Open eVision

Text Identification Tools

Terms of Use

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

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

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

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

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

Contents

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.
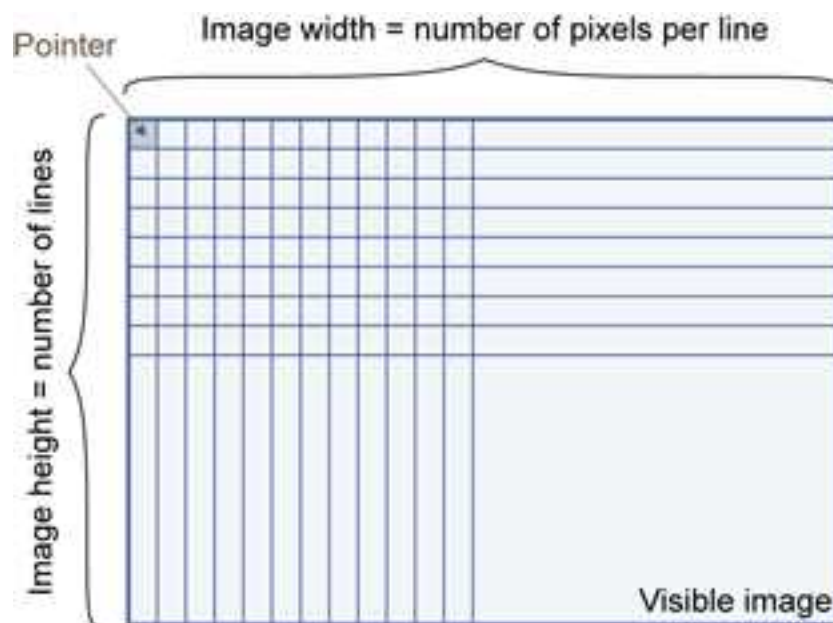
Image main parameters

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.
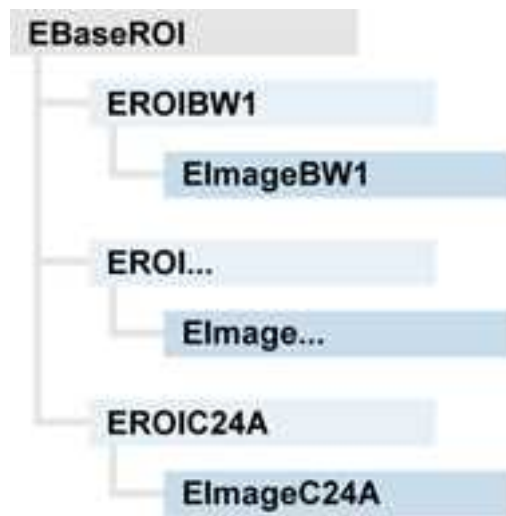
Depth maps

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

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

Point clouds

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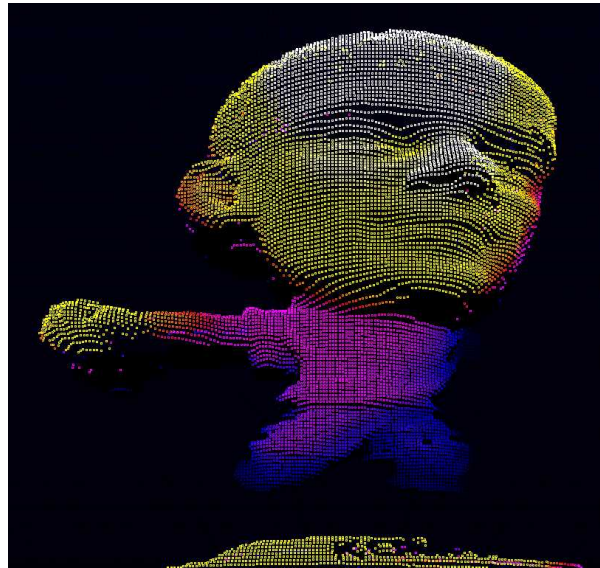
3D point clouds are produced by various 3D scanning techniques, such as Laser Triangulation, Time of Flight or Structured Lighting.

1.2. Pixel Container Types

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

C16	16-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB16 color images and MultiCam RGB16 format.	`EImageC16`
C24	C24 images store 24-bit color images (each pixel is stored in 3 bytes). Compatible with Microsoft® Windows RGB24 color images and MultiCam RGB24 format.	`EImageC24`
C24A	C24A images store 32-bit color images (each pixel is stored in 4 bytes). Compatible with Microsoft® Windows RGB32 color images and MultiCam RGB32 format.	`EImageC24A`

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)	`EDepthMap8`
EDepth16	16-bit depth map (each pixel is stored in 2 bytes as a fixed point)	`EDepthMap16`
EDepth32f	32-bit depth map (each pixel is stored in 4 bytes as a float)	`EDepthMap32f`

Point Clouds

Point Cloud	Set of points coordinates (stored as float)	`E3DPointCloud`

1.3. Supported Image File Types

Type	Description
BMP	Uncompressed image data format (Windows Bitmap Format)
JPEG	Lossy data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 10918-1. Compression irretrievably looses quality.
JFIF	JPEG File Interchange Format

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

1.4. Pixel and File Types Compatibility

Depth map to image conversion

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

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

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

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

Save throws an exception when:

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

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

(*) Default value.

Assigned image file type if argument is **ImageFileType_Auto** or missing

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

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

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

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

JPEG compression values

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

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

Representative JPEG 2000 compression quality values

JPEG 2000 compression	Description
-1	Default quality (*)
1	Highest image quality, lowest compression factor

JPEG 2000 compression	Description
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).

Point Clouds

- Use the `Save` method to save the point cloud in Open eVision proprietary file format.

- Use the `SavePCD` method to save the point cloud in a ASCII or a binary file compatible with other software such as PCL (Point Cloud Library).

> ✓ **TIP**
> The PCD format is supported in ASCII and binary modes.

2.2. Pixel Container File Load

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.

Point Clouds

- Use the `Load` method to save the point cloud in Open eVision proprietary file format.

- Use the `LoadPCD` method to save the point cloud in a ASCII or a binary file compatible with other software such as PCL (Point Cloud Library).

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

1. Construct an image object, for instance `EImageBW8`, either with width and height arguments, OR using the `SetSize` function.

2. Access a given pixel. There are several functions that do this. `GetImagePtr` returns a pointer to the first byte of the pixel at given coordinates.

External Memory Allocation

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

1. Declare an image object, for instance `EImageBW8`.

2. Create a suitably sized and aligned buffer (see Image Buffer).

3. Set the image size with the `SetSize` function.

4. Access the buffer with `GetImagePtr`. See also Retrieving Pixel Values.

2.4. Image and Depth Map Buffer

Image and depth map pixels are stored contiguously, from top row to bottom, from left to right, in Windows bitmap format (top-down DIB[1]) into an associated buffer.

The buffer address is a pointer to the start address of the buffer, which contains the top left pixel of the image.

---

[1]device-independent bitmap
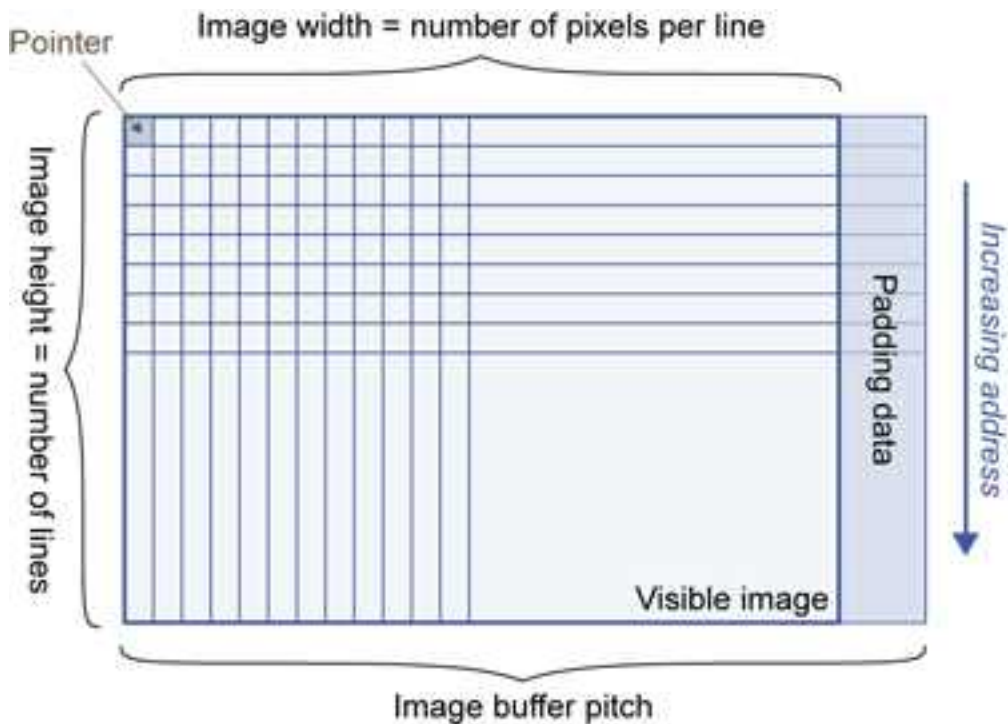
Image Buffer pitch

- Alignment must be a multiple of 4 bytes.

- Open eVision 1.2 onwards default pitch is 32 bytes for performance reasons (Open eVision 1.1.5 was 8 bytes).

Memory Layout

- `EImageBW1` stores 8 pixels in one byte.

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

- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

Example memory layout of the first pixels of a BW8 image buffer:
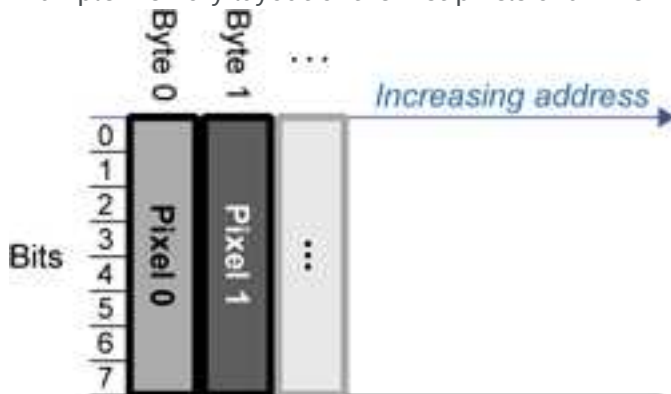
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).

Example memory layout of the first pixels of a BW16 image buffer:
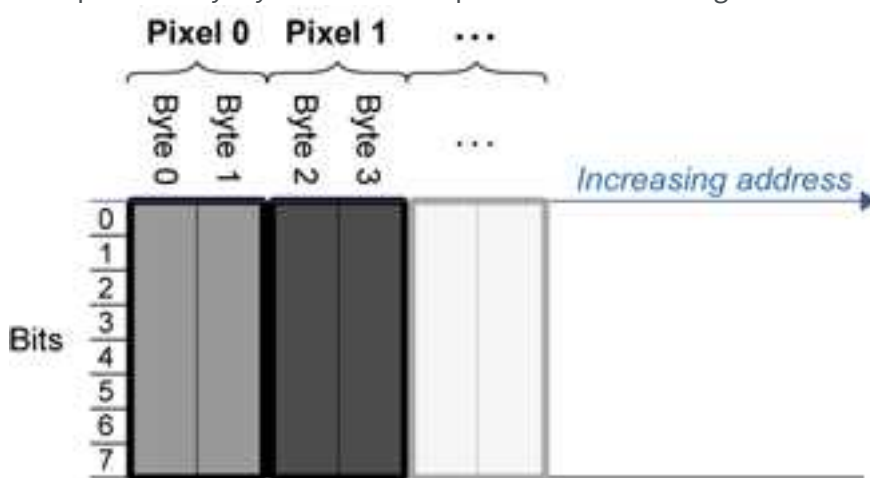
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

Example memory layout of the first pixels of a C15 image buffer:
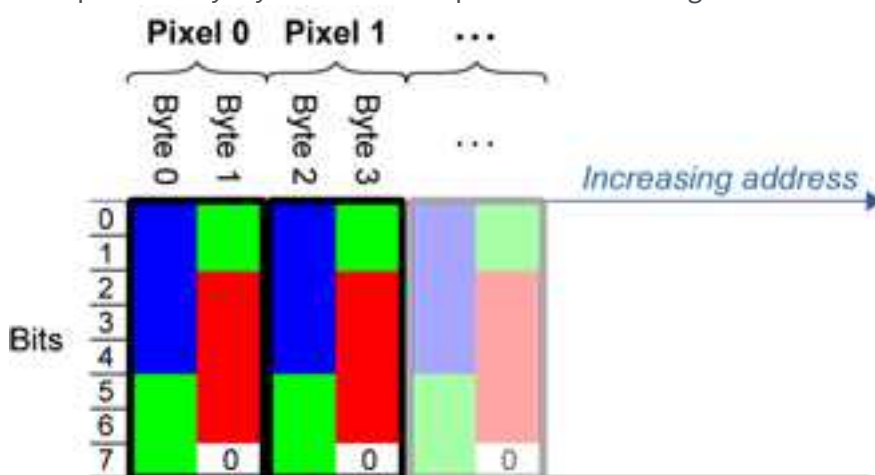
- `EImageC16` stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
The second color component is coded with 6-bits.

Example memory layout of the first pixels of a C16 image buffer:
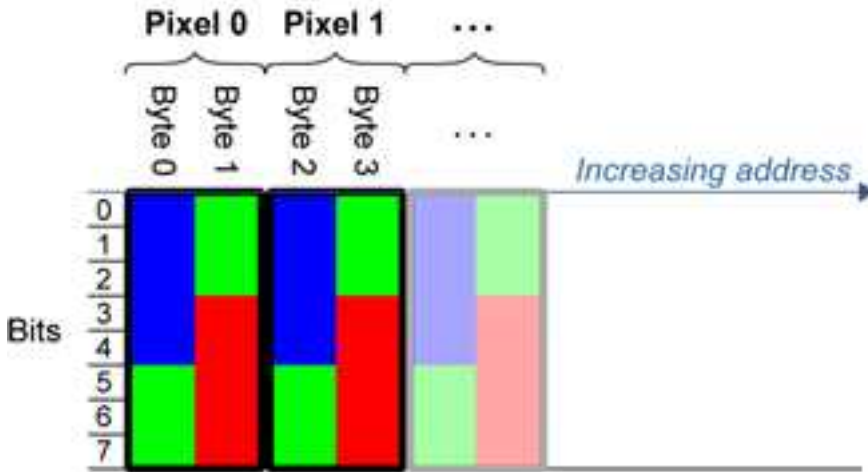
- `EDepthMap16` store each pixel in 2 bytes using a fixed point format.

- `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:
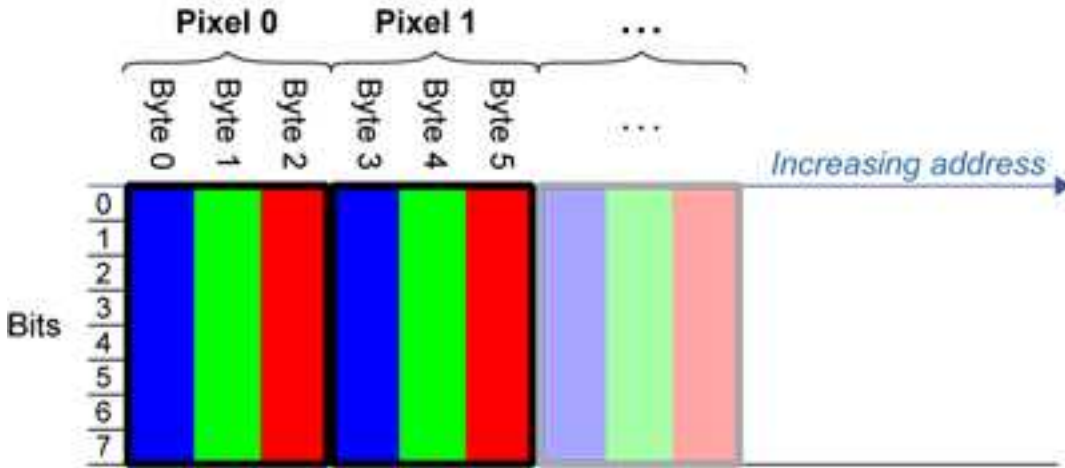
- `EImageC24A` stores each pixel in 4 bytes. Each color component is coded with 8-bits. The alpha channel is also coded with 8-bits.

Example memory layout of the first pixels of a C24A image buffer:
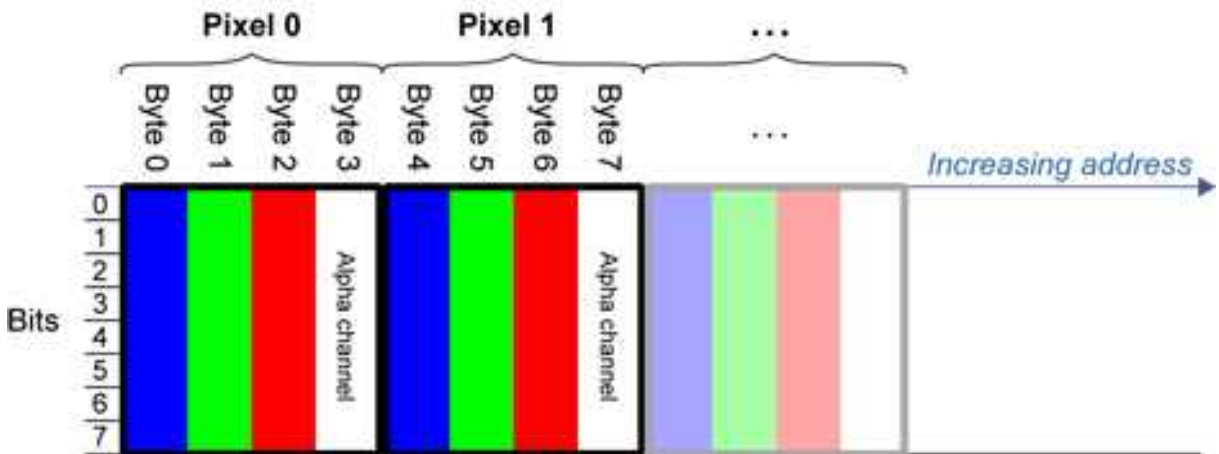
- `EDepthMap32f` store each pixel in 4 bytes using a float format.

2.5. Image Drawing and Overlay

- Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.

- The color palette in 256-color display mode gives optimal rendering. Gray-level images can be improved using LUT[3]s (using histogram stretching techniques or pseudo-coloring).

- The zoom can be different horizontally and vertically.

- `DrawFrameWithCurrentPen` method draws a frame.

- **Non-destructive overlaying** drawing operations do not alter the image contents, such as `MoveTo/LineTo`.

- **Destructive overlaying** drawing operations alter the image contents by drawing inside the image such as `Easy.OpenImageGraphicContext`. Gray-level [color] images can only receive a gray-level [color] overlay.

2.6. 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.
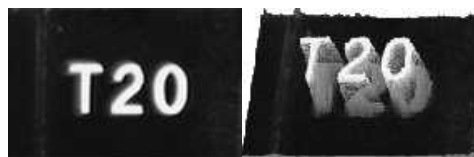
3D rendering

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

---

[1]Graphics Device Interface
[2]Microsoft Foundation Class
[3]LookUp Tables

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.
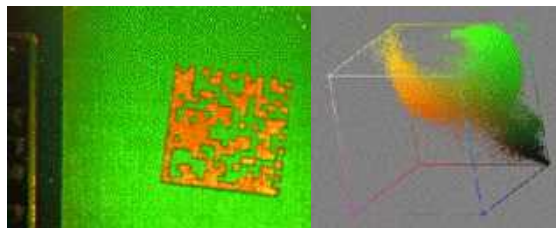
Color histogram rendering

2.7. 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 | RGB values array (`EC24Vector`) |
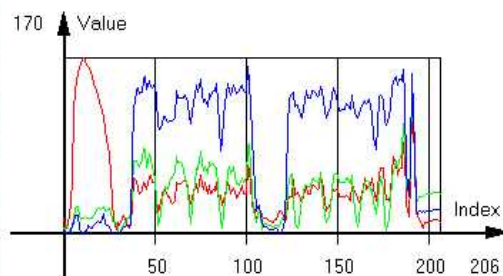
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

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

Vector types

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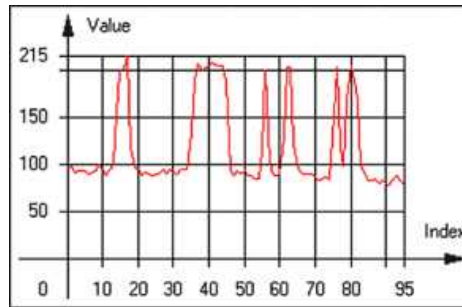
Graphical representation of an `EBW8Vector` **(see** `Draw` **method)**

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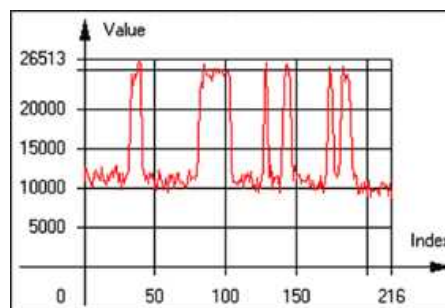
Graphical representation of an `EBW16Vector`

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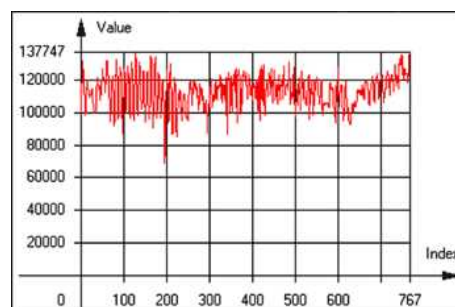
Graphical representation of an `EBW32Vector`

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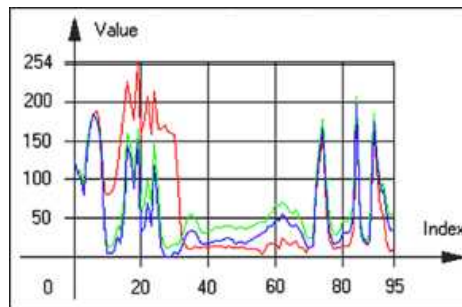
Graphical representation of an `EC24Vector`

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

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

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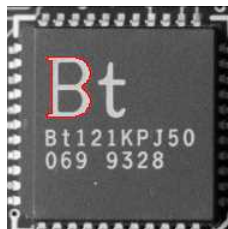
Graphical representation of an `EBW16PathVector` **(see** `Draw` **method)**

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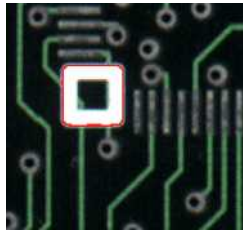
Graphical representation of an `EC24PathVector` **(see** `Draw` **method)**

- `EBWHistogramVector`: a sequence of frequency counts of pixels in a BW8 or BW16 image
  (used by `EasyImage.IsodataThreshold`, `EasyImage.Histogram`,
  `EasyImage.AnalyseHistogram`, `EasyImage.SetupEqualize`, ...).
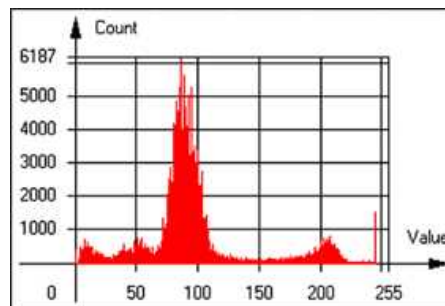
Graphical representation of an `EBWHistogramVector` **(see** `Draw` **method)**

- `EPathVector`: a sequence of pixel coordinates. The corresponding pixels need not be
  contiguous
  (used by `EasyImage.PathToImage` and `EasyImage.Contour`).
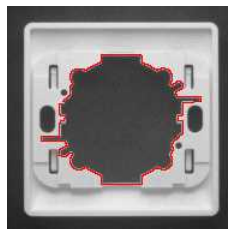
Graphical representation of an `EPathVector` **(see** `Draw` **method)**

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

2. **Fill a vector with values**. First empty it, using the `EVector.Empty` member, then add
   elements one at a time by calling the `EC24Vector.AddElement` member. You can access any
   element by means of indexing.

3. **Access a vector element**, either for reading or writing. Use the brackets operator, for instance, `EC24Vector.operator[]`.

4. **Determine the current number of elements**, use member `EVector.NumElements`.

5. **Draw the vector**.
   A pixel vector is a plot of the element values as a function of the element index, so its graphical appearance depends on its type. You can draw a vector in a window. For legibility, the drawing should appear on a neutral background.
   Drawing is done in the device context associated to the desired window. By default, curves are drawn in blue, annotations are drawn in black. The following parameters can be defined: graphicContext, width, height, origin, origin, color0, color1, color2.
   The `EC24Vector` has three curves drawn instead of one, each corresponding to a color component. By default, red, blue and green pens are used.

2.8. ROI Main Properties

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

Save and load

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

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

ROI Classes

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

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

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

Attachment

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

Nesting

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

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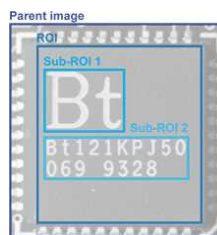
Nested ROIs: Two sub-ROIs attached to an ROI, itself attached to the parent image

Cropping

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

> **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).
    In VB6, MouseDown, MouseMove,MouseUp events return the current cursor position in twips rather than pixels, so conversion is mandatory.

2.9. 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
EasyImage	EasyImage::Threshold
	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
EasyObject	EImageEncoder::Encode
EasyFind	EPatternFinder::Find

> ✓ **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.
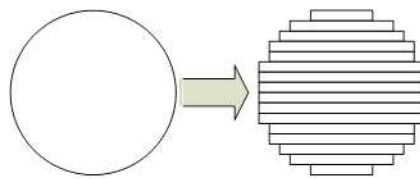
Run-Length Encoding of a circle-shaped region

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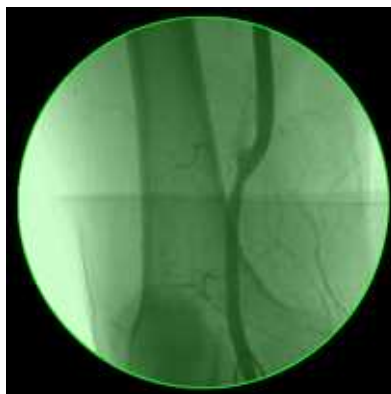
Circle region encompassing the useful part of an X-Ray image

- `EEllipseRegion`

  □ The contour of an `EEllipseRegion` class is an ellipse.

  □ Define it using its center, long and short radius and angle.
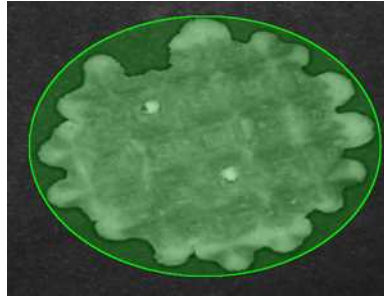
Ellipse region encompassing a waffle

- `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.
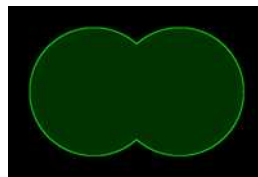
Union of 2 circles

● Intersection

☐ The `ERegion::Intersection(const ERegion&, const ERegion&)` method returns the region that is the intersection of the two regions passed as argument.
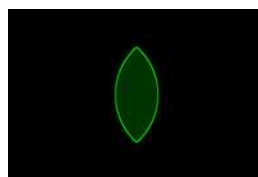
Intersection of 2 circles

- Subtraction

  - The `ERegion::Substraction(const ERegion&, const ERegion&)` method returns the first region passed as argument after removing the second one.
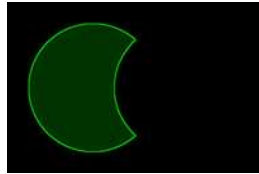
Subtraction of 2 circles

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 that your first call to a method takes 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 ways 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.

2.10. Flexible Masks

ROIs vs flexible masks

ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 24 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.
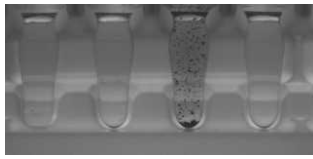
Source image	Associated mask	Processed masked image
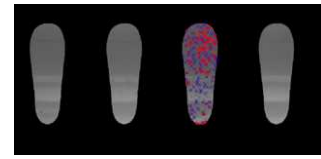
A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.

Flexible Masks in EasyImage
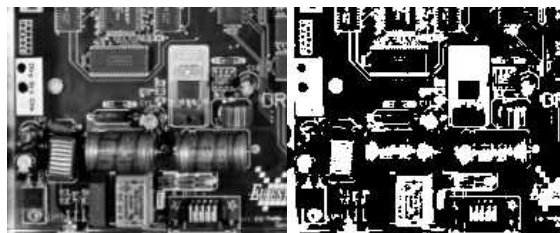
Source image (left) and mask variable (right)

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.**
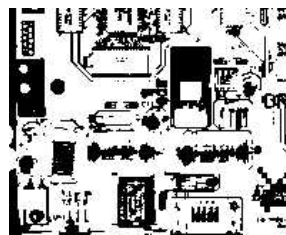
Resulting image

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
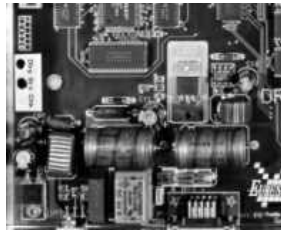
Source image

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`.
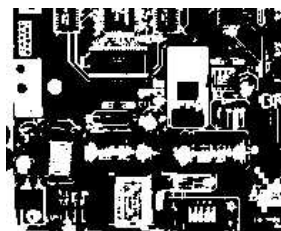
BW8 resulting image that can be used as a flexible mask

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.
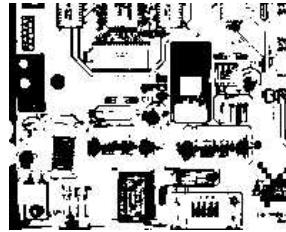
BW8 resulting image that can be used as a flexible mask

3) EObjectSelection.RenderMask: from a selection of blobs

`EObjectSelection.RenderMask` can, for example, discard small objects resulting from noise.
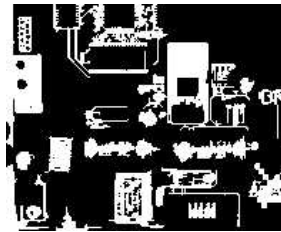
BW8 resulting image that can be used as a flexible mask

Example: Restrict the areas encoded by EasyObject
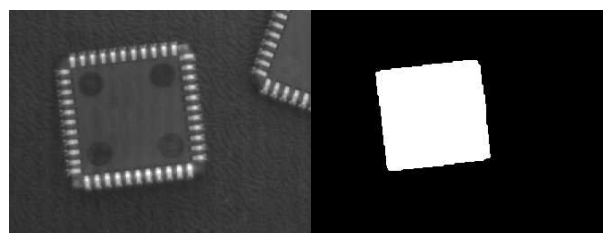
Find four circles (left) Flexible mask can isolate the central chip (right)

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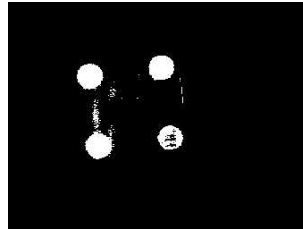
5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

2.11. Profile

Profile Sampling

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

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

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

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

Profile Analysis

The profile can be processed to find peaks or transitions:

- A transition corresponds to an object edge (black to white or white to black). It can be detected by taking the first **derivative** of the signal (which transforms transitions (edges) into peaks) and looking for peaks in it.
  `EasyImage.ProfileDerivative` computes the first derivative of a profile extracted from a gray-level image.
  The `EBW8` data type only handles unsigned values, so the derivative is shifted up by 128. Values under [above] 128 correspond to negative [positive] derivative (decreasing [increasing] slope).

- A peak is the portion of the signal that is above [or below] a given threshold - the maximum or minimum of the signal. This may correspond to the crossing of a white or black line or thin feature. It is defined by its:
  - Amplitude: difference between the threshold value and the max [or min] signal value.
  - Area: surface between the signal curve and the horizontal line at the given threshold.

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

Profile Insertion Into an Image

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

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

3. Text Identification Tools

3.1. EasyOCR - Reading Texts

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

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

There are 4 steps to recognizing characters:

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

Workflow
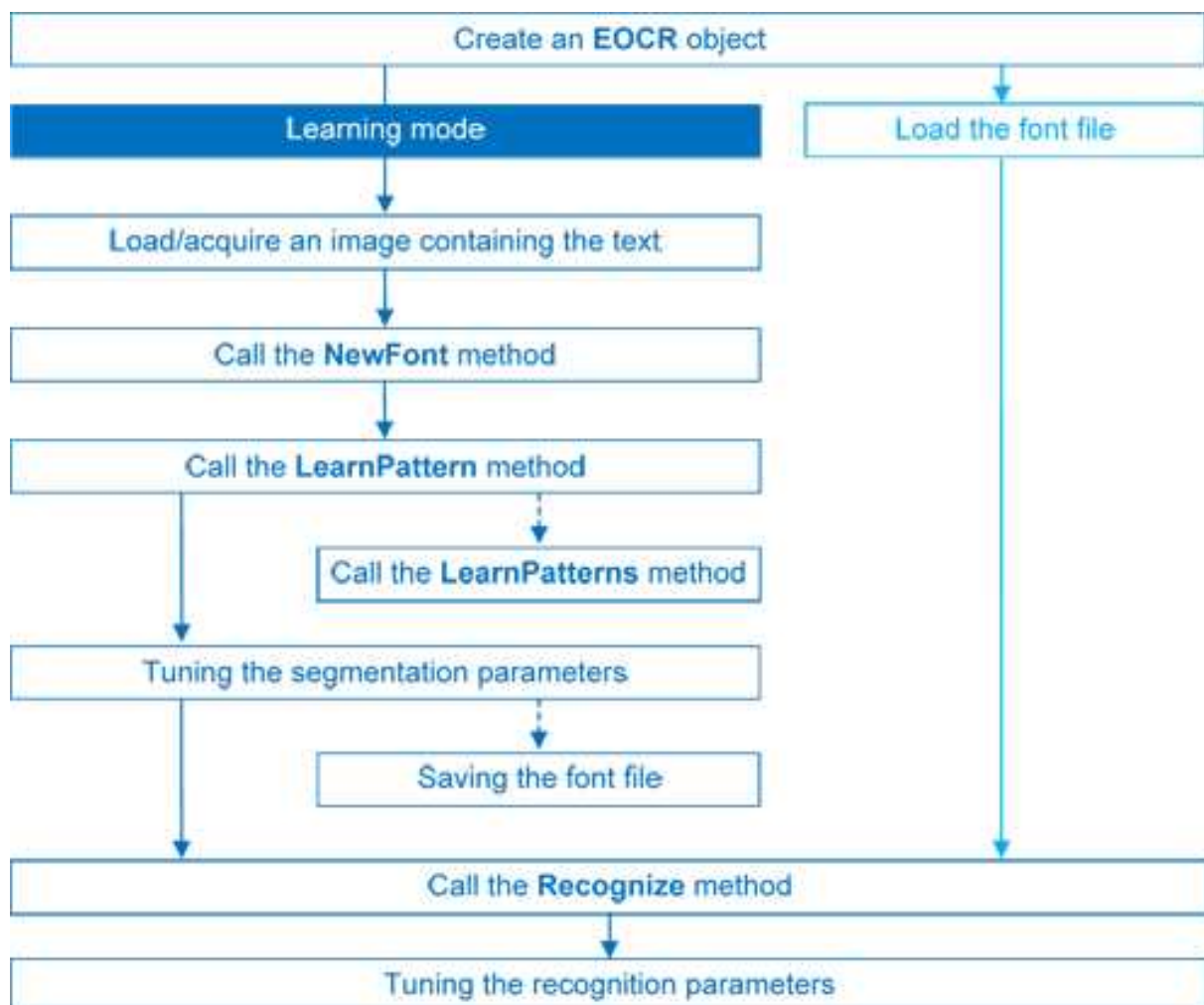
Learning Process

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

Font files are created as follows:

1. `NewFont` clears the current font.

2. `LearnPattern` or `LearnPatterns` adds the patterns from the source image to the font. Patterns are ordered by their index value, as assigned by the `FindAllChars` process. The patterns in a font are stored as a small array of pixels, by default 5 pixels wide and 9 pixels high. This size can be changed before learning, using parameters `PatternWidth`and `PatternHeight`.

3. `RemovePattern` removes unwanted patterns (optional).

4. `Save` writes the contents of the font to a disk file with parameter values: `NoiseArea`, `MaxCharWidth`, `MaxCharHeight`, `MinCharWidth`, `MinCharHeight`, `CharSpacing`, `TextColor`.

Segmenting

For learning as well as recognition, EasyOCR segments the characters, i.e. locates the characters and determines their bounding box. This is done by means of blob analysis (thresholding followed by a grouping of pixels of the same color, as is done by EasyObject). After blobs have been found, they can be filtered to remove unwanted features (small blobs of noise, large extraneous objects, ...).

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

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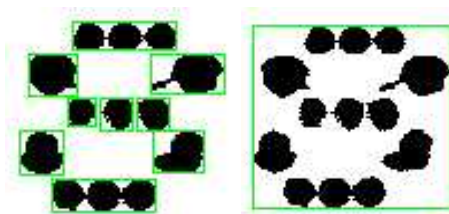
Character segmentation by blob grouping

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

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

Segmentation parameters

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

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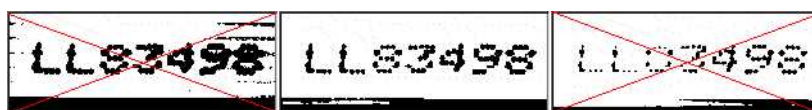
Too high threshold value (left), Threshold adjustment (middle), Too low threshold value (right)

- `NoiseArea`: Blob areas smaller than this value are discarded. Make sure small character features are preserved (i.e., the dot over an "i" letter).

- `MaxCharWidth`, `MaxCharHeight`: Maximum character size. If a blob does not fit in a rectangle with these dimensions, it is discarded or split into several parts using vertical cutting lines. If several blobs fit in a rectangle with these dimensions, they are grouped together.

- `MinCharWidth`, `MinCharHeight`: Minimum character size. If a blob or a group of blobs fits in a rectangle with these dimensions, it is discarded.

- `CharSpacing`: The width of the smallest gap between adjacent letters. If it is larger than `MaxCharWidth` it has no effect.
  If the gap between two characters is wider than this, they are treated as different characters. This stops thin characters being incorrectly grouped together.

- `RemoveBorder`: Blobs near image/ROI edges cannot normally be exploited for character recognition. By default, they are discarded.

Recognition

The characters are compared to a set of patterns, called a **font**. A character is recognized by finding the best match between a character and a pattern in the font. After the character has been located, it is normalized in size (stretched to fit in a predefined rectangle) for matching. The normalized character is compared to each normalized template in the font database and the best matches are returned.

1. `Load`: reads a pre-recorded font from a disk file.

2. `BuildObjects`: The image is segmented into **objects** or blobs (connected components) which help find the **characters**. This step can be bypassed if the exact position of the characters is known. If the character isolation process is bypassed, you must specify the known locations of the characters: `AddChar` and `EmptyChars`.

3. `FindAllChars`: selects the objects considered as characters and sorts them from top to bottom then left to right.

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

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

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

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

Recognition parameters

- `MaxCharWidth`, `MaxCharHeight`: if a blob does not fit within a rectangle with these dimensions, it is not considered as a possible character (too large) and is discarded. Furthermore, if several blobs fit in a rectangle with these dimensions, they are grouped together, forming a single character. The outer rectangle size should be chosen such that it can contain the largest character from the font, enlarged by a small safety margin.

- `MinCharWidth`, `MinCharHeight`: if a blob or a group of blobs does fit in a rectangle with these dimensions, it is not considered as a possible character (too small) and is discarded. The inner rectangle size should be chosen such that it is contained in the smallest character from the font, shrunk by a small safety margin.

- `RemoveNarrowOrFlat`: Small characters are discarded if they are narrow **or** flat. By default they are discarded when they are both narrow **and** flat.

- `CharSpacing`: if two blobs are separated by a vertical gap wider than this value, they are considered to belong to different characters. This feature is useful to avoid the grouping of thin characters that would fit in the outer rectangle. Its value should be set to the width of the smallest gap between adjacent letters. If it is set to a large value (larger than `MaxCharWidth`), it has no effect.

- **CutLargeChars**: when a blob or grouping of blobs is larger than `MaxCharWidth`, it is discarded. When enabled, the blob is split into as many parts as necessary to fit and the amount of white space to be inserted between the split blobs is set by `RelativeSpacing`. This is an attempt to separate touching characters.

- **RelativeSpacing**: when the `CutLargeChars` mode is enabled, setting this value allows specifying the amount of white space that should be inserted between the split parts of the blobs.

Too large character · Too small character · Too close characters
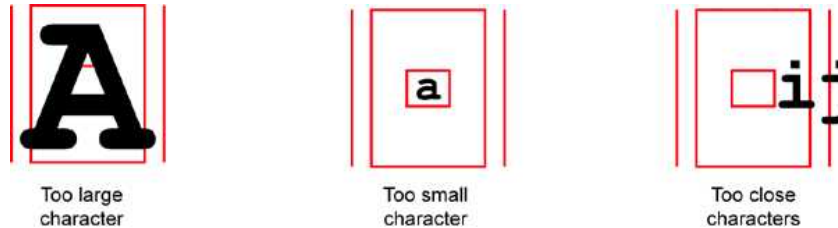
Invalid recognition settings

Advanced tuning

These recognition parameters can be tuned to optimize recognition:

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

Filtering the characters (in the `ReadText` method), can be used if the marking structure is fixed.

When objects are larger than the `MaxCharWidth` property, they can be split into as many parts as needed, using vertical cutting lines.

ESegmentationMode, character isolation mode defines how characters are isolated:

- **Keep objects** mode: a character is a blob; no attempt is made to group blobs, thus damaged characters cannot be handled and small features such as accents and dots may be discarded by the minimum character size criterion.

- **Repaste objects** mode: blobs are grouped to form distinct **characters** if they fit in the maximum character size and are not separated by a vertical gap, thus preserving accents and dots.

3.2. EasyOCR2 - Reading Texts (Improved)

Reference | Code Snippets

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

It uses an innovative segmentation method to detect blobs in the image, and then places textboxes over the detected blobs following a user-defined topology (number of lines, words and characters in the text). These methods support text rotation up to 360 degrees, can handle non-uniform illumination, textured backgrounds, as well as dot-printed or fragmented characters.

A character type (letter / digit / symbol) can be specified for each character in the text, improving recognition rate and speed. The character database that is used for recognition can be learned from sample images or read from a TrueType font (.ttf) file.

Text recognition with **EasyOCR2** follows four phases:

Input image (left) and image segmentation (right)
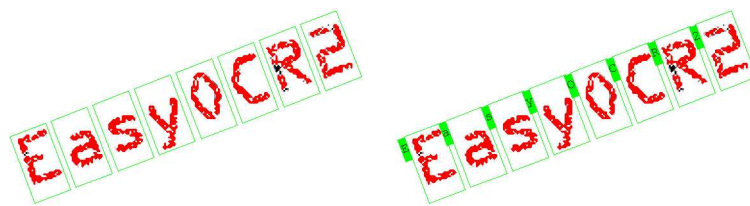
Fitting textboxes (left) and recognition (right)

EasyOCR2 vs EasyOCR

EasyOCR2 will give better results than **EasyOCR** when dealing with:

- ☐ Unknown text rotation

- ☐ Dotted or fragmented characters

- ☐ Non-uniform illumination or textured backgrounds

- When TrueType font files are available that match the text to be read, **EasyOCR2** allows the user to use those font files directly for recognition, while **EasyOCR** does not.

- When none of the above are relevant to the application, the user may prefer to use **EasyOCR** to **EasyOCR2** due to its superior computational speed.

Workflow
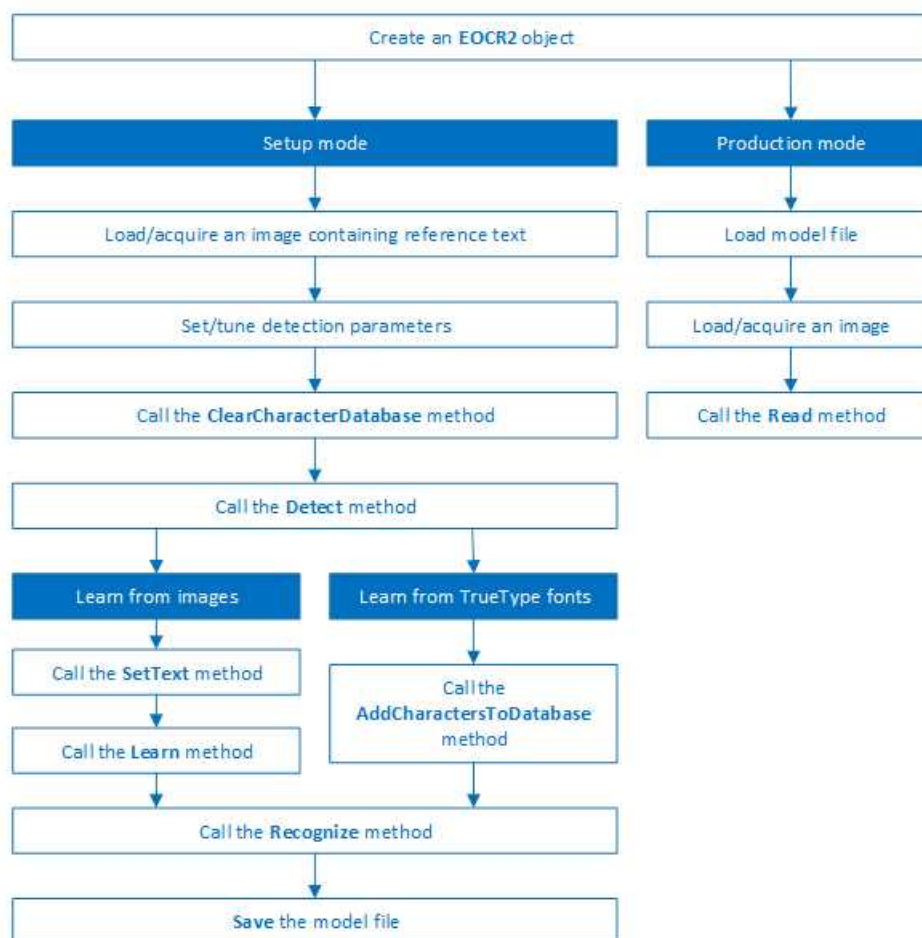
Detection

EasyOCR2 finds characters in an image as follows:

1. **EasyOCR2** segments the image, finding blobs that represent (parts of) the characters.

2. Blobs that are too large or too small to be considered part of a character are filtered out.

3. **EasyOCR2** fits character boxes to the detected blobs according to a given `topology` and `detectionMethod`.

   The topology describes the structure of the text in the image, defining the number of lines, the number of words per line and the number of characters per word.

4. **EasyOCR2** extracts the pixels inside each character box from the image.

   The resulting character-images can be used to learn or recognize the characters.

A workflow detecting text in an image could be as follows:

a. Set the required detection parameters.

b. Alternatively, call `Load` to read a pre-made model (.o2m) file containing detection parameters from disk.

45

   c. Call `Detect` to extract the text from the image.

The method `Detect` will return an `EOCR2Text` structure that contains a textbox and a bitmap image for each character, hierarchically stored in `EOCR2Line` -> `EOCR2Word` -> `EOCR2Char` structures.

See example in code snippet: "Detecting Characters" on page 96

An example of a fixed-width font, processed with the `detectionMethod` **'`EOCR2DetectionMethod_FixedWidth`'**

An example of a proportional font, processed with the `detectionMethod` **'`EOCR2DetectionMethod_Proportional`'**
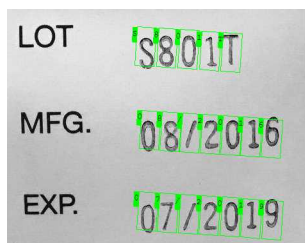
The text angle estimate for this image is slightly off when `NumDetectionPasses`=1
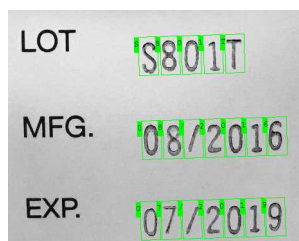
The text angle estimate is better when `NumDetectionPasses`=2
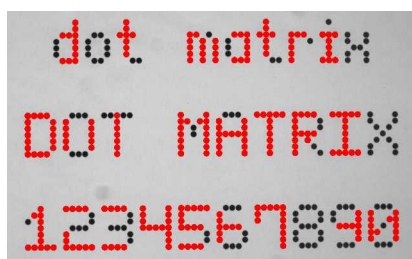
For this dotted text, setting `CharsMaxFragmentation` to 0.1 leads to incomplete segmentation results
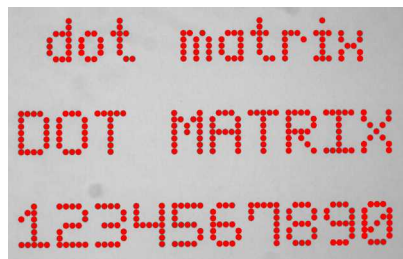
Setting `CharsMaxFragmentation` to 0.01 gives better segmentation results

Detection parameters

Required parameters

☐ The parameter `Topology` tells the box-fitting method how to structure the textboxes it fits to the detected blobs. Using a modified version of Regex expressions, the topology determines the number of lines in the text, the number of words per line and the number of characters per word. The section **Recognition Parameters** contains an extensive explanation of the syntax for the Topology.

☐ The parameter `CharsWidthRange` tells the segmentation and detection methods how wide the characters in the image can be.

☐ The parameter `CharsHeight` tells the segmentation and detection methods how high the characters in the image can be.

☐ The parameter `TextPolarity` tells the segmentation method whether it should look for light characters on a dark background or vice versa.

Advanced parameters for segmentation (optional):

☐ The `CharsMaxFragmentation` parameter tells the segmentation algorithm how small blobs can be to be considered (part of) a character. The minimum allowed area of a blob is given by:

$$minArea = CharsMaxFragmentation * CharsHeight * min(CharsWidthRange)$$

This parameter should be set between 0 and 1, the default setting is 0.1.

☐ The `MaxVariation` parameter determines how stable a blob in the image should be in order to be considered a potential character.
A region with clearly defined edges is generally considered stable while a blurry region is not. A high setting allows detection of blobs that are more unstable, a low setting allows only very stable blobs.
This parameter should be set between 0 and 1, the default setting is 0.25.

□ The `DetectionDelta` parameter determines the range of grayscale values used to determine the stability of a blob.
A low setting will make the algorithm more sensitive to noise; a high setting will make the algorithm insensitive to blobs with low contrast to the background.
This parameter should be set between 1 and 127, the default setting is 12.

Advanced parameters for detection (optional)

□ The parameter `DetectionMethod` selects the algorithm used for fitting. The setting `EOCR2DetectionMethod_FixedWidth` (default) is optimized for texts with fixed width fonts (including dotted text), the setting `EOCR2DetectionMethod_Proportional` is optimized for texts with proportional fonts.

□ The `TextAngleRange` parameter tells the box-fitting method how the text in the image is oriented. It will test the following range of rotation angles:

$$\min(\text{TextAngleRange}) \leq angle \leq \max(\text{TextAngleRange})$$

where angles are defined with respect to the horizontal. The unit for the angles (degrees/radians/revolutions/grades) can be set using `easy::SetAngleUnit()`.

The default setting for this parameter is [-20, 20] degrees.

□ The parameter `NumDetectionPasses` determines how many passes are made to fit textboxes to the detected blobs. The initial pass will fit textboxes to all detected blobs. Subsequent passes will select only those blobs that are covered by the textboxes from the previous pass and fit textboxes to that subset of blobs, potentially resulting in a more optimal fit.
This parameter should be set to either 1 or 2, the default setting is 1.

Advanced parameters, specific for the setting *EOCR2DetectionMethod_FixedWidth*

□ The `RelativeSpacesWidthRange` parameter tells the box-fitting method how wide the spaces between words may be. It will test the following range of spaces:

$$\min(\text{SpacesWidthRange}) * charWidth \leq space \leq \max(\text{SpacesWidthRange}) * charWidth$$

□ The parameter `CharsWidthBias` biases the optimization toward wider of narrower character boxes.

□ The parameter `CharsSpacingBias` biases the optimization toward smaller or larger spacing between characters boxes.

Additional remarks

□ When the setting `EOCR2DetectionMethod_FixedWidth` is selected, all character boxes will have the same width and they do not necessarily have to fit tightly around the characters.

□ When the setting `EOCR2DetectionMethod_Proportional` is selected, the character boxes will fit tightly around the characters, if any character falls outside the range of allowed character widths, the detection will fail.

Learning

In order to recognize characters, **EasyOCR2** requires a database of known reference characters. We may generate this character database from images and/or from TrueType system fonts.

A workflow to build a character database could be as follows:

a. Set the required detection parameters or call `Load` to read the model (.o2m) file from disk.

b. Optionally, call `ClearCharacterDatabase` to clear the current character database.

c. Call `Detect` to extract the text from the image.

d. Call `SetText` in the extracted text structure to set the correct value for each character.

e. Call `Learn` to add the detected characters and their correct value to the current character database.

f. Call `SaveCharacterDatabase` to save the current character database to disk.

g. Alternatively, call `Save` to save the model file to disk, including the detection parameters and the created character database.

See example in code snippet: "Learning Characters" on page 97

Recognition

EasyOCR2 recognizes characters using a classifier that is trained on the character database. For each input character, the classifier will calculate a score for all candidate outputs, the candidate with the highest score will be returned as the recognition result. Through the `Topology` parameter, prior information about each character can be passed to the classifier, reducing the number of candidates and improving the recognition rate.

The production workflow for recognizing text from images could be as follows:

☐ Call `Load` to read the model (.o2m) file from disk. The model file contains all detection parameters, as well as the topology and the reference character database.

☐ Load or acquire the image.

☐ Call `Read` to detect and recognize the characters.

☐ Alternatively, call `Detect` to extract the text from the image, followed by `Recognize` to recognize the extracted text. This allows the user to modify elements of the detected text before recognition if so desired.

The methods `Read` and `Recognize` will return a string with the recognition results. To access more in-depth information about the results, one may call `ReadText`. This returns an `EOCR2Text` structure that contains the coordinates and sizes of each textbox as well as a bitmap image and a list of recognition scores for each character.

See example in code snippet: "Reading Characters" on page 98

Recognition parameters

The `Topology` parameter specifies the structure of the text (number of lines/words/characters) as well as the type of characters in the text. The recognition method will limit the number of candidates for each character based on the given topology.

It uses modified regular expression wildcards:

- "." (dot) represents any character (not including a space).
- "L" represents an alphabetic character.
  - "Lu" represents an uppercase alphabetic character.
  - "Ll" represents a lowercase alphabetic character.
- "N" represents a digit.
- "P" represents the punctuation characters: ! " # % & ' ( ) * , - . / : ; < > ? @ [\ ] _ { | } ~
- "S" represents the symbols: $ + - < = > | ~
- "\n" represents a line break.
- " " (space) represents a space between two words.

Combinations can be made, for example: [LN] represents an alpha-numeric character. To specify multiple characters, simply add {n} at the end for n characters. If the amount of characters is uncertain, specify {n,m} for a minimum of n characters and a maximum of m characters.

The topology "[LuN]{3,5}PN{4} \n .{5} LL" represents a text comprised of 2 lines:

- The first line has 1 word composed of 3 to 5 uppercase alpha-numeric characters, followed by a punctuation character and 4 digits.
- The second line has 2 words. The first word comprises of 5 wildcard characters, the second word has 2 letters (upper- or lowercase).

The topology "L{3}P N{6} \n L{3}P NNPN{4}" represents a text with 2 lines:

- The first line has 2 words. The first word has 3 uppercase letters followed by a punctuation mark, the second word has 6 digits.
- The second line also has two words. The first word has 3 uppercase letters followed by a punctuation mark. The second word has 2 digits, followed by a punctuation mark and 4 additional digits.

The topology ".{10} \n .{7} \n .{5} .{5} \n .{5} .{7}" represents a text with 4 lines:
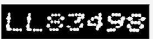
- The first line contains a single word of 10 (ASCII) characters
- The second line contains a single word of 7 characters
- The third line contains two words, each of 5 characters.
- The fourth line contains two words of 5 and 7 characters respectively.

3.3. EasyOCV - Validating Texts

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

Workflow

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

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

1. Threshold the image to separate the foreground (marking) from the background.

2. Free blobs: Blob analysis is performed and the detected objects may be selected depending on their size to generate a set of candidates. (Blobs can be selected or unselected manually, to add features of unusual size or delete spurious objects.)

3. Free characters: The blobs are aggregated into boxes corresponding to the characters. (Manual correction is possible.)

4. Create single text: The characters are aggregated into boxes corresponding to the texts. (Manual correction is possible.)

5. Create final model: The system will analyze the shape of the template and generate the required data structures that represent it and allow fast inspection. It will also perform some quality measurement on the template for later comparison with the sample. At this stage, only texts and their constituent characters are stored. The free characters and free objects are discarded.

6. Before the template can be saved, the inspection parameters must be defined . These include:

   - the allowed ranges for the location parameters of the texts with respect to their nominal position in the inspected ROI. These parameters correspond to translation and rotation (scaling in both horizontal and vertical directions, and shearing).

   - the allowed ranges of character location with respect to their nominal position in the texts (during translation).

   - the allowed ranges of the quality ratings with respect to their nominal values. These parameters can be chosen among the area of the character background and foreground, the accumulated gray level of these areas, and a similarity coefficient.

Learning Process

Model structure

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

Texts are sets of characters	Characters are sets of blobs	Blobs are "atomic" constituents of markings
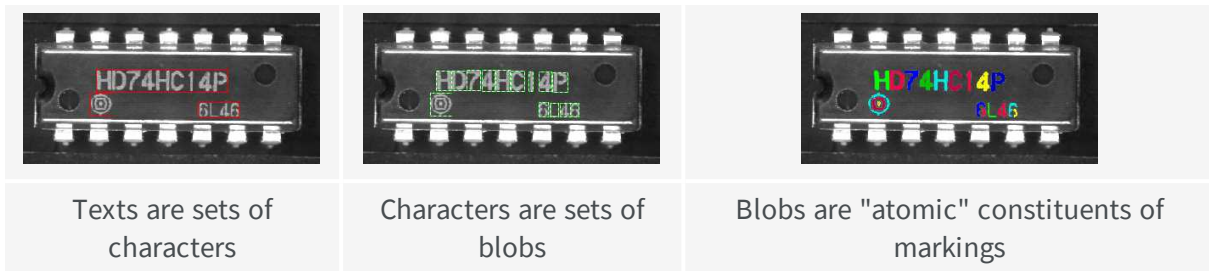
In a simple application, a ready-made template (Open eVision Studio provides a comprehensive template editor) is used in the inspection phase.

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

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

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

template design considerations

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

- **What movement may occur** ? Placement repeatability must be evaluated. If the part travels along a guide, sometimes only translation occurs which can be handled by a single alignment pattern (fiducial). If rotation or scaling can occur, two alignment patterns are preferable.
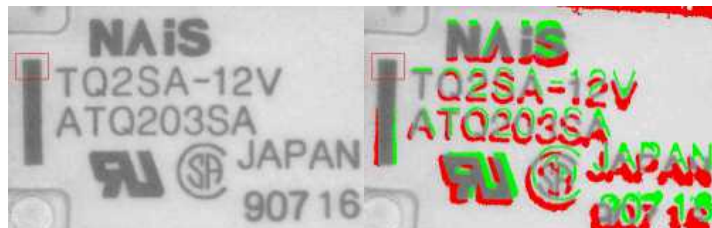
Bad: rotation not handled

- **Are the alignment patterns fixed, well-contrasted features, not subject to degradation, that move rigidly with the inspected part** ?
When two are used, they should be located as far apart as possible for optimal accuracy. The pattern ROIs should not contain extraneous features likely to change from sample to sample. The patterns should be small so that rotation and scaling has little impact, but large enough to contain information at different scales.
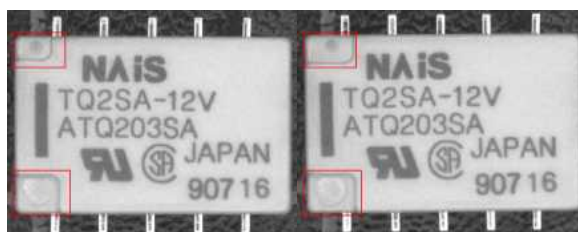
Bad: the location pattern is not repeatable

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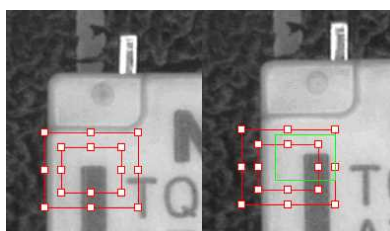
Bad: too tight search areas

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

Bad: undue inspection of the background

Access the template components

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

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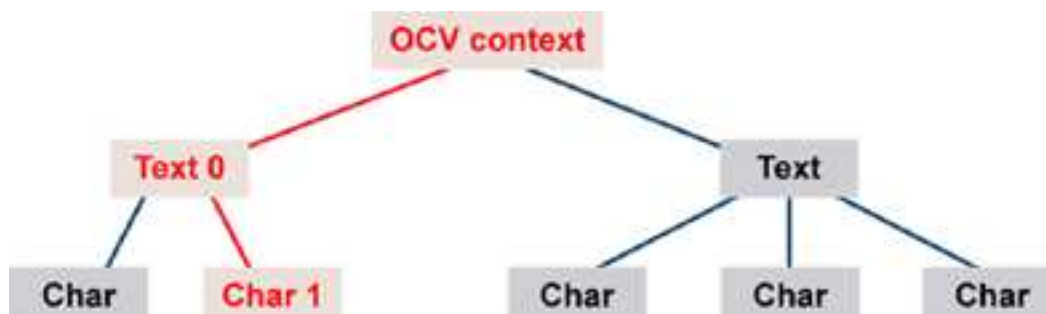
Access by index (second character of first text)

You can access and modify several components collectively. Every text and every character has a boolean **selected** property, so that in a single operation, you can read/write the value of a property for all currently selected elements. When reading a property, a value is only returned if it is identical for all selected elements.
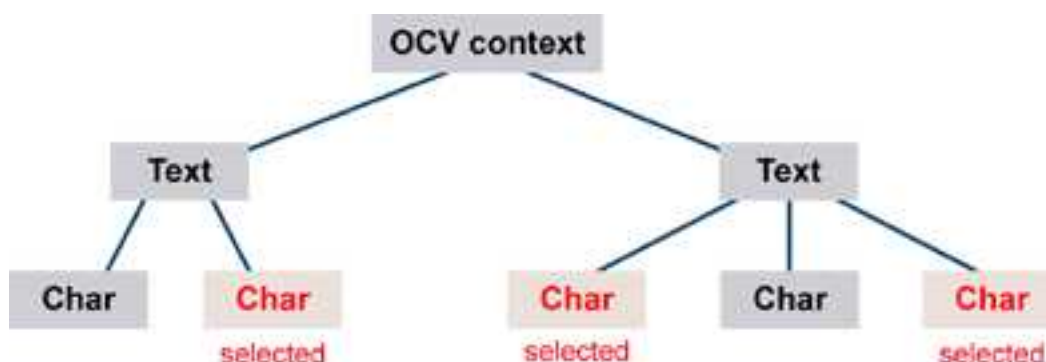
Access a group of selected components

Scattering operations over selected components is very handy for interactive editors that list and modify parameters for single items or groups of items.

Learning Passes

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

- To compute an **average** ideal, noise-free, image that reveals the central tendency of the part image.
  For each image, realign and normalize (`Register`).
  If the operation is successful (good pattern location), call
  `Learn`(ELearningMode_Average) for immediate processing (on-the-fly learning), or

`AddPathName` for deferred processing (batch learning).

- To measure **devations** around the average image.
  For each image, realign and normalize (`Register`).
  If the operation is successful, call
  `Learn(ELearningMode_AbsDeviation)` for immediate processing and recommended method,
  or
  `Learn(ELearningMode_RmsDeviation)` for enhancing large deviations.

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

Inspect and compare image with model

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

- **Locate**: A region of interest is scanned and the best match is found between it and the template, using all desired "Degrees of Freedom" below.

- **Score**: Every sample character is compared to the corresponding template character. Character quality indicators are computed, collated into "Quality Indicators" on page 58 quality indicators of the text, and compared to acceptance intervals. Unacceptable values have their corresponding characters flagged, a diagnostic code is generated, and global diagnostics summarizing all text and character defects are issued.

Degrees of Freedom

Degrees of freedom can compensate for misalignment and distortion. Each degree of freedom increases the running time, so use them sparingly. In many cases, text and character translation are sufficient. When large amplitude skewing is possible, text translation + skewing can be used. Care must be exercised when combining the other degrees of freedom.

Text translation

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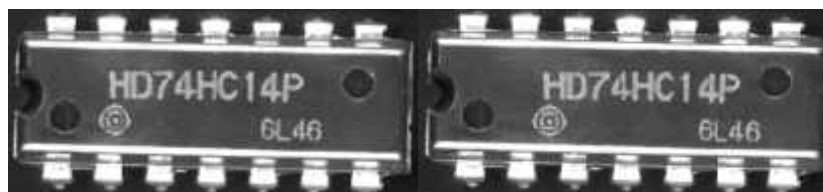
Text translation

Text skewing

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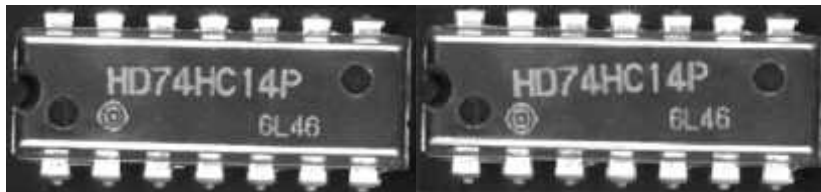
Text skewing

Character translation

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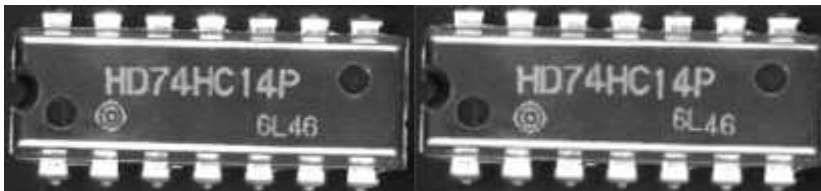
Character translation

Text X/Y-scaling (Advanced - only use if necessary)

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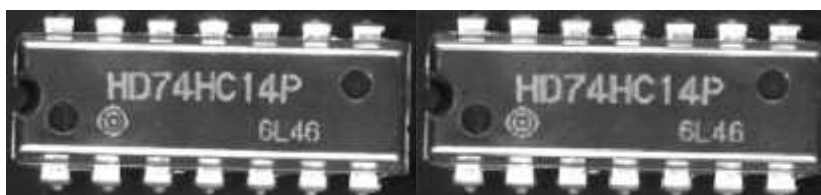
Anisotropic text scaling

Note: *By default, text scaling is supposed to be isotropic. If you think your application might generate anisotropic scaling for a text, you must set the* `IsotropicScaling` *parameters of the* `EOCVText` *corresponding object to* **FALSE** *before inspecting. Working with isotropic scaling results in an effective time saving during inspection.*

Text shearing (Advanced - only use if necessary)

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

their bounding box remains fixed.
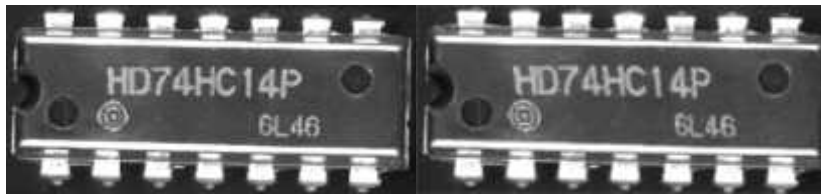
Text shearing

Degrees of Freedom parameters

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

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

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

- **Translation**: Every integer value in the ranges `ShiftXBias ± ShiftXTolerance` and `ShiftYBias ± ShiftYTolerance` is tried. However, for efficiency reasons, the `ShiftXStride` and `ShiftYStride` parameters can be set to a value larger than **1**, so that a gross location pass with the specified stride is followed by a finer one with unit stride. (The expected speed-up is on the order of the square of the `Stride` parameter.) Anyway, choosing too large a value may cause mismatches when local maxima are present. (A value on the order of a fraction of the character size is recommended.)
- **Skewing, scaling and shearing**: the `SkewCount`, `ScaleXCount`, `ScaleYCount` and `ShearCount` parameters indicate the number of values tried for each degree of freedom. The execution time increases as the product of these counts.
  When a degree of freedom is not used, its count must be left as **1**.
  When `SkewCount` is set to **0**, EasyOCV automatically chooses an appropriate count value.

Quality Indicators

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

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

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

Template image, foreground and background (white pixels)

Sample image, foreground and background (white pixels)

Area-Based Quality Indicators

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

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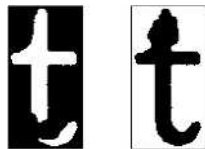
Foreground and background areas (white pixels)

> **Note:** *The area-based indicators rely on thresholding of the sample image. If necessary, the threshold level must be compensated for a change in intensity (automatic thresholding).*

Gray Sum-Based Quality Indicators

A different measure of the amount of light reflected by the marking is given by sums rather than counts: the foreground [background] sample sum is defined as the sum of the gray-level values of all pixels of the foreground [background] region of the sample image. The foreground [background] template sum is the same feature computed on the template image to provide a reference value. Optionally, the sums are normalized with respect to the reference foreground and background average gray-levels to compensate for possible changes in gain (contrast) and offset (intensity).
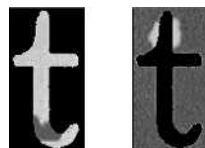
Foreground and background sums

Note: *The sum-based indicators do not rely on thresholding of the sample image, but the reference foreground and background gray-levels may take into account changes in gain and offset. Characters are accepted or rejected by comparing the indicator values of the sample and template: if the difference is larger than the specified tolerance, a defect is reported.*
A smaller foreground value indicates under-printing or missing character parts. A larger background value indicates over-printing or spurious character parts. Character mismatches provoke both kinds of anomalies.

Correlation-Based Quality Indicators

Normalized correlation rates mismatches between two images. The correlation parameter is a global score in range **0** to **1**, which is implicitly corrected for a change in gain and offset.

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

Reporting

Defects are reported in three ways:

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

Advanced Features

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

> ☐ Average values show the long-term trend behavior of the system and detect marking drifts.

> ☐ Standard deviations show process repeatability and detect appearance of slack.
> It is customary to select a tolerance value that is a small multiple of the observed standard deviation (±2 sigma or ±3 sigma criterion).

Programming with EasyOCV

Introduction

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

- A model can be loaded once for all.
- Every acquired image is inspected.

- The inspection results are graphically displayed on top of the image.
- A diagnostics report of quality indicators and statistics can be generated.

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

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

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

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

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

1. Set ROI for inspection

The ROI should be placed center on the marking, in a position that is repeatable with respect to the marking substrate. This is achieved either when the position of the inspected object is known and stable, or when the object has been located by pattern matching or edge measurement. .

The ROI :

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

2. Inspect

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

3. Draw inspected items

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

Selected and unselected items can be drawn in different colors.

Items with detected defects appear crossed by their main diagonal.

4. Retrieve Diagnostics and Quality Indicators

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

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

■ Retrieving text diagnostics and parameters involves a loop where all texts are visited.

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

5. Set Inspection Parameters

During operation working parameters can be adjusted in various ways:

■ **Global change**: a parameter value may be set for all texts and/or all characters. This is straightforward and requires a single call to `ScatterTextsParameters`, `ScatterTextsCharsParameters`, but before calling, make sure that the parameters you don't want to change are set to an undefined value.

■ **Custom change**: the values can be adjusted individually using a user-defined rule. This approach is similar to the retrieval of parameters using indexed access.

■ **Selective change**: parameter values can be set for texts or characters in a selected state by interactively selecting the text or characters.

Selecting Items Interactively

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

> **Note:** *The rectangle is usually obtained by a dragging operation. A degenerate rectangle (reduced to a single point) can be used to handle point clicking.*

Since the toggling mechanism combined with the possible rectangle extent and current selection mode is tricky, let us give a few examples.

Assume a model of three texts in the following states: `Selected`, `Selected`, `Unselected`.

■ Using a rectangle that contains all three of them will set them to states `Unselected`, `Unselected`, `Selected` (SSU -> UUS).

■ Using a rectangle that touches the first of them will set the states to `Unselected`, `Selected`, `Unselected` (SSU -> USU).

Now consider the same operations applied to the selected texts only.

■ Using a rectangle that contains all three texts will set them to states `Unselected`, `Unselected`, `Unselected` (SSU -> UUU).

■ Using a rectangle that touches the first of them will set the states to `Unselected`, `Selected`, `Unselected` (SSU -> USU).

Now consider the same operations applied to the unselected texts only.

■ Using a rectangle that contains all three texts will set them to states `Selected`, `Selected`, `Selected` (SSU -> SSS).

■ Using a rectangle touches the first of them will leave their states unchanged (SSU -> SSU).

> **Note:** *This selection mechanism applies to texts and characters at inspection time (`SelectSample...`). It also applies to free objects, free characters and texts during the model edition phases (`SelectTemplate...`).*

Compute Inspection Statistics

Using EasyOCV, gathering statistical information on the process is possible. For each measured parameter (location parameters and quality indicators), the average and standard deviation can be estimated from a number of samples.

The procedure is straightforward: after an image has been inspected, one can request that the measured parameters be taken into account as valid samples by calling `UpdateStatistics`. (If, for any reason, the sample is to be rejected, just do not call `UpdateStatistics`.) After at least two sample images have been processed, the average and standard deviations can be obtained. The standard mechanisms for text and character parameters retrieval can be used.

To compute the statistics afresh on new samples, start by calling `ClearStatistics`. The number of samples accumulated so far is given by `StatisticsCount`.

Adjust Inspection Parameters from Statistics

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

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

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

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

Interactively Edit a Model

Writing an OCV model editor requires a good understanding of windowed applications design. In particular, it is important to know how to manage the mouse cursor movements, when and how to refresh the display, handle dragging of selection rectangles and the like. It is out of the scope of this documentation to explain these features which are deeply related to Windows programming. Also note that the level of functionality, from blind -no operator intervention- to full fledged editing is a matter of taste and of programming skill.

Recall the steps in defining the model structure:

1. An `ECodedImage` object is used to segment the image into blobs (`BuildObjects` method).
2. Possibly, blob selection by all means provided in EasyObject (legacy), is performed (`SelectObjectsUsingFeature` or `SelectObjectsUsingPosition`). In particular, small blobs generated by noise should be unselected.
3. The selected blobs are passed to the OCV object and enter the free objects list (EasyObject (legacy)'s blobs become OCV's free objects, or `TemplateObjects`).

4. At this point, the objects in the free list can be selected/unselected interactively.

5. The (selected) free objects are then used to generate free characters, using one of the available grouping policy (free objects become free characters, or `TemplateChars`).

6. At this point, the free characters can be selected/unselected interactively.

7. The (selected) free characters are then used to generate texts. The default policy is to group all free characters in a single piece of text (free characters become texts, or `TemplateTexts`; these texts now contain embedded characters, or `TemplateTextChars`).

In the simplest form of a model editor, steps 4 and 6 can be skipped, meaning that all free objects and all free characters will enter the model. A better editor will allow withdrawal of unwanted items and explicit grouping (steps 4 and 6). An even more powerful editor should allow grouping as well as ungrouping (backwards from 3 to 2, from 5 to 4, from 7 to 6).

At any time, the following operations can be handled:

- The model components can be selected/unselected interactively (using `SelectTemplateObjects`, `SelectTemplateChars`, `SelectTemplateTexts`).
- New items can be grouped to form new higher level items (using `CreateTemplateObjects`, `CreateTemplateChars`, `CreateTemplateTexts`).
- Items can be ungrouped by destroying the higher level item (using `DeleteTemplateTexts`, `DeleteTemplateChars`, `DeleteTemplateObjects`).

A clean way to organize the editor is to define a sequence of separate phases dealing with objects, free objects, free characters and texts.

Advanced Features: Change contrast, location mode, Quality indicators and resample characters

Contrast Parameters

Image contrast is an important factor during both learning and inspection.

The background and foreground information must be separated using an appropriate threshold, that may be determined automatically. After a threshold is given, the average gray level of the background and foreground are computed separately, and become the reference gray levels. These are used to measure the image contrast and normalize the gray level quality indicators if needed.

The **background** and **foreground** reference gray levels are computed for both the **template** and **sample** images. See the `EOCVChar` properties. The sample contrast may then be compared to the templates contrast reference value, to diagnose an over-contrasted or under-contrasted image before further analysis.

> **Note:** *The template threshold directly influences the thickness of the blobs in the model.*

> **Note:** *The sample threshold influences the reference gray levels of the sample image.*

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

Location Modes

For location of the model components, a search process is used during which EasyOCV tries to

find the character edges in the sample image, possibly transformed. Four location modes are provided: **raw**, **binarized**, **gradient** and **Laplacian**. See enumeration constants `ELocationMode`.

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

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

Location Score

It is highly recommended to keep the default reduction of location scores option turned on. Reduction consists of dividing a raw location score by the number of points its computation required. Thus, the calculated scores do not depend on the number of used points any longer; the number of used points may be decreased without degrading localization to save potential time, if necessary.

Location scores may also be normalized. This option is useful if the lighting conditions of sample images are not the same, or when template and sample images have obviously different reference gray levels. The action of this option is equivalent to performing a global contrast correction on the sample image (or ROI). Location scores do not depend on reference gray levels, so are more reliable.

Finally, `AccurateTextsLocationScores` provides an alternative way to compute text location score. During the location process, EasyOCV tries first to locate texts using their contours, which are sets of fixed points one from each other. This rigid definition of text contour has the drawback of making the library return poor location score values if the sample characters have moved from their nominal positions, and may result in a false alarm.

If the `AccurateTextLocationScores` property is turned on, a text location score will be computed as sum or average of the characters location scores that form the text (depending on the state of the `ReduceLocationScore` property). This way, texts location scores become independent from the position of the characters they contain.

Used Quality Indicators

For a given inspection case, not all quality indicators are relevant. For instance, it sometimes suffices to use the location scores alone to detect absence of a given marking.

To avoid false alarms raised by unused quality indicators for which the tolerances have not been adjusted, and to avoid unnecessary processing, it is important to activate only the quality indicators in use through the `UsedQualityIndicators` property.

Character Resampling

Normally, when text is inspected with rotation, scaling and/or shearing, some resampling must be performed to compute the quality indicators on the separate characters. If the angles remain small and the scale factors remain very close to unity, this resampling can be avoided by setting the `ResampleChars` parameter to **FALSE**.

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:
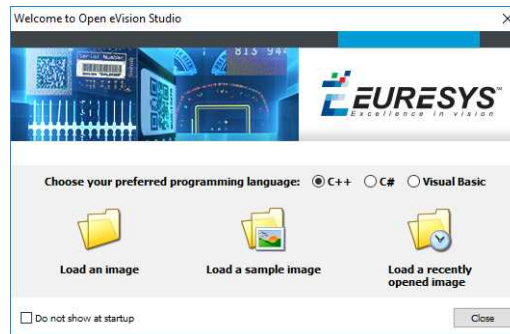
1. Select your programming language.

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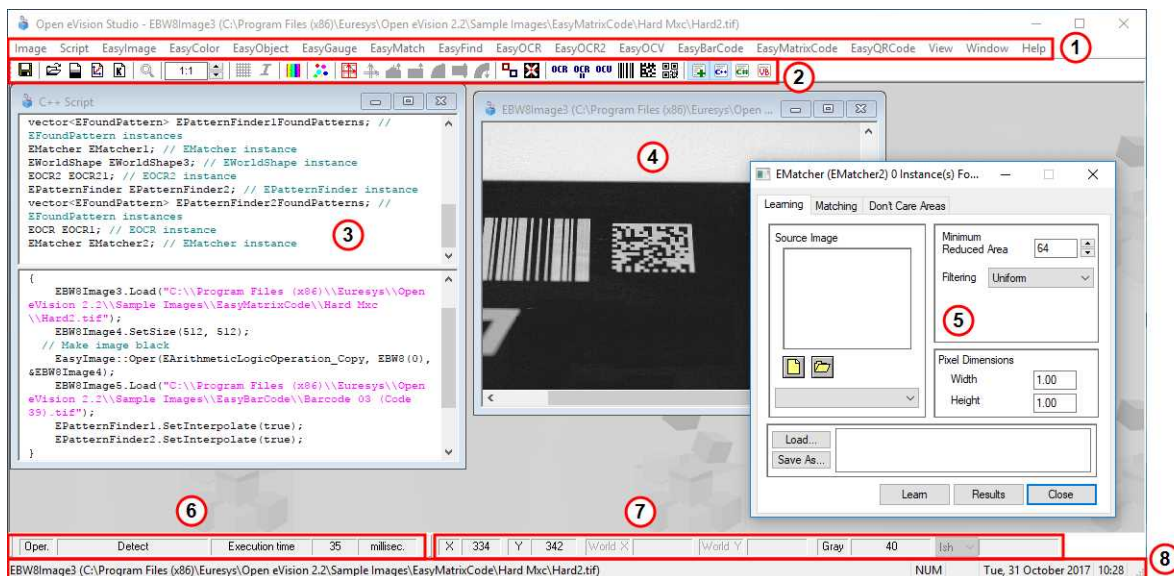
Open eVision Studio graphical user interface (GUI) is organized as follows:

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 a 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 a 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

Usually the first step, when using Open eVision Studio, 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.
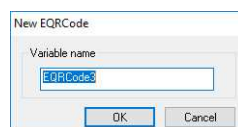
Example of variable creation dialog box for EasyQRCode

4. Click **OK**.

The selected tool dialog box opens.
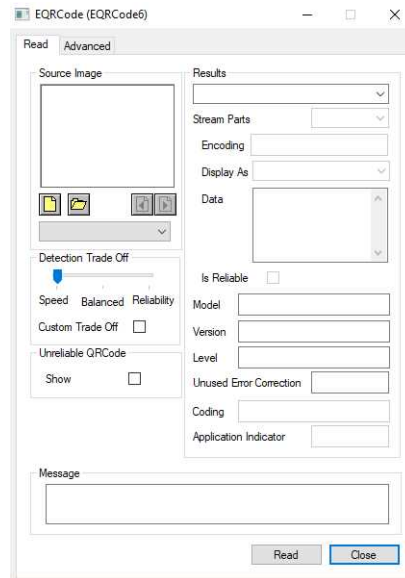
Example of variable creation dialog box for EasyQRCode

The next step is "Step 2: Opening an Image" below.

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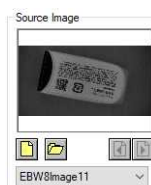
2. If you selected several images, activate one with the **Load Previous** or **Load Next** buttons.

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" below.

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.
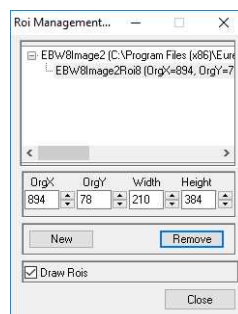
3. Select the image in the tree.

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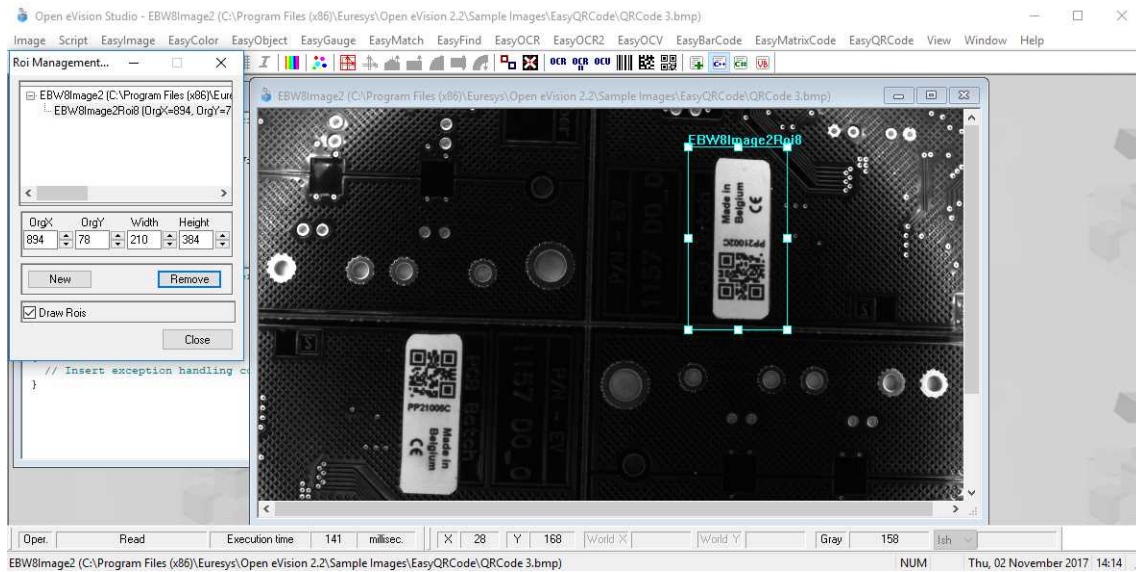
6. Drag the ROI corner and side handles to move it to the required position.

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

The next step is "Step 4: Configuring the Tool" on the next page.

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.
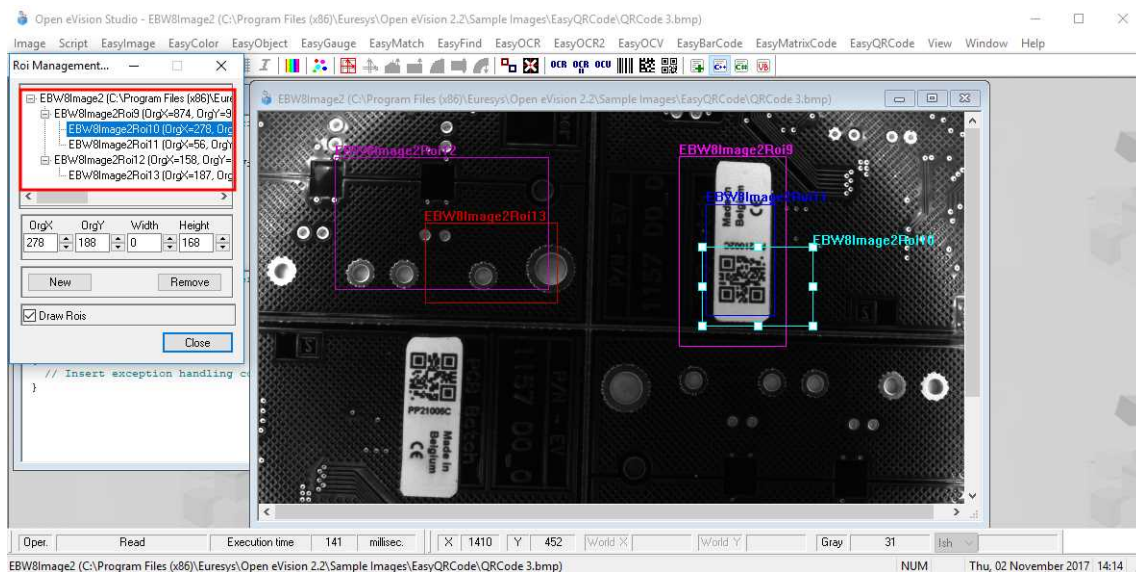
2. Select an ROI in the ROI relation tree.

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.
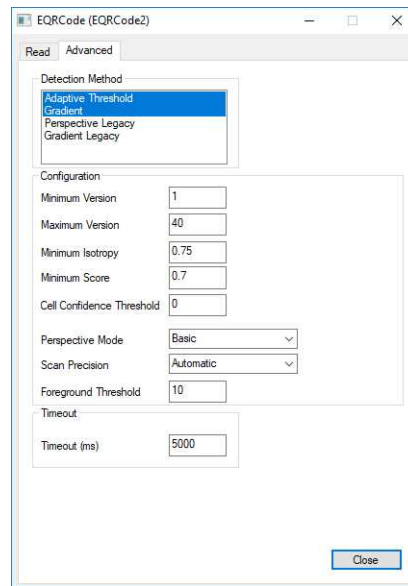
Example of the parameter tab of an EasyQRCode tool

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" below.

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.
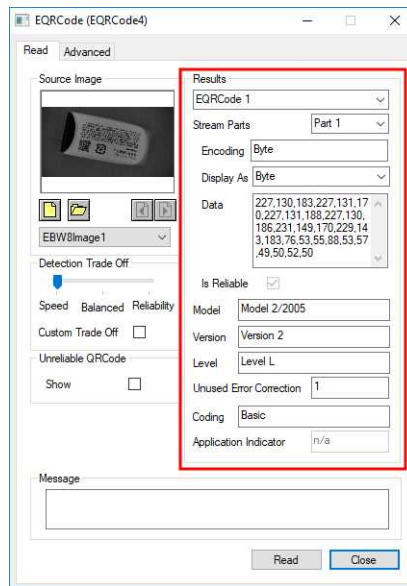
Example of results after reading a QRCode

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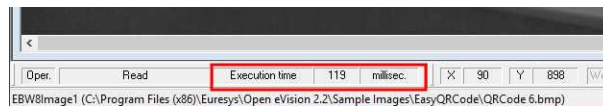
The execution time
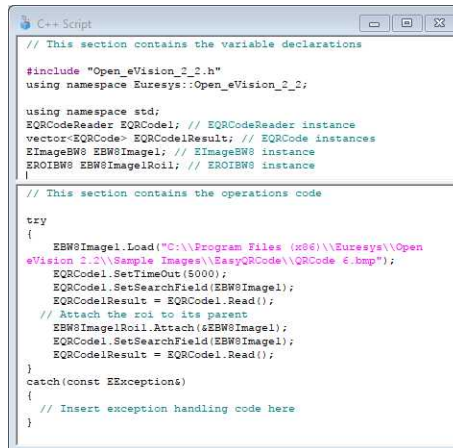
> **TIP**
> The execution time is the actual time that the processing took as measured on your computer. It depends your computer processor, memory, operating system... and, of course, on the processor load at the time of execution. Thus this execution time slightly varies from execution to execution.

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" below.

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.

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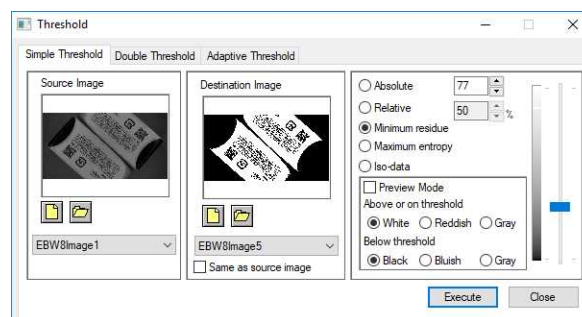
Example of a pre-processing dialog box (Threshold with EasyImage)

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 70).

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.
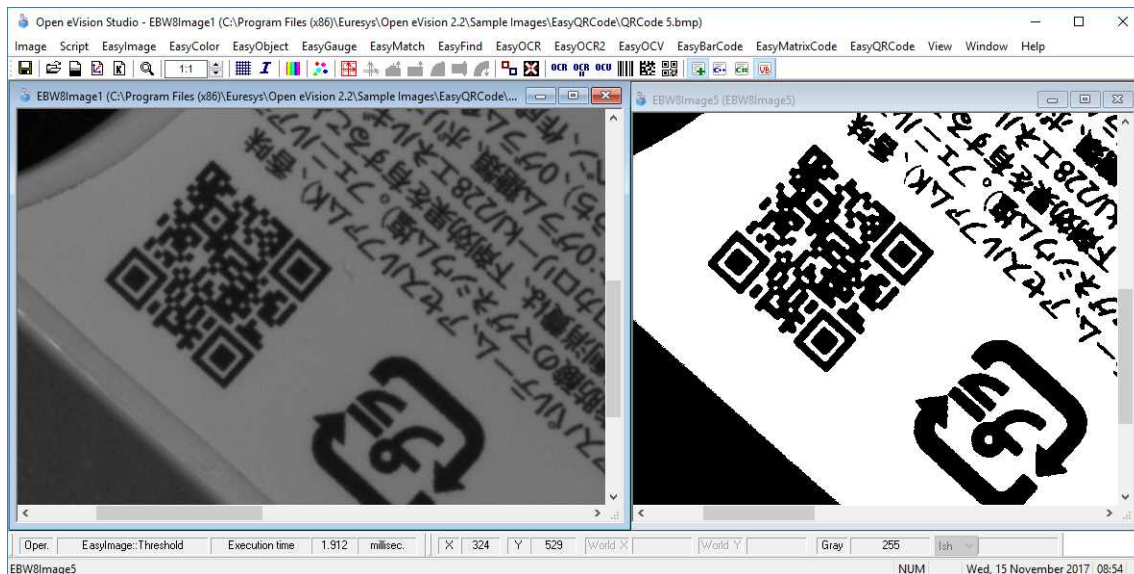
Source and destinations images (Threshold with EasyImage)

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

Saving an image

1. Click in the image you want to save to activate it.

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. EasyOCR

Learning Characters and Creating an EasyOCR Font

"Learning Characters" on page 95

Objective

Following this tutorial, you will learn how to use EasyOCR to learn new characters and save them in an EasyOCR font.

You'll need first to load a source image (step 1). Then you'll set the segmentation parameters to isolate each character (step 2). Each character will have to be learnt (step 3), and finally you'll save all the learnt characters as a font file (step 4). You can also add new characters to an existing font if needed (step 5).
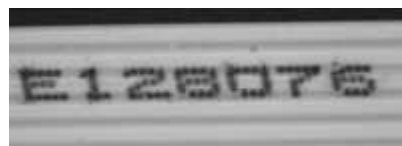
Source image

The image is segmented so that all the characters are detected
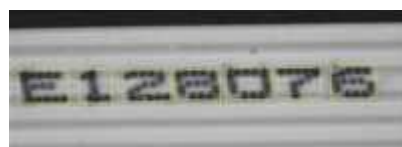
All the characters have been learn

Step 1: Load the source image

1. From the main menu, click **EasyOCR**, then **New OCR Tool**.

2. Keep the default variable name for the new OCR object, and click **OK**.

3. In the **Source Image** tab, click the **Open** icon of the Source Image area, and load the image file `EasyOCR\FlatCable\FlatCable1.tif`.

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

Step 2: Set segmentation parameters

1. Select the **Segmentation Parameters** tab, and move the red frame in the image above a character.

2. Tune each property to get a green bounding box around each character:

   □ **threshold value** = 113

   □ **characters** color = Black on White

   □ **min width** = 36

   □ **min height** = 31

   □ **spacing** = 4

   □ **max width** = 98

   □ **max height** = 72

   □ **noise area** = 9

Step 3: Learn new characters

1. Select the **Learn** tab, and click the character E in the image. You are then prompted to identify the character along with its class. Enter E in the character field, and select the '**EOcrClass_Uppercase**' class. Click **OK**. Whenever a character has been added to the current font, its bounding box turns yellow.

2. Click the character 1 in the image. Enter 1 in the character field, and select the '**EOcrClass_Digit**' class. Click **OK**.

3. Proceed with remaining characters.

Step 4: Save the EasyOCR font

- In the **Font File** tab, click the **Save As...** button. Type a file name for the new EasyOCR font file. Its extension will be .ocr. Finally, click **Save**.

Step 5: Add characters to an existing font

1. In the Source Image tab, click the **Open** icon of the Source Image area, and load the image file `EasyOCR\FlatCable\FlatCable2.tif`.

2. Keep the default variable name for the new image object, and click **OK**.

3. In the **Recognition** tab, click **Execute**. Characters 2 and 8 are read correctly, but A, W and G are not (**low confidence score**). They don't belong to the font.

4. Select the **Learn** tab, and learn the characters A, W, and G (refer to step 3).

5. Then save the font again (refer to step 4). The new characters have been added.

Recognizing Characters According to a Font

Objective

Following this tutorial, you will learn how to use EasyOCR to recognize characters, regarding to a specific font.

You'll need first to load a source image (step 1), and an EasyOCR font file (step 2). Then you'll perform the characters recognition (step 3).
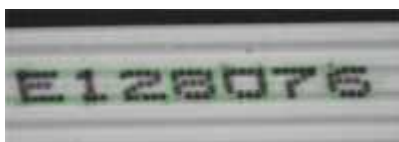
Characters matching the font are automatically detected

Results after explicit recognition

Step 1: Load the source image

1. From the main menu, click **EasyOCR**, then **New OCR Tool**.

2. Keep the default variable name for the new OCR object, and click **OK**.

3. In the Source Image tab, click the **Open** icon of the Source Image area, and load the image file `EasyOCR\FlatCable\FlatCable1.tif`.

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

Step 2: Load the font file

● In the **Font File** tab, click **Load**, and select the font file `EasyOCR\FlatCable\FlatCable.ocr`. In the image, the detected characters are highlighted in green.

Step 3: Recognize the characters

● In the **Recognition** tab, click **Execute** to trigger the recognition of the detected characters. The recognized characters appear in the Recognition results area. Further information about each character can be found in the table.

5.2. EasyOCV

Creating an EasyOCV Model File

Objective

Following this tutorial, you will learn how to build an EasyOCV model file.

You'll need first to load an image (step 1). Then you'll learn the model file (step 2), and perform the mark inspection (step 3).

Learned model

Step 1: Load the source image

1. From the main menu, click **EasyOCV**, then **New OCV Tool**.

2. Keep the default variable name, and click **OK**.

3. In the **Learn** tab, click the **Open** icon of the Source Image area, and load the image file `EasyOCV\T20\T20 A.bmp`.

4. Keep the default variable name, and click **OK**.

Step 2: Learn the model

1. In the **Learn** tab, click **Next** subsequently for three times.

2. Click **Finish**.

The learning process is complete.

Step 3: Save the model file

1. In the **Inspect** tab, click the **Save As...** button.

2. Type a file name for the new EasyOCV model file. Its extension will be .ocv.

3. Finally, click **Save**.

Inspecting Characters in an Image According to a Model File

Objective

Following this tutorial, you will learn how to load an EasyOCV model file, and to perform mark inspection in an image.

You'll need first to load source images (step 1), and a model file (step 2). Then, you'll perform the mark inspection (step 3).

Mark inspection in multiple source images

Step 1: Load the source images

1. From the main menu, click **EasyOCV**, then **New OCV Tool**.

2. Keep the default variable name for the new OCV object, and click **OK**.

3. In the **Inspect** tab, click the **Open** icon of the **Source Image** area, and load the image files `EasyOCV\T20\T20 A.bmp` to `T20 E.bmp`. Use the shift key to select multiple files.

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

Step 2: Load the model file

1. In the **Inspect** tab, click the **Load** button.

2. Select the model file `EasyOCV\T20\T20.ocv`.

Step 3: Perform the mark inspection

1. In the **Inspect** tab, click **Execute** to trigger the mark inspection.

Texts and characters are highlighted in the source image.

2. In the **Parameters and results** tab, click **Character Overview** and **Text Overview** to display further information on characters and texts.

3. Click the **Quality** button, and select characters in the image, to display quality indicators.

4. In the **Inspect** tab, click the **Load Next** and **Load Previous** icons to browse through the multiple images.

In the `T20 E` image, the character 2 is underprinted, and the character 0 is overprinted.

Inspecting Characters in an ROI According to a Model File

Objective

Following this tutorial, you will learn how to load an EasyOCV model file, and to perform mark inspection in an ROI of a source image.

You'll need first to load a source image and define an ROI where to perform the mark inspection (steps 1-2). Then, you'll load a model file (step 3). Finally, you'll perform the mark inspection (step 4).
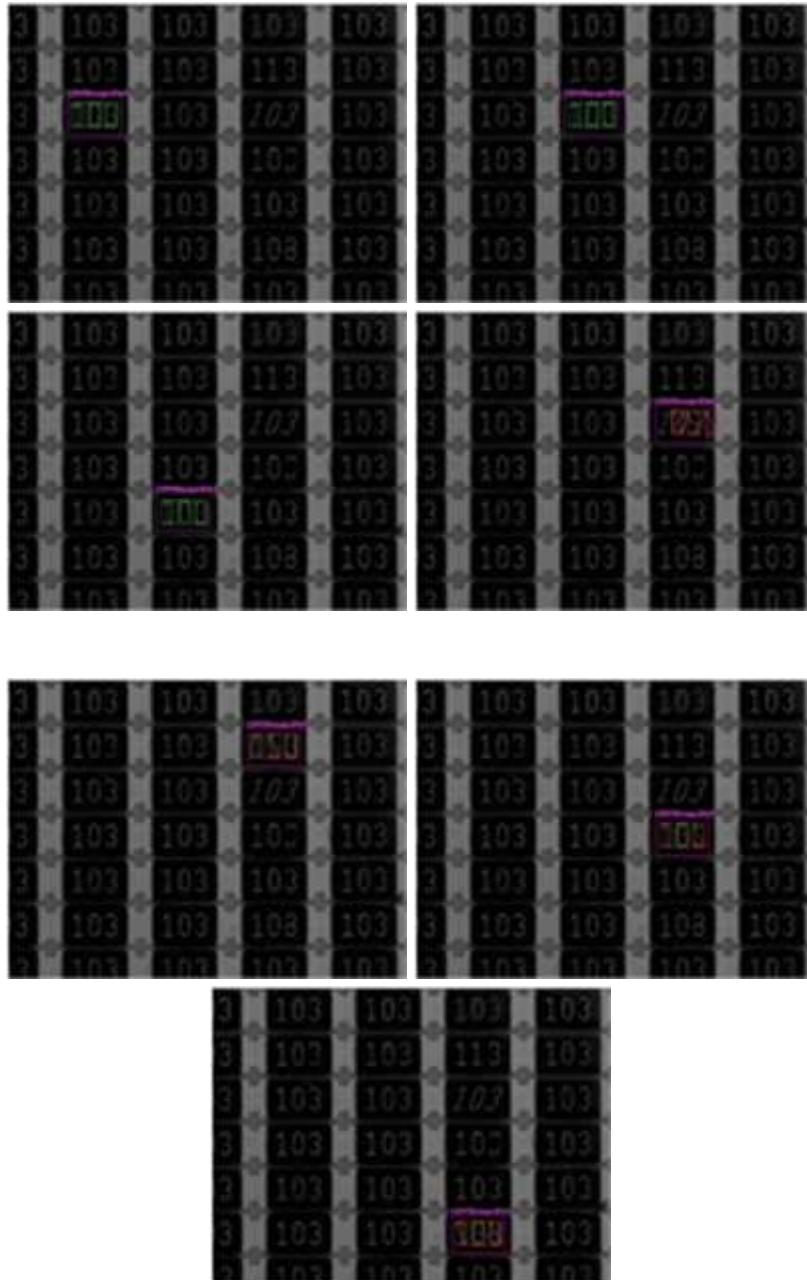
Mark inspection in multiple ROIs

Step 1: Load the source image

1. From the main menu, click **EasyOCV**, then **New OCV Tool**.

2. Keep the default variable name for the new OCV object, and click **OK**.

3. In the **Inspect** tab, click the **Open** icon of the Source Image area, and load the image file `EasyOCV\103\103.bmp`.

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

Step 2: Define the 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. Enter the following coordinates in the ROI management dialog box: 118, 181, 104, 74 for OrgX, OrgY, Width, and Height respectively.

Step 3: Load the model file

1. In the **Inspect** tab, click the **Load** button.

2. Select the model file `EasyOCV\103\103.ocv`.

Step 4: Perform the mark inspection

1. In the **Inspect** tab, select the ROI object from the Source Image drop-down list, and click **Execute** to trigger the mark inspection. Texts and characters are highlighted in the source image.

2. In the **Parameters and results** tab, click **Character Overview** and **Text Overview** to display further information on characters and texts.

3. Click the **Quality** button, and select characters in the image, to display quality indicators.

4. Change the ROI position (OrgX, OrgY), an click **Execute Inspection** at each new position to trigger the inspection again:

   □ (288, 181): OK.

   □ (288, 349): OK.

   □ (462, 181): the numbers are slanted.

   □ (462, 95): the number 0 is wrong.

   □ (462, 266): the number 3 is broken.

   □ (462, 438): the number 3 is out of shape.

Learning a Model Using Statistics (1)

"Statistical Learning" on page 103

Objective

Following this tutorial, you will learn how to use EasyOCV to learn an EasyOCV model, and tune it using statistics.

You'll need first to load a reference image (step 1), and learn the model (step 2). Then you'll add tolerances on the model position (step 3), and use statistics on several images to tune the model (step 4). Finally, you'll perform the mark inspection on multiple images (step 5).

EasyOCVISE

Mark inspection in multiple images

Step 1: Load the reference image

1. From the main menu, click **EasyOCV**, then **New OCV Tool**.

2. Keep the default variable name, and click **OK**.

3. In the **Learn** tab, click the **Open** icon of the Source Image area, and load the image file `EasyOCV\ISE\ISE 01.bmp`.

4. Keep the default variable name, and click **OK**.

Step 2: Learn the model

1. In the **Learn** tab, click **Next** to edit the segmented objects. The characters "ASS", "MB" and "an" are grouped (same color segmentation). They need to be separated.

2. In the image, draw lines to separate touching characters.

3. Click **Next** twice, and click **Finish**.

The learning process is complete.

Step 3: Add tolerances to the model

1. In the **Parameters and results** tab, click **Position** in the **Selected texts parameters** area.

2. To allow more translation freedom, enter 80 and 60 as ShiftX and ShiftY tolerances.

3. Click **Close**. At this stage, the building of the basic EasyOCV model is achieved.

Step 4: Use statistics to tune the model

1. In the **Inspect** tab, click the **Open** icon of the Source Image area, and load the image files `EasyOCV\ISE\ISE 01.bmp` to `ISE 05.bmp`. Use the shift key to select multiple files.

2. Keep the default variable name and click **OK**.

3. In the **Inspect** tab, click **Add to statistics**. Click the **Load Next** icon, and click **Add to statistics**, and so on for all the images.

4. In the **Parameters and results** tab, click the **Quality** button of the **Statistic Tolerance Adjustment** area.

5. Keep the default value in the prompt box. Click **OK**.

Step 5: Inspect multiple images

1. In the **Inspect** tab, click the **Open** icon of the Source Image area, and load the image files `EasyOCV\ISE\ISE 01.bmp` to `ISE 14 (bad character).bmp`. Use the shift key to select multiple files.

2. Keep the default variable name for the new image object, and click **OK**.

3. In the **Inspect** tab, click the **Load Next** and **Load Previous** icons to browse through the multiple images. Take note of the following observations and results.

   □ The images `ISE 01` to `ISE 05` have no defects.

   □ In the image `ISE 06`, the text PBGA is badly printed.

   □ In the image `ISE 07`, the mark is distorted.

   □ In the image `ISE 08`, the mark is blurred.

   □ In the image `ISE 09`, the characters 5 and 2 are incorrect.

☐ In the image `ISE 10`, characters are badly shaped.

☐ In the image `ISE 11`, characters 388 are underprinted.

☐ In the image `ISE 12`, characters ISE are overprinted.

☐ In the image `ISE 13`, two characters are missing.

☐ In the image `ISE 14`, a character is broken.

Learning a Model Using Statistics (2)

"Statistical Learning" on page 103

Objective

Following this tutorial, you will learn how to use the statistic feature and rotation tolerance of EasyOCV to build a model file.

You'll need first to load a reference image (step 1), and learn the model (text and logo) (step 2). Then you'll add tolerances on the model position (step 3), and use statistics on several images to tune the model (step 4). Finally, you'll perform the mark inspection on multiple images (step 5).
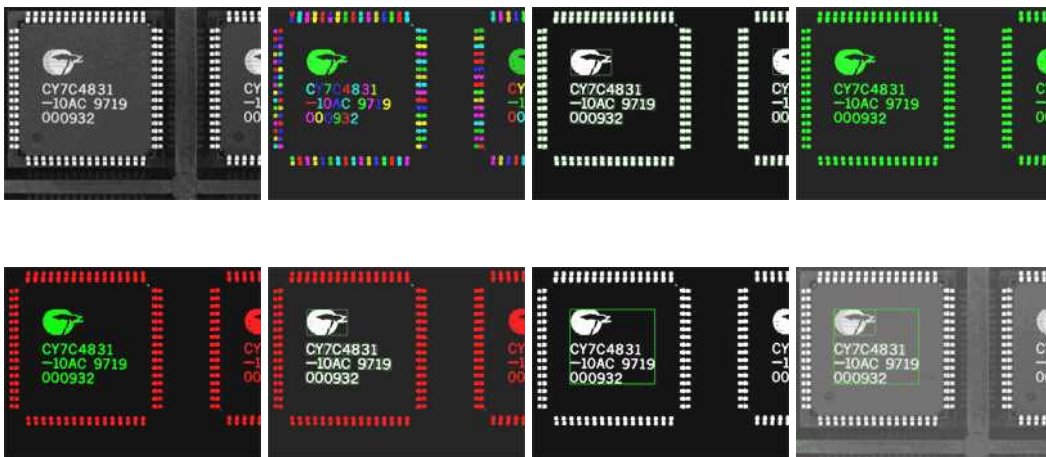
Learning text and logo

Step 1: Load the reference image

1. From the main menu, click **EasyOCV**, then **New OCV Tool**.

2. Keep the default variable name, and click **OK**.

3. In the **Learn** tab, click **Open** icon of the source image area, and load the image file `EasyOCV\CY7C\CY7C 01.jpg`.

4. Keep the default variable name, and click **OK**.

Step 2: Learn the model

1. In the **Learn** tab, click **Next** twice, the characters building dialog box appears.

2. Click **Undo Char(s)**. The text, logo and pins are now highlighted in green.

3. Using your mouse cursor, click-and-drag the rectangle over the text and the logo on the component.

4. Click **Form Auto Chars**.

5. Click **Next**, and then **Finish**. The learning process is complete.

Step 3: Add tolerances to the model

1. In the **Parameters and results** tab, click **Position** in the **Selected texts parameters** area.

2. To allow more translation freedom, enter 90 and 80 as ShiftX and ShiftY tolerances.

3. To allow translation and rotation, enter 5 as Skew tolerance, and 11 as Skew stride/count.

4. To speed up the location process, enter 7 as ShiftX and ShiftY stride/count.

5. Click **Close**. At this stage, building of the basic OCV model is achieved.

Step 4: Use statistics to tune the model

1. In the **Inspect** tab, click the **Open** icon of the Source Image area, and load the image files `EasyOCV\CY7C\CY7C 01.jpg` to `CY7C 15.jpg`. Use the shift key to select multiple files.

2. Keep the default variable name, and click **OK**.

3. Click **Add to statistics**. Click the **Load Next** icon, and click **Add to statistics**, and so on for all the images.

4. In the **Parameters and results** tab, click the **Quality** button from the Statistic Tolerance Adjustment area.

5. Set the Security Factor to 3.5, and click **OK**.

Step 5: Inspect multiple images

1. In the **Inspect** tab, click the **Open** icon of the Source Image area, and load the image files `EasyOCV\CY7C\CY7C 01.jpg` to `CY7C 27.jpg`. Use the shift key to select multiple files.

2. Keep the default variable name for the new image object, and click **OK**.

3. In the **Inspect** tab, click the **Load Next** and **Load Previous** icons to browse through the multiple images. Observe the detected defaults.

6. Code Snippets

6.1. Basic Types

Loading and Saving Images

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

// Images constructor
EImageBW8 srcImage= new EImageBW8();
EImageBW8 dstImage= new EImageBW8();

// 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

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

// Images constructor
EImageBW8 srcImage= new EImageBW8();

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

//Pointer to the third-party image buffer
IntPtr imgPtr = bufferPointer;

// ...

// 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

```
///////////////////////////////////////////////////////////
```

```
// This code snippet shows the recommended method to access  //
// the pixel values in a BW8 image.                          //
/////////////////////////////////////////////////////////////

using System.RunTime.InteropServices;

IntPtr pixAddr;
byte pix;

//...

for(int y = 0; y < height; ++y)
    pixAddr = bw8Image.GetImagePtr(0,y)
    for(int x = 0; x < width; ++x)
        pix = Marshal.ReadByte(pixAddr,x)
```

ROI Placement

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

// Image constructor
EImageBW8 parentImage= new EImageBW8();

// ROI constructor
EROIBW8 myROI= new EROIBW8();

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

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

Vector Management

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

// EBW8Vector constructor
EBW8Vector ramp= new EBW8Vector();
EBW8 bw8 = new EBW8();

// Clear the vector
ramp.Empty();

// Fill the vector with increasing values
for(int i= 0; i < 128; i++)
{

    bw8.Value = (byte)i;
    ramp.AddElement(bw8);
}

// Retrieve the 10th element value
EBW8 value = ramp.GetElement(9);
```

Exception Management

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

try
{
    // Image constructor
    EImageC24 srcImage= new EImageC24();

    // ...

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

catch(EException exc)
{

    // Retrieve the exception description
    string error = exc.What();
}
```

6.2. EasyOCR

Learning Characters

```
////////////////////////////////////////////////////
// This code snippet shows how to learn characters  //
// based on an image featuring a known text and     //
// save the corresponding font file.                //
////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage= new EImageBW8();

// EOCR constructor
EOCR ocr= new EOCR();

// Text to be learned (all digits)
// Assuming the image contains this text
string text= "0123456789";

// ...

// Create a new font
ocr.NewFont(8, 11);

// Adjust the segmentation parameters
ocr.TextColor= EOCRColor.BlackOnWhite;
ocr.MinCharWidth= 15;
ocr.MaxCharWidth= 50;
ocr.MinCharHeight= 15;
ocr.MaxCharHeight= 75;
ocr.NoiseArea= 15;

// Segment the characters
ocr.BuildObjects(srcImage);
ocr.FindAllChars(srcImage);

// Learn the characters
ocr.LearnPatterns(srcImage, text, (int)EOCRClass.Digit);

// Save the font into a file
ocr.Save("myFont.ocr");
```

Recognizing Characters

```
////////////////////////////////////////////////////
// This code snippet shows how to load a font file, //
// perform a default character recognition operation //
// and perform a character recognition operation    //
// using a class filter.                            //
////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage= new EImageBW8();

// EOCR constructor
EOCR ocr= new EOCR();
```

```
// Load the font file
ocr.Load("myFont.ocr");

// ...

// Recognize the characters
string text= ocr.Recognize(srcImage, 10, (int)EOCRClass.AllClasses);

// Alternatively
// Define the character filter (2 letters and 3 digits)
int[] charFilter = new int[5];
charFilter[0] = (int)EOCRClass.UpperCase;
charFilter[1] = (int)EOCRClass.UpperCase;
charFilter[2] = (int)EOCRClass.Digit;
charFilter[3] = (int)EOCRClass.Digit;
charFilter[4] = (int)EOCRClass.Digit;

// Recognize the characters with class filtering
text = ocr.Recognize(srcImage, 10, charFilter);
```

6.3. EasyOCR2

Detecting Characters

```
/////////////////////////////////////////////////////////
// This code snippet shows how to detect characters    //
// in an image, using a few parameters and a topology  //
/////////////////////////////////////////////////////////

// Load an Image
EImageBW8 image = new EImageBW8();
image.Load("image.tif");

// Attach a ROI to the image
EROIBW8 roi = new EROIBW8();
roi.Attach(image, 50, 224, 340, 96);

// Create an EOCR2 instance
EOCR2 ocr2 = new EOCR2();

// Set the expected character sizes
ocr2.CharsWidthRange = new EIntegerRange(25,25);
ocr2.CharsHeight = 37;

// Set the text polarity, in this case WhiteOnBlack
ocr2.TextPolarity = EasyOCR2TextPolarity.WhiteOnBlack;

// Set the topology
ocr2.Topology = ".{10}\n.{3} .{4}";

// Detect the text in the image. The output Text structure contains:
// - an individual textbox for each character
// - an individual bitmap image for each character
// - a threshold value to binarize the bitmap image for each character
// All structured in a hierarchy with Lines -> Words -> Characters
EOCR2Text text = ocr2.Detect(roi);
```

```
// Cleanup
text.Dispose();
ocr2.CharsWidthRange.Dispose();
ocr2.Dispose();
roi.Dispose();
image.Dispose();
```
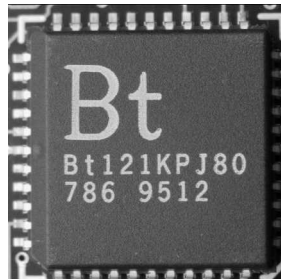

The image used in this code snippet

Learning Characters

```
////////////////////////////////////////////////////
// This code snippet shows how to learn characters  //
// based on an image featuring a known text and     //
// save the corresponding character database        //
////////////////////////////////////////////////////

// Load an Image
EImageBW8 image = new EImageBW8();
image.Load("image.tif");

// Attach a ROI to the image
EROIBW8 roi = new EROIBW8();
roi.Attach(image, 50, 224, 340, 96);

// Create an EOCR2 instance
EOCR2 ocr2 = new EOCR2();

// Set the required parameters
ocr2.CharsWidthRange = new EIntegerRange(25,25);
ocr2.CharsHeight = 37;
ocr2.TextPolarity = EasyOCR2TextPolarity.WhiteOnBlack;
ocr2.Topology = ".{10}\n.{3} .{4}";

// Learn from the reference image:
//   1) Detect the text in the image
EOCR2Text text = ocr2.Detect(roi);
//   2) Set the true values of the text
text.Text = "Bt121KPJ80\n786 9512";
//   3) Add the characters to the character database
ocr2.Learn(text);

// Save the character database
ocr2.SaveCharacterDatabase("myDB.o2d");

// Alternatively, save the model file.
// This will store the character database and the parameter settings
ocr2.Save("myModel.o2m");
```

```
// Cleanup
text.Dispose();
ocr2.CharsWidthRange.Dispose();
ocr2.Dispose();
roi.Dispose();
image.Dispose();
```


The image used in this code snippet

Reading Characters

Reading using TrueType fonts

```
//////////////////////////////////////////////////
// This code snippet shows how to                 //
// - create a character database from TrueType fonts //
// - read the text in an image                    //
//////////////////////////////////////////////////

// Load an image
EImageBW8 image = new EImageBW8();
image.Load("image.tif");

// Attach an ROI
EROIBW8 roi = new EROIBW8();
roi.Attach(src, 50, 224, 340, 96);

// Create an EOCR2 instance
EOCR2 ocr2 = new EOCR2();

// Set the required parameters
ocr2.CharsWidthRange = new EIntegerRange(25,25);
ocr2.CharsHeight = 37;
ocr2.Topology = "[LN]{10}\nN{3} N{4}";
ocr2.TextPolarity = EasyOCR2TextPolarity.WhiteOnBlack;

// Add TrueType character to the character database
ocr2.AddCharactersToDatabase("C:\\Windows\\Fonts\\calibrib.ttf");
ocr2.AddCharactersToDatabase("C:\\Windows\\Fonts\\yugothb.ttc");

// Read text from the image
string result = ocr2.Read(roi);

// Cleanup
ocr2.CharsWidthRange.Dispose();
ocr2.Dispose();
roi.Dispose();
image.Dispose();
```

The image used in this code snippet

Reading using EOCR2 Character Database

```
/////////////////////////////////////////////////
// This code snippet shows how to                //
// - load a pre-made character database          //
// - read the text in an image                   //
/////////////////////////////////////////////////

// Load an image
EImageBW8 image = new EImageBW8();
image.Load("image.tif");

// Attach an ROI
EROIBW8 roi = new EROIBW8();
roi.Attach(src, 50, 224, 340, 96);

// Create an EOCR2 instance
EOCR2 ocr2 = new EOCR2();

// Set the required parameters
ocr2.CharsWidthRange = new EIntegerRange(25,25);
ocr2.CharsHeight = 37;
ocr2.Topology = "[LN]{10}\nN{3} N{4}";
ocr2.TextPolarity = EasyOCR2TextPolarity.WhiteOnBlack;

// Add a pre-made character database to the EOCR2 instance
ocr2.AddCharactersToDatabase("myDB.o2d");

// Read text from the image
string result = ocr2.Read(roi);

// Cleanup
ocr2.CharsWidthRange.Dispose();
ocr2.Dispose();
roi.Dispose();
image.Dispose();
```

Reading using EOCR2 Model file

```
/////////////////////////////////////////////////
// This code snippet shows how to                //
// - load a pre-made model file                  //
// - read the text in an image                   //
/////////////////////////////////////////////////
```

```
// Load an image
EImageBW8 image = new EImageBW8();
image.Load("image.tif");
```

```
// Attach an ROI
EROIBW8 roi = new EROIBW8();
roi.Attach(src, 50, 224, 340, 96);
```

```
// Create an EOCR2 instance
EOCR2 ocr2 = new EOCR2();
```

```
// Load a pre-made model file, this will:
// - (re)set all parameters
// - add the character database in the model file to the EOCR2 instance
ocr2.Load("myModel.o2m");
```

```
// Read text from the image
string result = ocr2.Read(roi);
```

```
// Cleanup
ocr2.Dispose();
roi.Dispose();
image.Dispose();
```

6.4. EasyOCV

Creating an OCV Model

```
///////////////////////////////////////////////////
// This code snippet shows how to create an OCV model //
// from a golden template and save it into a file.    //
///////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage= new EImageBW8();

// EOCV constructor
EOCV ocv= new EOCV();

// ECodedImage constructor
ECodedImage blobs= new ECodedImage();

// ...

// Reset the OCV context
ocv.DeleteTemplateTexts();
ocv.DeleteTemplateChars();
ocv.DeleteTemplateObjects();
ocv.ClearStatistics();

// Set the OCV context
ocv.TemplateImage= srcImage;

// Segment the source image
blobs.Threshold= (int)EThresholdMode.MinResidue;
blobs.BuildObjects(srcImage);

// Compute blobs area and unselect small objects
blobs.AnalyseObjects(ELegacyFeature.Area);
blobs.SelectObjectsUsingFeature(ELegacyFeature.Area, 0, 50,
ESelectOption.RemoveLesserOrEqual);

// Add remaining blobs to the OCV context
ocv.CreateTemplateObjects(blobs);

// Add all selected free objects
ocv.CreateTemplateChars(ESelectionFlag.True, ECharCreationMode.Separate);

// Group all selected free characters in a single text
ocv.CreateTemplateTexts();

// Perform the learning
ocv.Learn(srcImage);

// Save the ocv model into a file
ocv.Save("myModel.ocv");
```

Inspecting

```
///////////////////////////////////////////////////
// This code snippet shows how to load an OCV model //
```

```
// file and perform an inspection.                    //
/////////////////////////////////////////////////////
```

```
// Image constructor
EImageBW8 srcImage= new EImageBW8();
```

```
// EOCV constructor
EOCV ocv= new EOCV();
```

```
// ...
```

```
// Load an EasyOCV model file
ocv.Load("myModel.ocv");
```

```
// Perform the inspection
ocv.Inspect(srcImage, (int)EThresholdMode.MinResidue);
```

Setting Inspection Parameters

```
/////////////////////////////////////////////////
// This code snippet shows how to set characters //
// and texts inspection parameters.              //
/////////////////////////////////////////////////
```

```
// EOCV constructor
EOCV ocv= new EOCV();
```

```
// Temporary EOCVText object for parameters modification
EOCVText text= new EOCVText();
```

```
// Reset the text parameters
text.ResetParameters();
```

```
// Set the text shift tolerance
text.ShiftXTolerance= 30;
text.ShiftYTolerance= 20;
```

```
// Apply the new parameters to all the texts of the ocv context
ocv.ScatterTextsParameters(text, (int)ESelectionFlag.Any);
```

```
// Retrieve the first text (index 0) parameters
text.ResetParameters();
ocv.GetTextParameters(text, 0);
```

```
// Double the shift tolerance
text.ShiftXTolerance= text.ShiftXTolerance * 2;
text.ShiftYTolerance= text.ShiftYTolerance * 2;
```

```
// Apply the new parameters to the ocv context first text only
ocv.SetTextParameters(text, 0);
```

```
// Temporary OCVChar object for parameters modification
EOCVChar ch= new EOCVChar();
```

```
// Reset the character parameters
ch.ResetParameters();
```

```
// Set the character shift tolerance
ch.ShiftXTolerance= 15;
ch.ShiftYTolerance= 10;
```

```
// Apply the new parameters to all the characters of the ocv context
ocv.ScatterTextsCharsParameters(ch, ESelectionFlag.Any, ESelectionFlag.True);
```

Retrieving Diagnostics

```
//////////////////////////////////////////////////////
// This code snippet shows how to perform an inspection //
// and retrieve the diagnostics.                       //
//////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage= new EImageBW8();

// EOCV constructor
EOCV ocv= new EOCV();

// ...

// Load an EasyOCV model file
ocv.Load("myModel.ocv");

// Perform the inspection
ocv.Inspect(srcImage, (int)EThresholdMode.MinResidue);

// Retrieve the OCV inspection diagnostics
if(ocv.Diagnostics != (int)EDiagnostic.Undefined)
{

        // Check if texts have been found
        bool bTextNotFound= ((ocv.Diagnostics & (int)EDiagnostic.TextNotFound) > 0);

        // Check if there is text mismatch
        bool bTextMismatch= ((ocv.Diagnostics & (int)EDiagnostic.TextMismatch) > 0);

        // Check if there is text overprinting
        bool bTextOverprinting= ((ocv.Diagnostics & (int)EDiagnostic.TextOverprinting) > 0);

        // Check if there is text underprinting
        bool bTextUnderprinting= ((ocv.Diagnostics & (int)EDiagnostic.TextUnderprinting) >
0);

        // Check if characters have been found
        bool bCharNotFound= ((ocv.Diagnostics & (int)EDiagnostic.CharNotFound) > 0);

        // Check if there is character mismatch
        bool bCharMismatch= ((ocv.Diagnostics & (int)EDiagnostic.CharMismatch) > 0);

        // Check if there is character overprinting
        bool bCharOverprinting= ((ocv.Diagnostics & (int)EDiagnostic.CharOverprinting) > 0);

        // Check if there is character underprinting
        bool bCharUnderprinting= ((ocv.Diagnostics & (int)EDiagnostic.CharUnderprinting) >
0);
}
```

Statistical Learning

```
//////////////////////////////////////////////////////
// This code snippet shows how to perform a statistical  //
// learning based on several good quality templates.     //
//////////////////////////////////////////////////////
```

```
// Image constructor
EImageBW8 srcImage= new EImageBW8();
```

```
// EOCV constructor
EOCV ocv= new EOCV();
```

```
// ...
```

```
// Clear the statistics
ocv.ClearStatistics();
```

```
// Loop on the number of good quality sample images
for(int i= 0; i < numSampleImages; i++)
{
        // acquire the next sample image into srcImage
        // ...

        // Perform the inspection
        ocv.Inspect(srcImage, (int)EThresholdMode.MinResidue);

        // Update the statistics
        ocv.UpdateStatistics();
}
```

```
// Adjust the tolerance values based on
// the inspected good quality sample images
ocv.AdjustTextsQualityRanges(3.3f, ESelectionFlag.Any);
ocv.AdjustCharsQualityRanges(3.3f, ESelectionFlag.Any, ESelectionFlag.Any);
```