create a gauge hierarchy //
// and save it into a file. //
////////////////////////////////////

// EWorldShape constructor
EWorldShape worldShape;

// Gauges constructor
ERectangleGauge rectangleGauge;
ECircleGauge circleGauge1, circleGauge2;

// ...

// Attach the rectangle gauge to the EWorldShape
rectangleGauge.Attach(&worldShape);

// Attach the circle gauges to the rectangle gauge
circleGauge1.Attach(&rectangleGauge);
circleGauge2.Attach(&rectangleGauge);

// Set the first circle gauge name
circleGauge1.SetName("myCircleGauge1");

// ...

// Save worldShape together with its daughters
worldShape.Save("myWorldShape.gge", true);
```

### Complex Measurement

Functional Guide | Reference: [Load](#), [GetNumDaughters](#), [Process](#), [GetDaughter](#), [GetShapeNamed](#)

```
////////////////////////////////////
// This code snippet shows how to trigger the measurement //
// of a whole gauge hierarchy and retrieve the results. //
////////////////////////////////////
```

```

// Image constructor
EImageBW8 srcImage;

// EWorldShape constructor
EWorldShape worldShape;

// Load the EWorldShape together with its daughters
worldShape.Load("myWorldShape.gge", true);

// Retrieve the number of worldShape's daughters
int numDaughters= worldShape.GetNumDaughters();

// ...

// Trigger the measurement of all the
// gauges attached to the EWorldShape
worldShape.Process(&srcImage, true);

// Retrieve the measurement result of
// the first daughter (a rectangle gauge)
ERectangleGauge* rectangleGauge= (ERectangleGauge*)worldShape.GetDaughter(0);
float sizeX= rectangleGauge->GetMeasuredRectangle().GetSizeX();

// Retrieve the measurement result of a
// daughter gauge called "myCircleGauge"
ECircleGauge* circleGauge= (ECircleGauge*)worldShape.GetShapeNamed("myCircleGauge");
EPoint center= circleGauge->GetMeasuredCircle().GetCenter();

```

## Calibration using EWorldShape

Functional Guide | [Reference](#)

## Calibration by Guesswork

Functional Guide | Reference: [SetSensor](#), [GetXResolution](#), [GetYResolution](#)

```

////////////////////////////////////
// This code snippet shows how to perform a calibration //
// by guesswork. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// EWorldShape constructor
EWorldShape worldShape;

// ...

// Compute the calibration coefficients
// Field of view: 32x24 mm
worldShape.SetSensor(srcImage.GetWidth(), srcImage.GetHeight(), 32.f, 24.f);

// Retrieve the spatial resolution
float resolutionX= worldShape.GetXResolution();
float resolutionY= worldShape.GetYResolution();

```

## Landmark-Based Calibration

Functional Guide | Reference: [EmptyLandmarks](#), [AddLandmark](#), [Calibrate](#)

```

////////////////////////////////////
// This code snippet shows how to perform a landmark-based //
// calibration.                                           //
////////////////////////////////////

// EWorldShape constructor
EWorldShape worldShape;

// ...

// Reset the calibration context
worldShape.EmptyLandmarks();

// Loop on the landmarks
for(int index= 0; index < numLandmarks; index++)
{
    // Get the I-th landmark as a pair of EPoint(x, y)
    EPoint sensorPoint, worldPoint;

    // Retrieve and store the relevant data into worldPoint and sensorPoint
    // ...

    // Add the I-th pair
    worldShape.AddLandmark(sensorPoint, worldPoint);
}

// Perform the calibration
worldShape.Calibrate(ECalibrationMode_Skewed);

```

## Dot Grid-Based Calibration

Functional Guide | Reference: [EmptyLandmarks](#), [AddPoint](#), [RebuildGrid](#), [AutoCalibrate](#)

```

////////////////////////////////////
// This code snippet shows how to perform a dot grid-based //
// calibration.                                           //
////////////////////////////////////

// EWorldShape constructor
EWorldShape worldShape;

// ...

// Reset the calibration context
worldShape.EmptyLandmarks();

// Loop on the dots
for(int index= 0; index < numDots; index++)
{
    // Get the I-th dot as an EPoint(x, y)
    EPoint dotPoint;

    // Retrieve and store the relevant data into dotPoint
    // ...

    // Add the I-th dot

```

```

worldShape.AddPoint(dotPoint);
}

// Reconstruct the grid topology
// pitch X and Y = 5 units
worldShape.RebuildGrid(5, 5);

// Perform the calibration
// the calibration modes are computed automatically
worldShape.AutoCalibrate(true);

```

## Coordinates Transform

Functional Guide | Reference: [SensorToWorld](#), [WorldToSensor](#)

```

////////////////////////////////////
// This code snippet shows how to convert coordinates from //
// the Sensor space to the World space and conversely.    //
////////////////////////////////////

// EWorldShape constructor
EWorldShape worldShape;

// EPoint constructor
EPoint sensor;
EPoint world;

// ...

// Perform the calibration
worldShape.Calibrate(ECalibrationMode_Scaled | ECalibrationMode_Skewed);

// Retrieve the world coordinates of a point, knowing its sensor coordinates
world= worldShape.SensorToWorld(sensor);

// Retrieve the sensor coordinates of a point, knowing its world coordinates
sensor= worldShape.WorldToSensor(world);

```

## Image Unwarping

Functional Guide | Reference: [SetupUnwarp](#), [Unwarp](#)

```

////////////////////////////////////
// This code snippet shows how to unwarp an image based //
// of the computed calibration coefficients.            //
////////////////////////////////////

// Images constructor
EImageBW8 srcImage;
EImageBW8 dstImage;

// EWorldShape constructor
EWorldShape worldShape;

// Lookup table constructor
EUnwarpingLut lut;

// ...

// Perform the calibration

```

```
worldShape.Calibrate(ECalibrationMode_Tilted | ECalibrationMode_Radial);  
  
// Setup the lookup table for unwarping  
worldShape.SetupUnwarp(&lut, &srcImage, true);  
  
// Perform the image unwarping  
worldShape.Unwarp(&lut, &srcImage, &dstImage, true);
```

## 6.4. EasyFind

### Pattern Learning

Functional Guide | Reference: [Learn](#)

```

////////////////////////////////////
// This code snippet shows how to learn a pattern //
// defined by a region of interest (ROI).        //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// ROI constructor
EROIBW8 pattern;

// EPatternFinder constructor
EPatternFinder finder;

// ...

// Attach the ROI to the source image
// and set its position
pattern.Attach(&srcImage);
pattern.SetPlacement(24, 52, 200, 200);

// Learn the pattern
finder.Learn(&pattern);

```

### Setting Search Parameters

Functional Guide | Reference: [SetMaxInstances](#), [SetAngleTolerance](#), [SetMinScore](#), [Save](#)

```

////////////////////////////////////
// This code snippet shows how to tune pattern finding //
// search parameters and save them into a file.        //
////////////////////////////////////

// Image constructor
EImageBW8 pattern;

// EPatternFinder constructor
EPatternFinder finder;

// ...

// Learn the pattern
finder.Learn(&pattern);

// Set the maximum number of occurrences
finder.SetMaxInstances(5);

// Set the rotation tolerances
finder.SetAngleTolerance(20.f);

// Set the minimum score
finder.SetMinScore(0.70f);

```

```
// Save the finding context into a model file
finder.Save("myModel.fnd");
```

## Pattern Finding and Retrieving Results

Functional Guide | Reference: [Load](#), [Find](#), [GetScore](#), [GetCenter](#)

```
////////////////////////////////////
// This code snippet shows how to perform pattern //
// finding operations and retrieve the results. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// EPatternFinder constructor
EPatternFinder finder;

// EFoundPattern constructor
std::vector<EFoundPattern> foundPattern;

// ...

// Load a model file
finder.Load("myModel.fnd");

// Perform the pattern finding
foundPattern= finder.Find(&srcImage);

// Retrieve the number of instances
int numInstances= (int)foundPattern.size();

// Retrieve the score and the
// position of the first instance
float score= foundPattern[0].GetScore();
float centerX= foundPattern[0].GetCenter().GetX();
float centerY= foundPattern[0].GetCenter().GetY();
```



## 6.5. EasyMatch

### Pattern Learning

Functional Guide | Reference: [LearnPattern](#)

```

////////////////////////////////////
// This code snippet shows how to learn a pattern //
// defined by a region of interest (ROI).          //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// ROI constructor
EROIBW8 pattern;

// EMatcher constructor
EMatcher matcher;

// ...

// Attach the ROI to the source image
// and set its position
pattern.Attach(&srcImage);
pattern.SetPlacement(24, 52, 200, 200);

// Learn the pattern
matcher.LearnPattern(&pattern);

```

### Setting Search Parameters

Functional Guide | Reference: [SetMaxPositions](#), [SetMinAngle](#), [SetMaxAngle](#), [SetMinScore](#), [SetInterpolate](#), [Save](#)

```

////////////////////////////////////
// This code snippet shows how to tune pattern matching //
// search parameters and save them into a file.          //
////////////////////////////////////

// Image constructor
EImageBW8 pattern;

// EMatcher constructor
EMatcher matcher;

// ...

// Learn the pattern
matcher.LearnPattern(&pattern);

// Set the maximum number of occurrences
matcher.SetMaxPositions(5);

// Set the rotation tolerances
matcher.SetMinAngle(-20.f);
matcher.SetMaxScale(20.f);

// Enable sub-pixel accuracy

```

```
matcher.SetInterpolate(true);

// Set the minimum score
matcher.SetMinScore(0.70f);

// Save the matching context into a model file
matcher.Save("myModel.mch");
```

## Pattern Matching and Retrieving Results

Functional Guide | Reference: [Load](#), [Match](#), [GetNumPositions](#), [GetPosition](#)

```
////////////////////////////////////
// This code snippet shows how to perform pattern //
// matching operations and retrieve the results. //
////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// EMatcher constructor
EMatcher matcher;

// ...

// Load a model file
matcher.Load("myModel.mch");

// Perform the matching
matcher.Match(&srcImage);

// Retrieve the number of occurrences
int numOccurrences= matcher.GetNumPositions();

// Retrieve the first occurrence
EMatchPosition myOccurrence= matcher.GetPosition(0);

// Retrieve its score and position
float score= myOccurrence.Score;
float centerX= myOccurrence.CenterX;
float centerY= myOccurrence.CenterY;
```