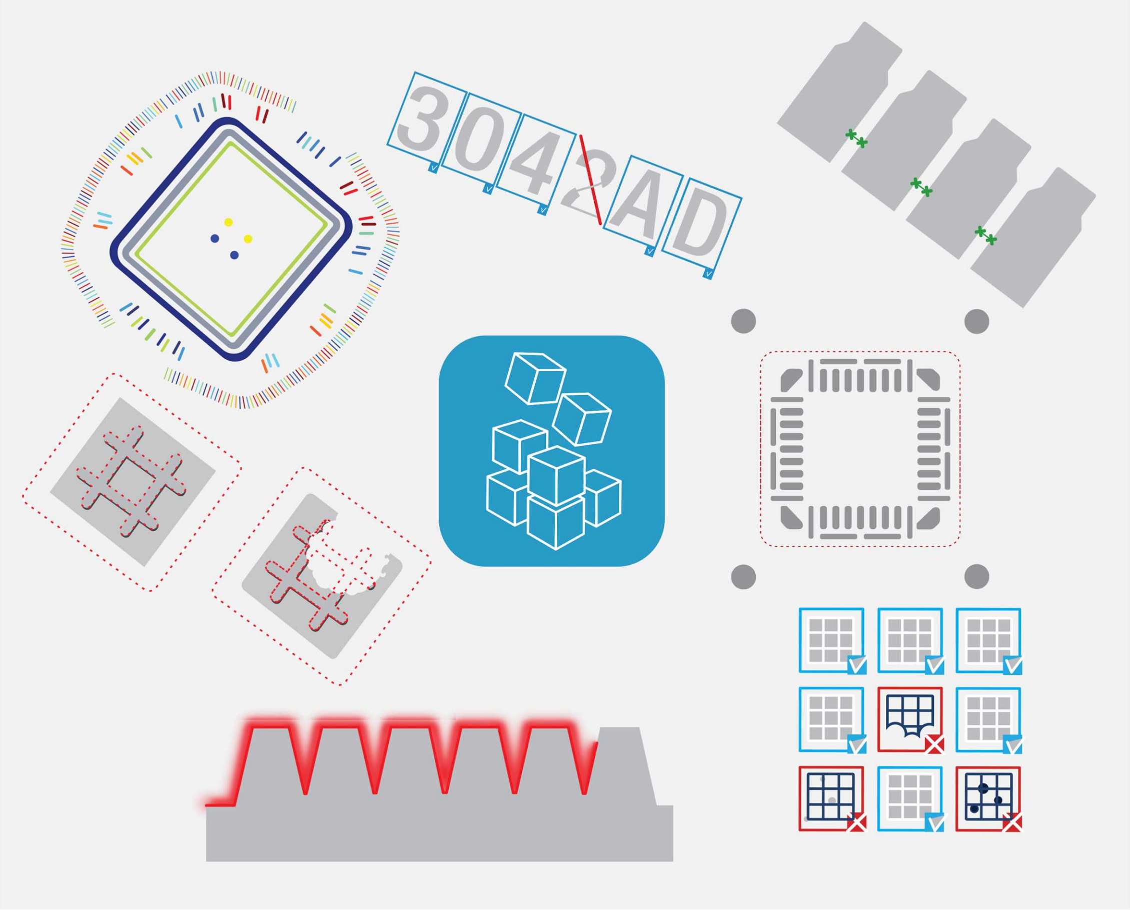
USER GUIDE

Open eVision



Terms of Use

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

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

This documentation is provided with Open eVision 2.11.1 (doc build 1125).

Contents

PART I

GLOBAL FEATURES

1. Installing Open eVision

Installer Package

Open eVision comes as a single installer package `Open eVision X.Y.Z Installer.msi`. It contains everything needed to run or develop applications using **Open eVision**

Installation Types

The **Open eVision** Installer provides the following installation types:

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

Older Versions

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

Command-Line Interface

To install Open eVision with the command line, use:

```
msiexec /i "Open eVision X.Y.Z Installer.msi" /qn INSTALLTYPE=[install_type]"
```

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

For the command prompt to wait for the end of the installation add "`start /wait`" at the start of the command:

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

License Activation

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

> **NOTE**
> On Windows XP, the license Manager requires .NET 2.0.

Supported platforms and requirements

Open eVision in C++

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

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

Open eVision in .NET

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

Using Open eVision in ActiveX

Add a reference to the `Open_eVision_ActiveXApi_X_Y.dll` component.

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

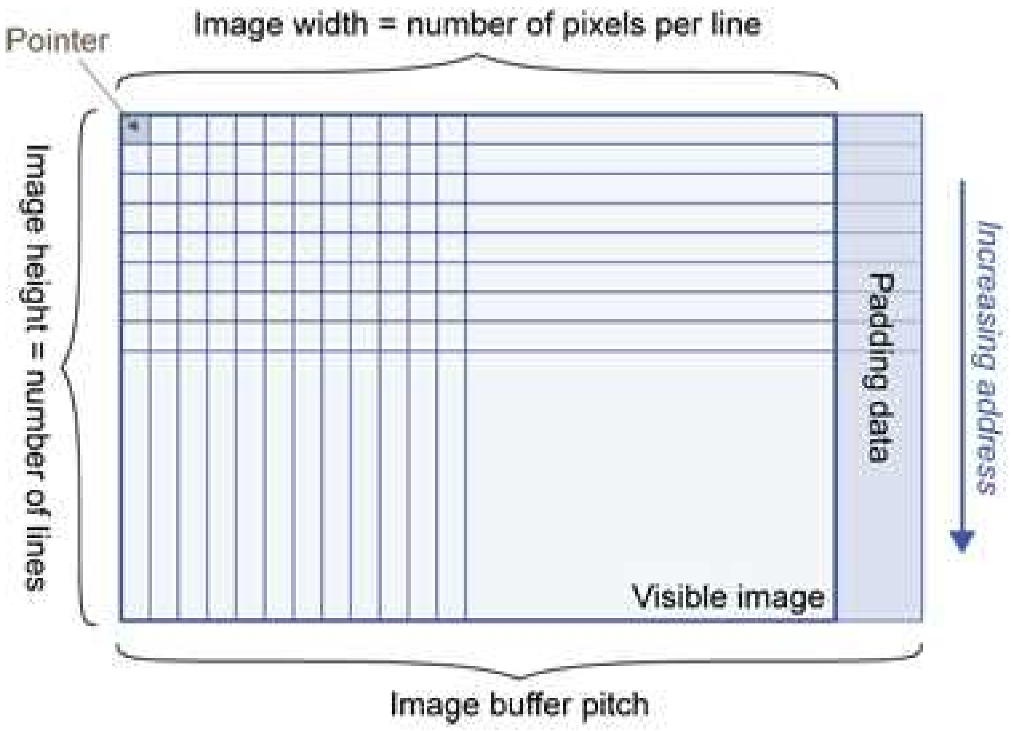

Pointer
Image width = number of pixels per line
Image height = number of lines
Padding data
Increasing address
Visible image
Image buffer pitch

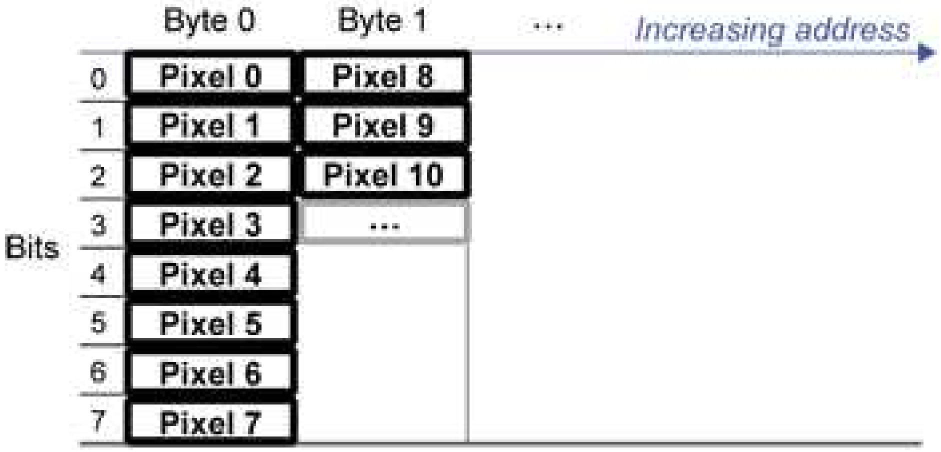
Image Buffer pitch

- 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

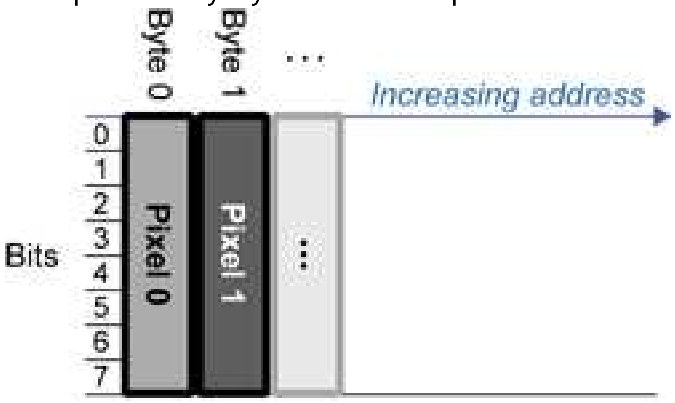
- `EImageBW1` stores 8 pixels in one byte.

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



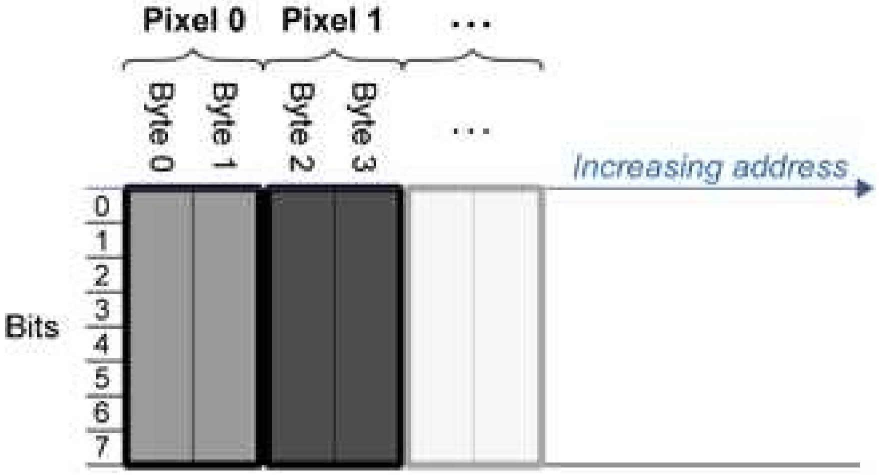
- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

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



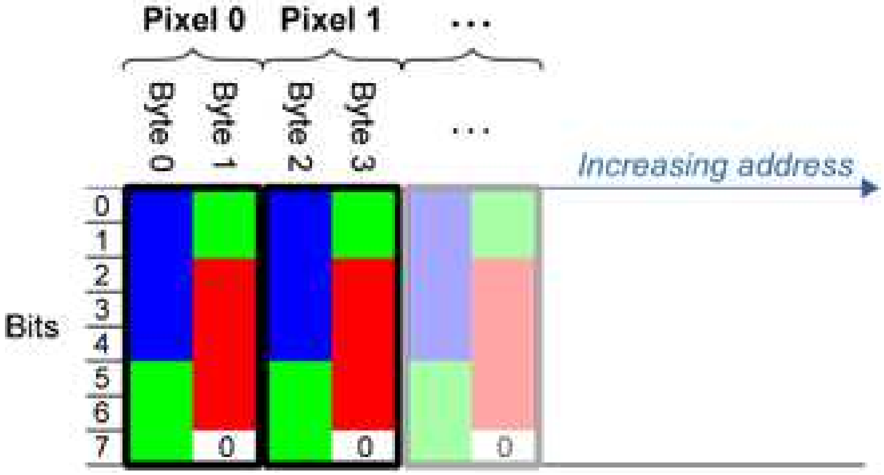
- EImageBW16 stores each pixel in a 16-bit word (two bytes).

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



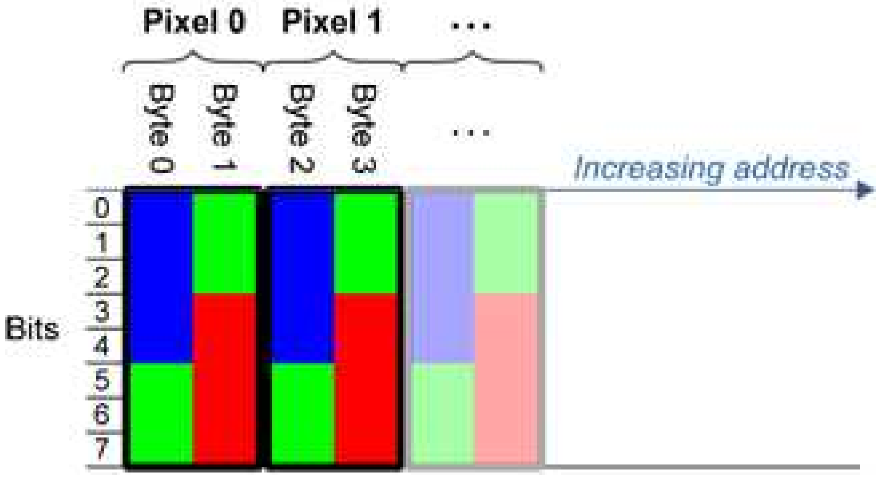
- EImageC15 stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

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



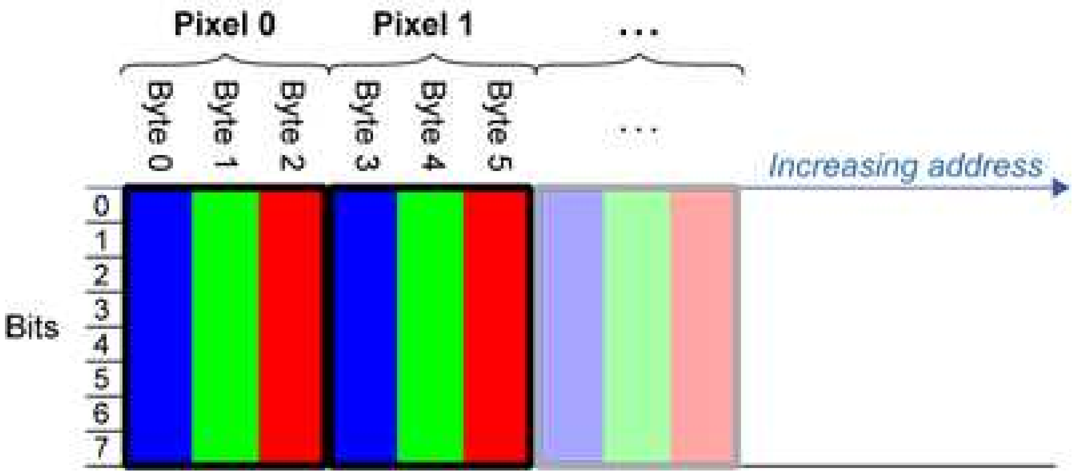
- EImageC16 stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
  The second color component is coded with 6-bits.

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



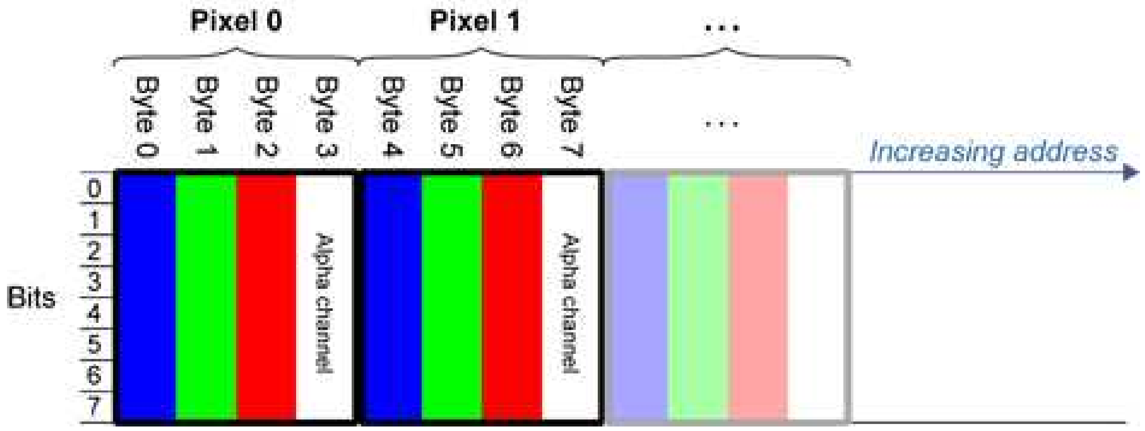
- `EDepthMap16` store each pixel in 2 bytes using a fixed point format.
- `EImageC24` stores each pixel in 3 bytes. Each color component is coded with 8-bits.

  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

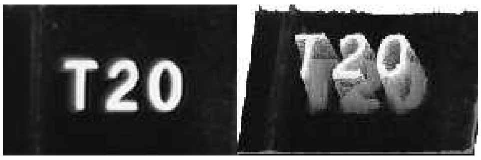
- Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.
- The color palette in 256-color display mode gives optimal rendering. Gray-level 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.

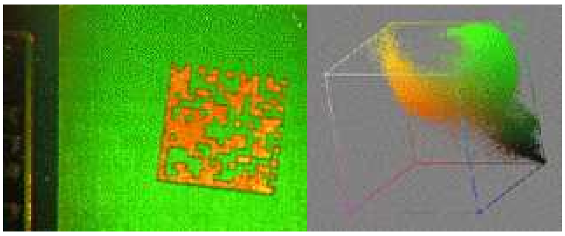
[1] Graphics Device Interface

[2] Microsoft Foundation Class

[3] LookUp Tables

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

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

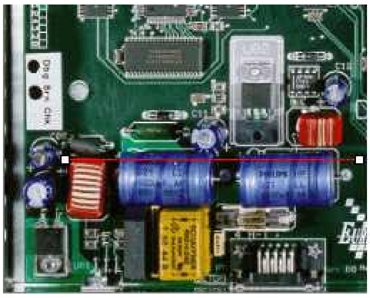


Color histogram rendering

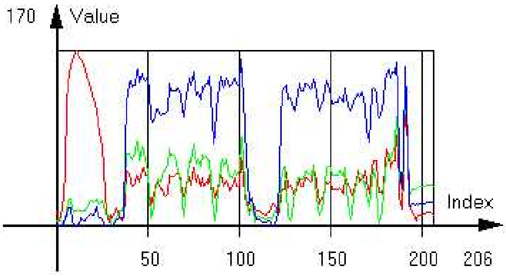
2.7. Vector Types and Main Properties

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

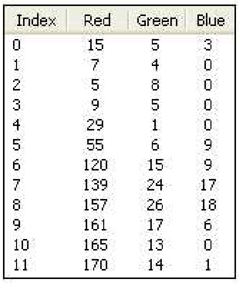
`EVector` is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.



Profile in a C24 image



RGB values plot along profile



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

RGB values array (`EC24Vector`)

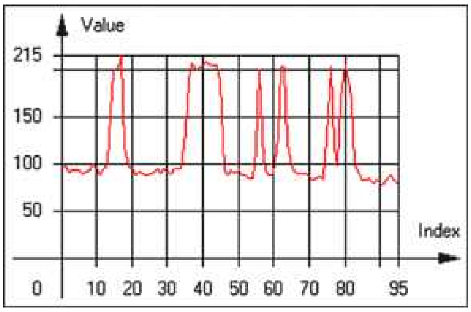
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

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

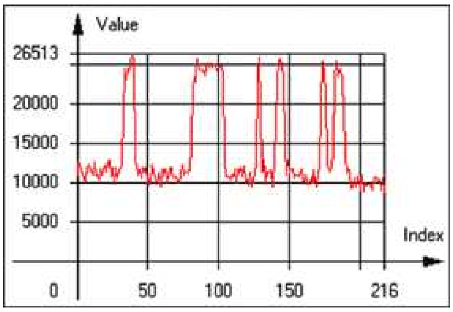
Vector types

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



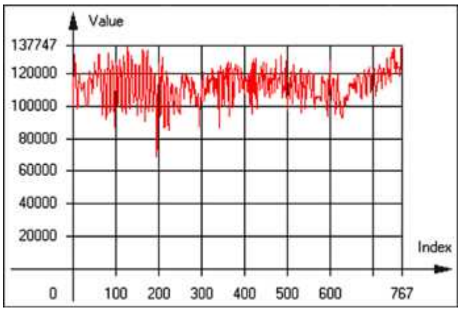
Graphical representation of an EBW8Vector (see Draw method)

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



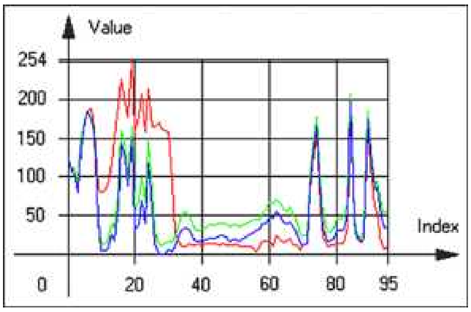
Graphical representation of an EBW16Vector

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



Graphical representation of an EBW32Vector

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



Graphical representation of an EC24Vector

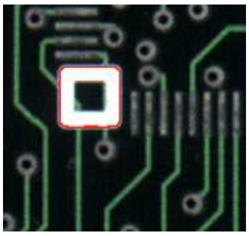
- EBW8PathVector: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
(used by EasyImage::.ImageToPath, EasyImage::.PathToImage, ...).

Graphical representation of an EBW8PathVector (see Draw method)

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

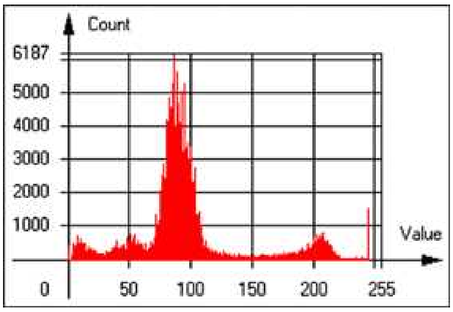
Graphical representation of an EBW16PathVector (see Draw method)

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



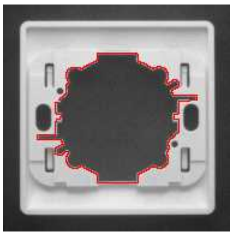
Graphical representation of an EC24PathVector (see Draw method)

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



Graphical representation of an EBWHistogramVector (see Draw method)

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



Graphical representation of an EPathVector (see Draw method)

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

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

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

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

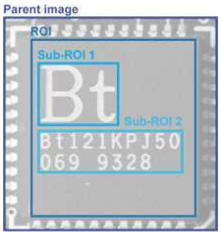
Attachment

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`
	`E3DObjectExtractor::Extract`
	`EZMapToPointCloudConverter::Convert`
EasyObject	`EImageEncoder::Encode`
EasyFind	`EPatternFinder::Find`
	`EPatternFinder::Learn`
EasyOCR2	`EOCR2::Read`
	`EOCR2::Detect`

Library	Method
EasyGauge	EPointGauge::Measure
	ELineGauge::Measure
	ERectangleGauge::Measure
	ECircleGauge::Measure
	EWedgeGauge::Measure

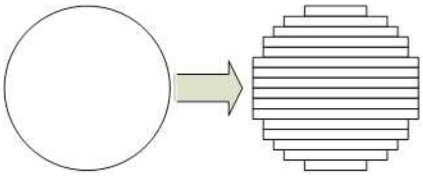
TIP
In the future Open eVision releases, the support of `ERegions` will be gradually extended to all operators.

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.

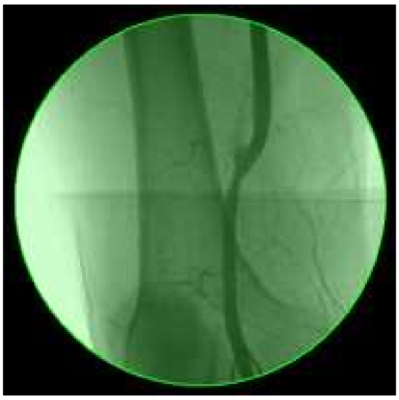
Use these classes to setup geometric regions and modify them with translation, rotation and scaling. The transformation operators return new regions, leaving the source object unchanged.

- `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.

ATTENTION
Observe Precautions for Handling
Electrostatic Sensitive Devices
12-12345-6789089
PS

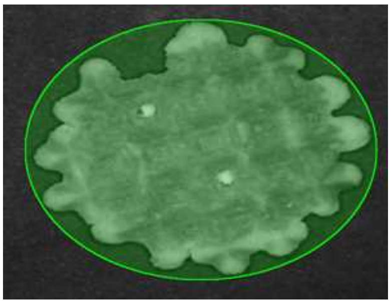
Rectangle region separating a bar code from the background

- `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

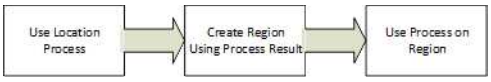
- EPolygonRegion
  - The contour of an EPolygonRegion class is a polygon.
  - It is constructed using the list of its vertices.

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:

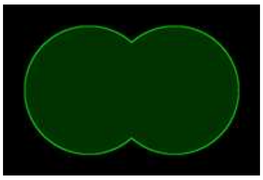
- □ EasyFind: `EFoundPattern`
- □ EasyMatch: `EMatchPosition`
- □ EasyGauge: `ECircle` and `ERectangle`
- □ EasyObject: `ECodedElement`

> **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

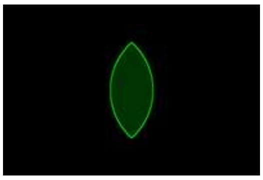
Use the following operations to create a new region by combining existing regions:

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



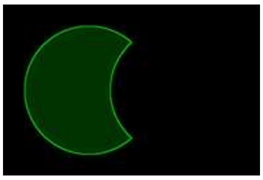
Union of 2 circles

- 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

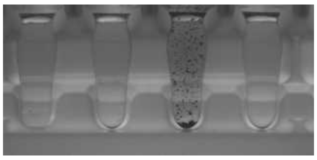
ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 21 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.

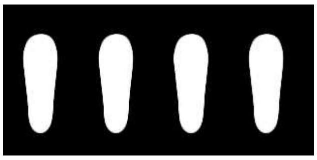
Flexible Masks

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):

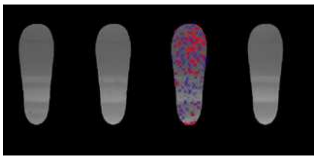
- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image



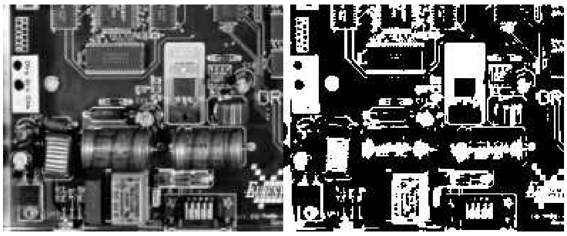
Associated mask



Processed masked image

A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.

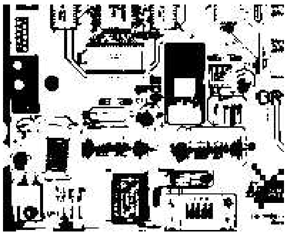
Flexible Masks in EasyImage



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

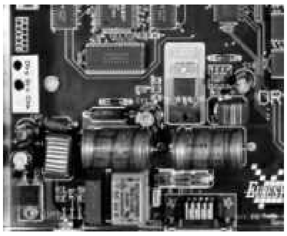
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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.

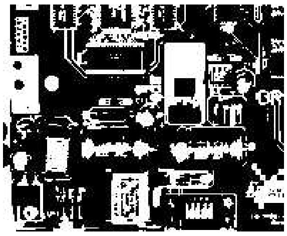
EasyObject functions that create flexible masks



Source image

1) ECodedImage2::RenderMask: from a layer of an encoded image

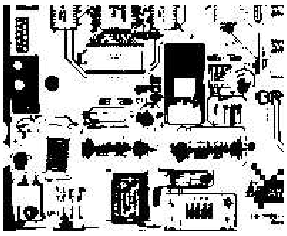
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers 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

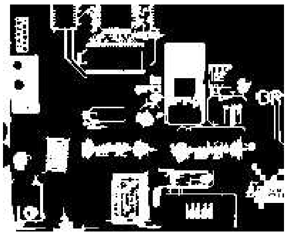
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::.RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

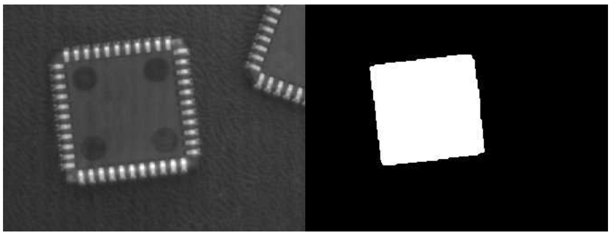
3) EObjectSelection::.RenderMask: from a selection of blobs

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



BW8 resulting image that can be used as a flexible mask

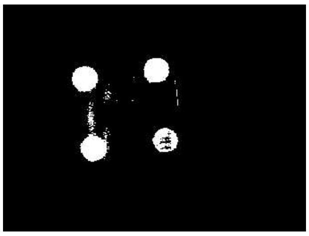
Example: Restrict the areas encoded by EasyObject



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. Multicore Processing

Multicore processing support in Open eVision

Since release 2.7, Open eVision supports the multicore processing and some algorithms are optimized to take advantage of modern multicore CPUs.

- By default, parallel processing is disabled.
- To enable parallel processing in your current thread:
  - □ Use `Easy::SetMaxNumberOfProcessingThreads()` with a value greater than 1.
  - □ Set the number of threads up to the number of physical CPU cores available in your system (without including hyper-threading).
  - □ Of course, you can use less threads than the maximum possible to preserve some of your CPU power for other processes.

> **NOTE**
> `Easy::SetMaxNumberOfProcessingThreads()` only sets the maximum number of processing threads for the thread in which the function is called.

Multiprocessor-enabled features

Currently, only some features of Open eVision are multiprocessor-enabled.

These methods as well as the speed improvements that you can expect are:

Library	Method	Improvement (per additional processor)
EasyMatrixCode 2	`EMatrixCodeReader::Read`	50%
EasyImage	Threshold on ERegion	75%
	Statistics on ERegion	75%

Library	Method	Improvement (per additional processor)
Easy3D	EPointCloudToZMapConverter::Convert	50%
	EDepthMapToPointCloudConverter::Convert	30%
	EDepthMapToMeshConverter::Convert	30%
	EZMapToPointCloudConverter::Convert	30%
	ELaserLineExtractor::ExtractProfileFromFrame	30%
EasyClassify	EClassifier::Classify	2%
EasySegment	EUnsupervisedSegmenter::Apply	4%
Regions	ERegion::ToImage	
	ERegion::Union	
	ERegion::Intersection	
	ERegion::Subtraction	

4. EGrabberBridge - Using Images from Coaxlink

See also:code snippets: EGrabberBridge

EGrabberBridge and EGrabber

EGrabberBridge is a user-friendly namespace of conversion classes. These classes perform the direct conversion from a buffer originating from **EGrabber**, the API of Euresys **Coaxlink** frame grabbers, to an Open eVision data container.

See EGrabber documentation for more information about the **EGrabber** library.

Prerequisites and libraries

- Prerequisites:
  - **Coaxlink** 11.0.3 (or newer)
  - **Open eVision** 2.9 (or newer)
- To use **EGrabberBridge** in C++, include the main **EGrabber** headers before the Open eVision header.

```
#include "EGrabber.h"
#include "FormatConverter.h"
#include "Open_eVision_X_Y.h"
```

- To use **EGrabberBridge** in C#, reference the `Coaxlink_NetApi.dll` in addition to the Open eVision .NET assembly.

Data containers

- **EGrabberBridge** is part of the main Open eVision header.
- The `FormatConverter` header is required only if you need to perform pixel format conversion (see code snippets).
- Each **EGrabberBridge** class derives from a specific Open eVision data container.

  The following classes are implemented:

Base class	EGrabberBridge class	Corresponding GenAPI pixel format
DepthMap16	EGrabberBridge::EGrabberDepthMap16	Coord3D_C16
DepthMap8	EGrabberBridge::EGrabberDepthMap8	Coord3D_C8

Base class	EGrabberBridge class	Corresponding GenAPI pixel format
EImageBW16	EGrabberBridge::EGrabberImageBW16	Mono16
EImageBW8	EGrabberBridge::EGrabberImageBW8	Mono8
EImageC24	EGrabberBridge::EGrabberImageC24	BGR8

- These classes have 2 constructors:
  - □ 1 that requires only an **EGrabber** buffer descriptor (see EGrabber reference).
  - □ 1 that requires an **EGrabber** buffer descriptor and an additional FormatConverter parameter used to perform the conversion from the pixel format of the buffer to the pixel format of your **EGrabberBridge** class, if these are different (see Using EGrabberBridge with Format Conversion code snippet).
- Scope and copy of the buffer:
  - □ Open eVision does not perform any copy of the buffer unless you require a pixel format conversion.
  - □ The availability of the **EGrabberBridge** data container buffer depends on the **EGrabber** buffer object or on the FormatConverter object if a copy is performed.

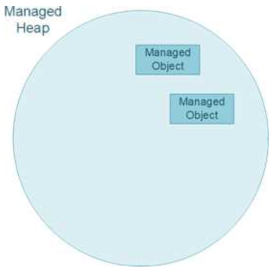
Examples and code snippets

- A sample (Using EGrabberBridge) illustrates the use of the **EGrabberBridge** classes using callbacks. The sample is available in C++ and in C# and is present in its corresponding samples solution under the EGrabberBridge name.
- A code snippet (Using EGrabberBridge with Format Conversion) is available to show how to use **EGrabberBridge** to perform the inversion of an image acquired using **EGrabber**.
- Camera and **GenICam** parameters can be handled through the **EGrabber** Object setters and getters (see EGrabber documentation).
- You can also test out parameters using the **GenICam** application (see GenICam documentation).

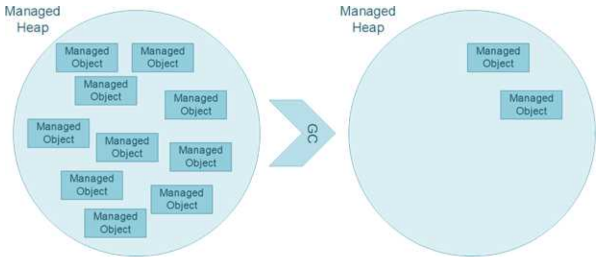
5. Handling the Memory in .NET

The memory management in .NET

- .NET keeps its objects in a special part of the memory called the *managed heap*.

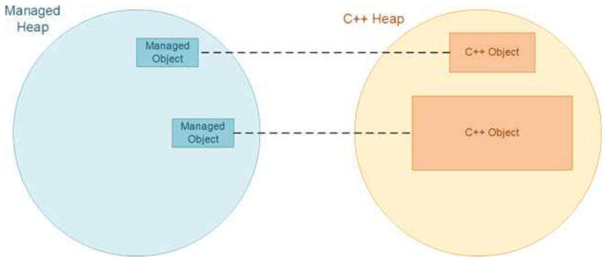


- If that managed heap becomes too full, a specialized process called the *garbage collector* (*GC*) starts. This process cleans and removes from the memory the objects that are no longer in use.

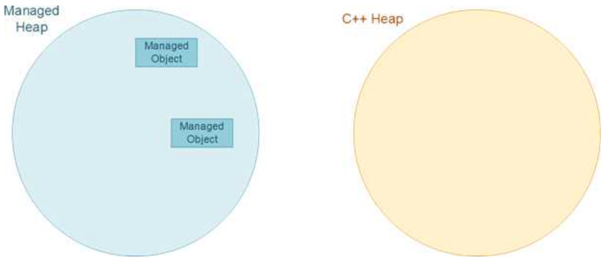


Open eVision specificity

- The managed objects of Open eVision are linked to C++ objects that are stored outside of the managed memory space, in the heap memory of the DLL.



- As the .NET side of Open eVision objects is significantly smaller than their C++ side:
  - □ The memory usage is usually (much) bigger than the GC computation.
  - □ The GC is unable to free memory when needed.
  - □ So, it is important, in a .NET application using Open eVision managed objects, to call `.Dispose()` on them when you do not need them anymore.
    This frees the C++ memory and allows the GC to work efficiently.



When to call `.Dispose()`

Call `.Dispose()` as soon as you don't need the object anymore.

Simple case

```
// Create finder
EPatterFinder finder = new EPatternFinder()
...
// Use finder
finder.Find(image);
...
// Finder has done its job, dispose
finder.Dispose();
```

TIP
However, be aware of a few tricky cases, especially on temporary values and nested classes as illustrated below.

Temporary values

```
// Call EasyMatrixCode2 Reader
mxc2Reader.Read(image);

// Get the first MXC decoded string
string decodedString = mxc2Reader.ReadResults[0].DecodedString;

// Cleanup
mxc2Reader.Dispose();
```

- Regarding the above code example:
  - □ It seems pretty straightforward but `mxc2Reader.ReadResults[0]` returns a temporary `EMatrixCode` object that you should dispose too.
  - □ If called a few times, it doesn't pose any real problem.
  - □ If called a few thousand times, it can lead to memory issues.

- The correct safe code is:

```
// Call EasyMatrixCode2 Reader
mxc2Reader.Read(image);

// Get the first MXC decoded string
EMatrixCode code = mxc2Reader.ReadResults[0];
string decodedString = code.DecodedString;

// Cleanup
code.Dispose();
mxc2Reader.Dispose();
```

Nested objects

Some of the Open eVision objects have other objects nested inside them. In that case, when you use the nested object, it's important to dispose it before its host.

- As an example, lets take the case of the `EImageEncoder` class of **EasyObject2**:

```
// Set the segmentation method to GrayscaleDoubleThreshold
encoder.SegmentationMethod= ESegmentationMethod.GrayscaleDoubleThreshold;

// Configure the segmenter object
encoder.GrayscaleDoubleThresholdSegmenter.HighThreshold = 150;
encoder.GrayscaleDoubleThresholdSegmenter.LowThreshold = 50;

// Cleanup
encoder.Dispose();
```

  - `GrayscaleDoubleThresholdSegmenter` is a segmenter object nested inside the `EImageEncoder` class.
  - Accessing it with the `encoder.GrayscaleDoubleThresholdSegmenter.HighThreshold = 150` and the `...LowThreshold = 50` lines create a wrapper around that segmenter that you should dispose to avoid any memory issue.

- The correct safe code is:

```
// Set the segmentation method to GrayscaleDoubleThreshold
encoder.SegmentationMethod= ESegmentationMethod.GrayscaleDoubleThreshold;

// Retrieve the segmenter object
EGrayscaleDoubleThresholdSegmenter segmenter =
encoder.GrayscaleDoubleThresholdSegmenter;

// Configure the segmenter
segmenter.HighThreshold = 150;
segmenter.LowThreshold = 50;

// Cleanup
segmenter.Dispose();
encoder.Dispose();
```

NOTE

Always dispose of a nested object before its host object to avoid crashes of your application.

Good practice: use the `using` statement

An elegant way to manage the lifetime of a variable and to ensure that the disposal is done correctly is to code with the `using` statement.

- Instead of writing your code as follows:

```
EMatrixCodeReader reader = new EMatrixCodeReader();
...
reader.Dispose();
```

- Use the `using` statement to ensure that `.Dispose()` is automatically called on `reader` when the statement closes.

```
using (EMatrixCodeReader reader = new EMatrixCodeReader())
{
    ...
}
```

Function scope and limitation

```
void function()
{
  EMatrixCodeReader reader = new EMatrixCodeReader();
  ...
}
```

- In the above code, when the variable is released at the end of the function scope, you can think that it is not necessary to call `.Dispose()` on a corresponding object.

  But the object is not automatically disposed at the end of the function, even if the variable itself does not exist anymore.

  In practice, at the end of the function:

  - The object is released.
  - The GC can dispose of the object, but this is not immediate, as the GC only frees memory when needed.
  - Since it may take some time, if the C++ memory is already quite full, it may not be efficient enough.

- The correct safe code is:

```
void function()
{
  EMatrixCodeReader reader = new EMatrixCodeReader();
  ...
  reader.Dispose();
}
```

Disposing function arguments

In .NET, the Open eVision objects are always passed as references, without copy. This means that the object inside the function is the same one that has been passed to the function in the calling code.

```
void UseImage(EImageBW8 imageInFunction)
{
  imageInFunction.SetSize(128, 128);
  ...
}

void MainFunction()
{
  EImageBW8 imageOutOfFunction = new EImageBW8();
  UseImage(imageOutOfFunction);
}
```

- In the above example code:
  - □ `imageInFunction` and `imageOutOfFunction` are in fact the same image.
  - □ When you call `.Dispose()` on one image, both are disposed.
  - □ Call `.Dispose()` on a function argument, but only when you don't need the object anymore, neither inside nor outside the function.

> **TIP**
> To have a better view of an object lifetime, it is recommended, whenever possible, to dispose the objects in the same scope as their creation.

- The correct safe code is:

```
void UseImage(EImageBW8 imageInFunction)
{
  imageInFunction.SetSize(128, 128);
  ...
}

void MainFunction()
{
  EImageBW8 imageOutOfFunction = new EImageBW8();
  UseImage(imageOutOfFunction);
  imageOutOfFunction.Dispose();
}
```

Good practice: set the variables to `null`

- Set a variable to `null` after disposing of its object.
    - □ It is not mandatory but it is considered a good practice.
    - □ It unlinks the reference to the object, and so it informs the GC that the object is not used anymore.
    - □ The GC is more susceptible to cleanup the object the next time it runs.
- Example:

```
// Cleanup
code.Dispose();
code = null;
```

PART II

LIBRARIES AND TOOLS

PART III

GENERAL PURPOSE LIBRARIES

0.1. EasyImage - Pre-Processing Image

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

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

The main functions are:

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

Intensity Transformation

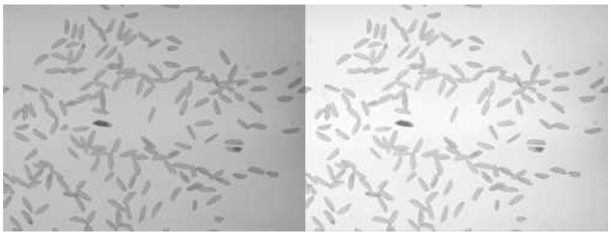
These EasyImage functions change the gray-levels of pixels to increase contrast.

Gain offset

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

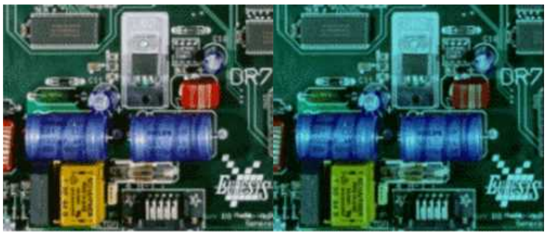
- **gain** adjusts **contrast**. It should remain close to 1.
- **offset** adjusts **intensity** (brightness). It can be positive or negative.
- The resulting values are always saturated to range [0..255].

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



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

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

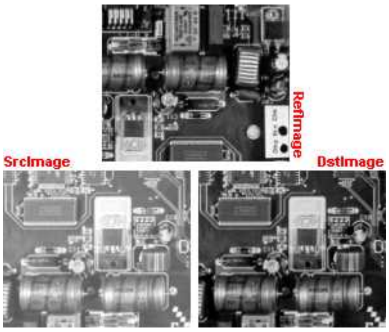


Example of gain/offset applied on a color image

Normalization

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

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

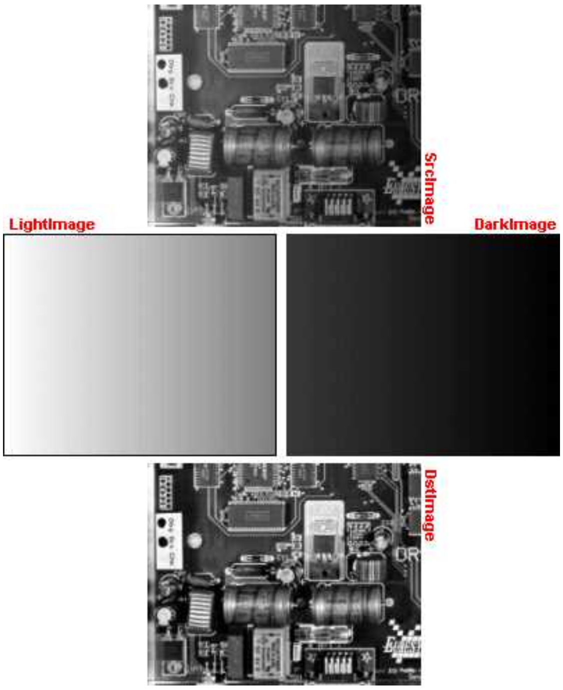


The reference image (from which the average and standard deviation are computed),
the source image (too bright),
and the normalized image (contrast and brightness are the same as the reference image)

Uniformization

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

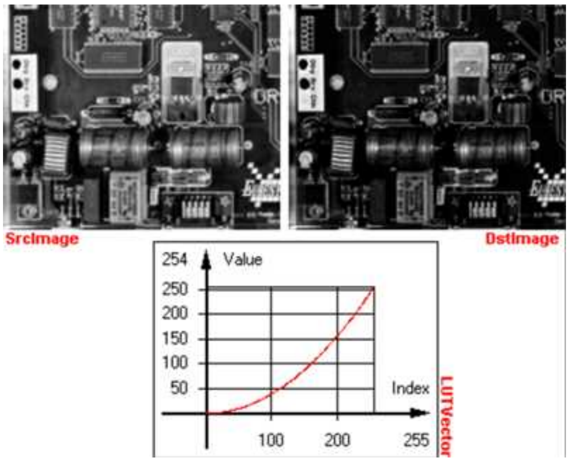
- When one reference image is used, the transformation is similar to an adaptive (space-variant) gain; each pixel in the reference image encodes the gain for the corresponding pixel in the source image.
- When two reference images are used, the transformation is similar to an adaptive gain and offset;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



Example of an image uniformized with two reference images

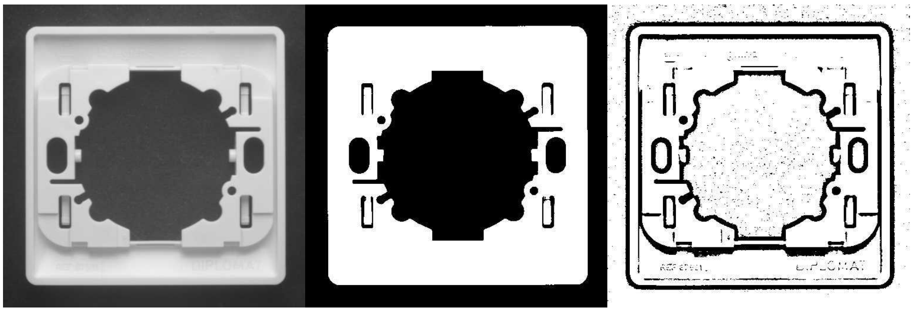
Lookup tables

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



Example of a transform

Thresholding



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

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

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

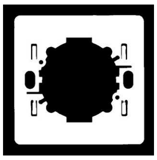
Keys to successful thresholding

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

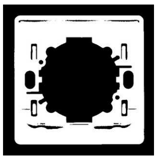
Automatic thresholding

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

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



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



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

Manual thresholding

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

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

Histogram based

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

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

AutoThreshold

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

This function supports flexible masks.

Arithmetic and Logic

Reasons you may use arithmetic and logic are:

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

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

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

arithmetic and logic combinations

Allowed combinations

	General	Copy	Invert	Shift	Logical	Overlay	Set
Const BW8 -> Image BW8		x					
Const C24 -> Image C24		x					
Image BW8 -> Image BW8		x	x				
Image BW8 -> Image C24		x	x			x	
Image C24 -> Image C24		x	x				
Const BW8, Image BW8 -> Image BW8	x						
Image BW8, Const BW8 -> Image BW8	x			x			x
Image BW8, Image BW8 -> Image BW8	x				x		x
Image BW8, Image BW8 -> Image C24	x					x	
Const C24, Image C24 -> Image C24	x						
Image C24, Const C24 -> Image C24	x			x			
Image C24, Image C24 -> Image C24	x				x		
Image C24, Image BW8 -> Image C24	x				x	x	
Image BW8, Image C24 -> Image C24	x				x		x

NOTE

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

The classification of operations in the above table are:

General

- Compare (abs. value of the difference)
- Saturated sum
- Saturated difference
- Saturated product
- Saturated quotient
- Modulo
- Overflow-free sum
- Overflow-free difference
- Overflow-free product
- Overflow-free quotient
- Bitwise AND
- Bitwise OR
- Bitwise XOR
- Minimum
- Maximum
- Equal
- Not equal
- Greater or equal
- Lesser or equal
- Greater
- Lesser

Copy

- Sheer Copy

Invert

- Invert (negative)

Shift

- Left Shift
- Right Shift

Logical

- Logical AND
- Logical OR
- Logical XOR

Overlay

- Add an overlay

Set

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

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

Non-Linear Filtering

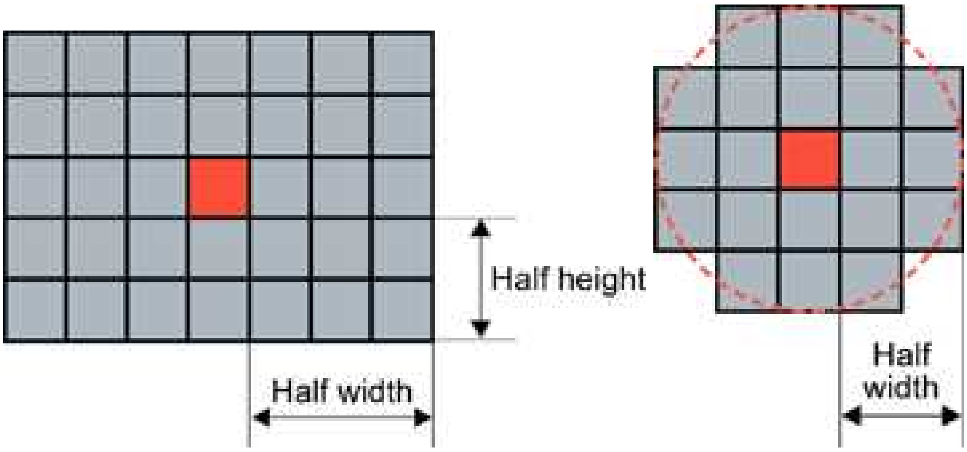
These functions use non-linear combinations of neighboring pixels to highlight a shape, or to remove noise.

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

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

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

Kernel



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

The morphological operators combine the pixel values in a neighborhood of given shape (square, rectangular or circular) and replace the central pixel of the neighborhood by the result.The combining function is non-linear, and in most cases is a rank filter: which considers the N values in the given neighborhood, sorts them increasingly and selects the K-th largest.
Three special cases are most often used erosion, dilation *and* **median filter** *where : K can be 1 (minimum of the set), N (maximum) or N/2 (median).*

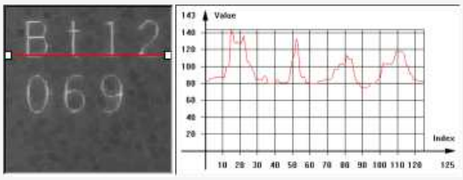
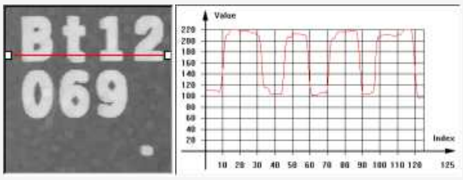
Erosion, Dilation, Opening, Closing, Top-Hat and Morphological Gradient operations all use rectangular or circular kernels of odd size. Kernel size has an important impact on the result.

examples

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

Erosion, Dilation

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

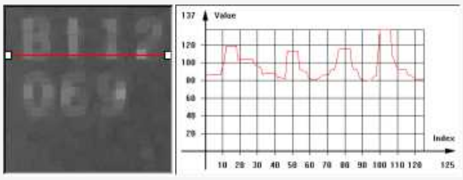
	
Erosion	Dilation

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

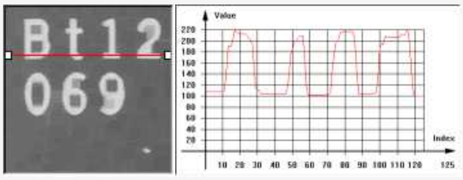
Dilation thickens white objects by adding a layer of pixels along the objects edges: DilateBox, DilateDisk. As the kernel size increases, white objects get fatter and black ones disappear.

Opening, Closing

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



Opening



Closing

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

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

Thinning, Thickening

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

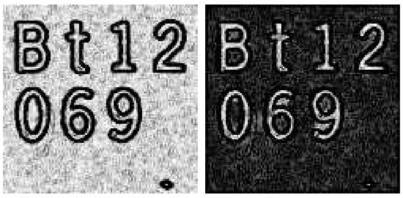
- Thinning: can help edge detectors by reducing lines to single pixel thickness.
- Thickening: can help determine approximate shape, or skeleton.

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

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

Top-Hat filters

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



White top-hat filter: source and destination images

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

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

Morphological distance

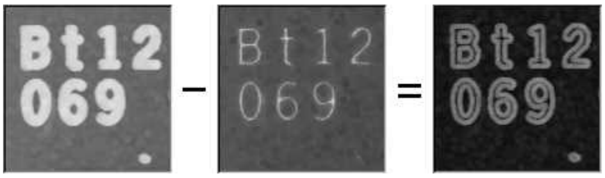
Distance computes the morphological distance (number of erosion passes to set a pixel to black) of a binary image (0 for black, non 0 for white) and creates a destination image, where each pixel contains the morphological distance of the corresponding pixel in the source image.

Morphological gradient

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

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

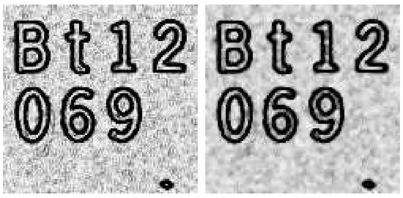
MorphoGradientBox, MorphoGradientDisk.



Dilation – Erosion = Gradient

Median

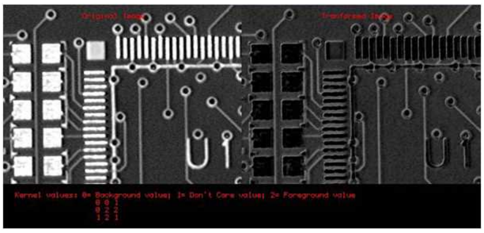
The Median filter removes impulse noise, whilst preserving edges and image sharpness.
It replaces every pixel by the median (central value) of its neighbors in a 3x3 square kernel, thus, outer pixels are discarded.



Median filter: source and destination images

Hit-and-Miss transform

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


Original Image
Transformed Image
Kernel values: 0= Background value; 1= Don't Care value; 2= Foreground value
0 0 1
0 2 2
1 2 1

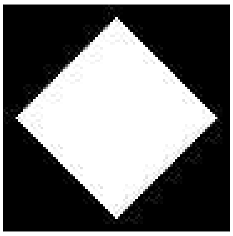
Hit-and-miss transform

The `HitAndMiss` function has three arguments:

- A pointer to the source image of type `EROIBW8`, `EROIBW16`, or `EROIC24`
- A pointer to the destination image of type corresponding to the type of the source image. The sizes of the source and destination images must be identical.
- A kernel of type `EHitAndMissKernel` Two constructors are available for the kernel object:
  - `EHitAndMissKernel(int startX, int startY, int endX, int endY)` where:
    `startX`, `startY` are coordinates of the top left of the kernel, must be less than or equal to zero.
    `endX`, `endY` are coordinates of the bottom right of the kernel, must be greater than or equal to zero.
    The constructed kernel has no explicit restrictions on its size, and the following characteristics:
    `kernel width = (endX – startX + 1)`, `kernel height = (endY – startY + 1)`
  - `EHitAndMissKernel(unsigned int halfSizeX, unsigned int halfSizeY)` where:
    `halfSizeX` is half of the kernel width – 1, must be greater than zero.
    `halfSizeY` is half of the kernel height – 1, must be greater than zero.
    The constructed kernel has the following characteristics:
    `kernel width = ((2 x halfSizeX) + 1)`, `kernel height = ((2 x halfSizeY) + 1)`
    `kernel StartX = - halfSizeX`, `kernel StartY = - halfSizeY`

example: detecting corners in a binary image.

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

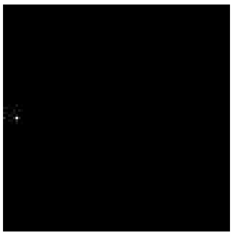


Binary source image

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

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

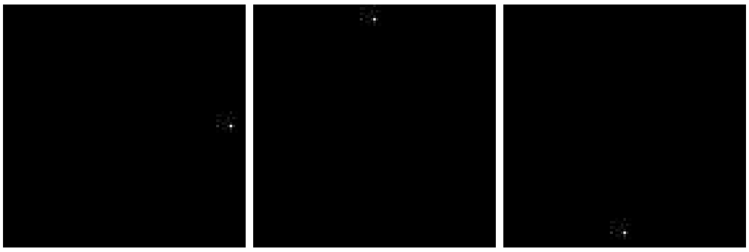
2. Apply the filter on the source image. Note that the resulting image should be properly sized.



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

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

4. Detect the right, top and bottom corners.



Geometric Transforms

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

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

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

Possible geometric transformations are:

ReAlignment

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

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

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

Mirroring

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

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

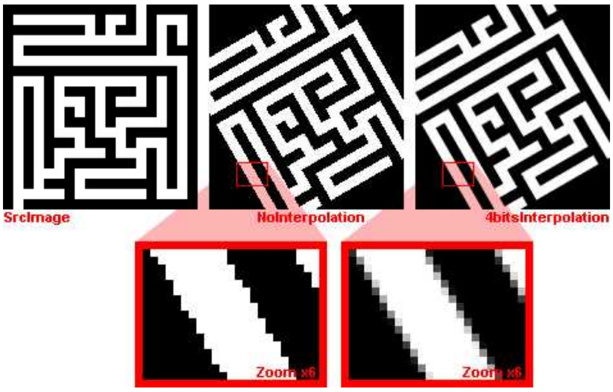
Translation, Scaling and Rotation

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

`EasyImage::.ScaleRotate` performs:

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

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



Scale and rotate example

Shrink

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

LUT-based unwarping

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

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

Noise Reduction and Estimation

Noise can degrade the visual quality of images, and certain processing operations (thresholding, high-pass filtering) will enhance noise in a non-acceptable way. Acquired images are always noisy (this is best observed on live images where the pixel values fluctuate around the true intensity). When acquired with 8 bits of accuracy, the noise level typically amounts to about 3 to 5 gray-level values. One distinguishes several forms of noise:

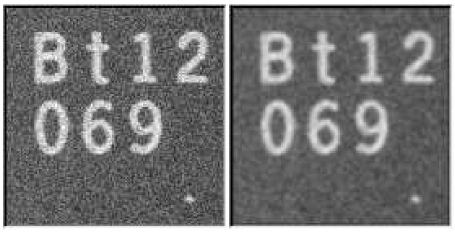
- **additive**: noise amplitude is not related to image contents
- **multiplicative**: noise amplitude is proportional to local intensity
- **uniform**: noise amplitude follows a smooth distribution centered around zero
- **impulse**: noise amplitude is infinite.

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

Spatial noise reduction (if you only have 1 image):

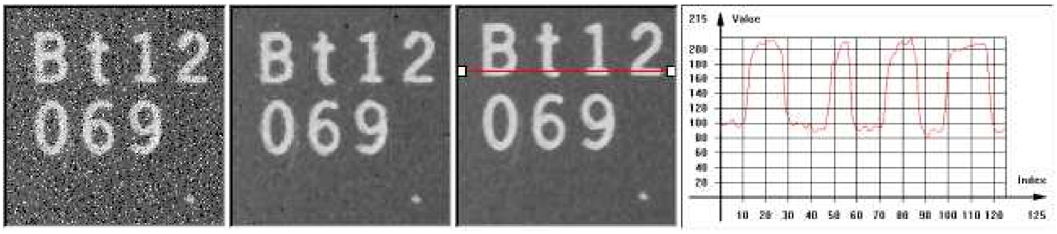
Reduces uniform and impulse noise but blurs edges.
Cannot distinguish noise from actual signal changes, so always spoils part of the signal.
Uses the correlation between neighboring pixel values to perform convolution or median filtering:

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



Uniform noise reduction by low-pass filtering

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



Impulse noise reduction by median filtering

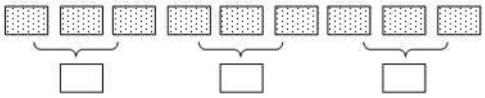
- EasyImage::.Median
- EasyImage::.BiLevelMedian

Temporal noise reduction (for several images, e.g. moving objects):

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

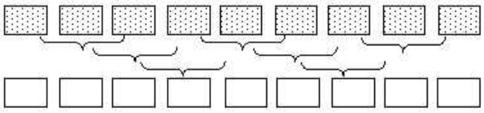
EasyImage provides three ways to minimize noise by means of several images:

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



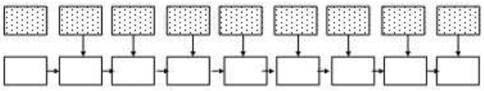
Simple average

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



Moving average

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



Recursive average

Recursive averaging

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

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

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

This procedure is effective when applied to still images, but generates a trailing effect on moving objects. The larger the noise reduction ratio, the heavier the trailing effect is. To work around this, a non-linearity can be introduced in the blending process: small gray-level value variations between successive images are usually noise, while large variations correspond to changes in the image.

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

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

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

Noise estimation (of image compared to reference image):

To estimate the amount of noise, two or more successive images are required. In the simplest mode, two noisy images are compared. (Other modes are available: if a noise-free image is available, it is compared to a noisy one; a noise-free image can also be built by temporal averaging.) Calculates the root-mean-square amplitude and signal-to-noise ratio.

- `EasyImage::.RmsNoise` computes the root-mean-square amplitude of noise, by comparing a given image to a reference image. This function supports flexible mask and an input mask argument. BW8, BW16 and C24 source images are supported.
  The reference image can be noiseless (obtained by suppressing the source of noise), or affected by a noise of the same distribution as the given image.

- EasyImage::.SignalNoiseRatio computes the signal to noise ratio, in dB, by comparing a given image to a reference image. This function supports flexible mask and an input mask argument. BW8, BW16 and C24 source images are supported.
  The reference image can be noiseless (obtained by suppressing the source of noise) or be affected by a noise of the same distribution as the given image.

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

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

Scalar Gradient

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

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

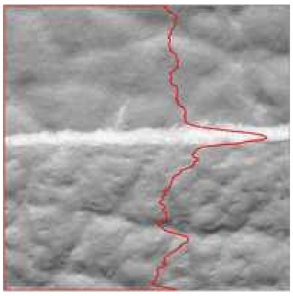
The gradient of a grayscale image corresponds to a vector, the components of which are the partial derivatives of the gray-level signal in the horizontal and vertical direction. A vector can be characterized by a direction and a length, corresponding to the gradient orientation, and the gradient magnitude.

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

Vector Operations

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

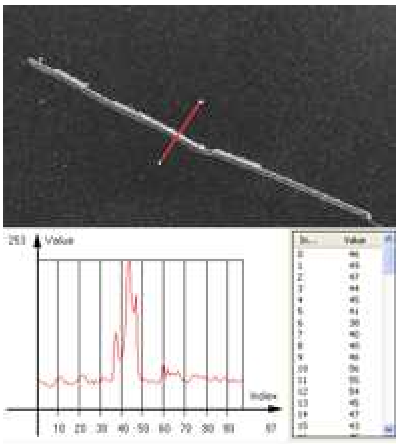
Projection



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

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

Profile



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

1. Obtain the profile of a line segment /path /contour.

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

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

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

2. Analyse the profile to find peaks or transitions.

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

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

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

3. Insert the profile into an image.

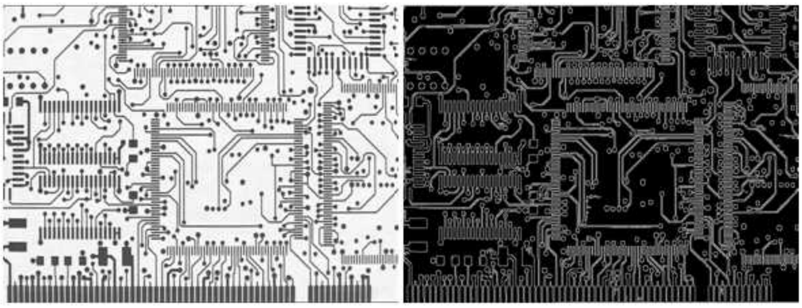
   EasyImage::.LineSegmentToImage copies the pixel values from a vector or a constant to the pixels of a given line segment (arbitrarily oriented). The line segment must be wholly contained within the image.

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

Canny Edge Detector

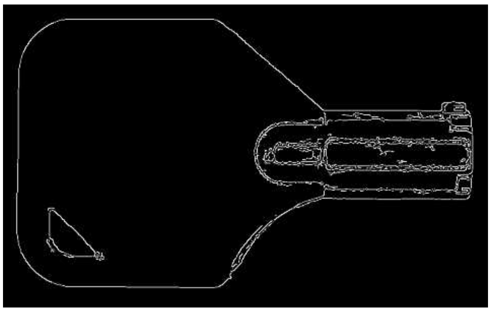
The Canny edge detector facilitates:

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



Source image and the result after a Canny edge detection

The EasyImage Canny edge detector operates on a grayscale BW8 image and delivers a black-and-white BW8 image where pixels have only 2 possible values: **0** and **255**. Pixels corresponding to edges in the source image are set to **255**; all others are set to **0**. It can adjust the scale analysis, it doesn't allow sub-pixel interpolation and it delivers a binary image after thresholding.



Canny edge detector example

The Canny edge detector requires only two parameters:

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

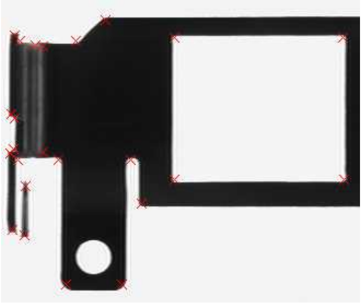
The API of the Canny edge detector is a single class, `ECannyEdgeDetector`, with the following methods:

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

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

Harris Corner Detector

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



Harris corner detector example

The EasyImage Harris corner detector requires three parameters:

- The integration scale σi: the standard deviation of the Gaussian Filter used for scale analysis. σd = 0,7 x σi, where σd is the differentiation scale: the standard deviation of the Gaussian Filter used for noise reduction during computation of the gradient.

- A corner threshold: a fraction ranging from 0 to 1 of the maximum value of the cornerness of the source image.
- A Boolean that toggles sub-pixel detection.

The following characteristics are available for every point of interest:

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

The API of the Harris corner detector is a single class named `EHarrisCornerDetector` and these methods:

- `Apply`: applies the Harris corner detector on an image/ROI.
- `EHarrisCornerDetector`: constructs a `EHarrisCornerDetector` object initialized to its default values.
- `GetDerivationScale`: returns the current derivation scale.
- `GetScale`: returns the integration scale.
- `GetThreshold`: returns the current threshold.
- `GetThresholdingMode`: returns the current thresholding mode for the cornerness measure.
- `IsGradientNormalizationEnabled`: returns whether the gradient is normalized before the computation of the cornerness measure.
- `IsSubpixelPrecisionEnabled`: returns whether the sub-pixel interpolation is enabled.
- `SetDerivationScale`: sets the derivation scale.
- `SetGradientNormalizationEnabled`: sets whether the gradient is normalized before the computation of the cornerness measure.
- `SetScale`: sets the integration scale.
- `SetSubpixelPrecisionEnabled`: sets whether the sub-pixel interpolation is enabled.
- `SetThreshold`: sets the threshold on the cornerness measure for a pixel to be considered as a corner.
- `SetThresholdingMode`: sets the thresholding mode for the cornerness measure.

Basic usage of Harris Corner Detector

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

1. **Create an instance of the detector** and set the appropriate method, for instance, the integration scale, `SetScale`, with the structures of interest that could have a spatial extent of 2 pixels.
2. `Apply` **the detector** with two arguments to the new image : the input image and the interest points in the input image `EHarrisInterestPoints`.
3. Access the individual elements of the output vector.

Overlay

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

If a color image is provided as the source image, all the pixels of this image are copied to the destination image, except the ones that equal the reference color. When a C24 image is used as overlay source image, the color of the overlay in destination image is the same as the one in the overlay source image, thus allowing multicolored overlays.

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

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

Operations on Interlaced Video Frames

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

EasyImage::.RealignFrame cures this problem if the movement is uniform and horizontal (objects on a conveyor belt), by shifting one of the frames horizontally. The amplitude of the shift can be estimated automatically.

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

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

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

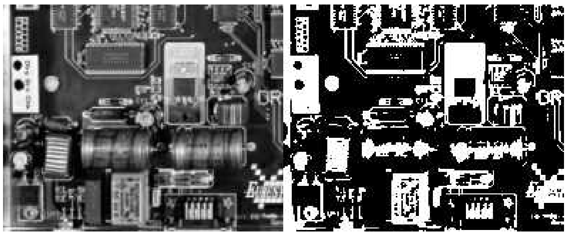
EasyImage::.SwapFrames: interchanges the even and odd rows of an image. This is helpful when acquisition of an interlaced image has confused even and odd frames.

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

The size of the destination image is determined as follows:

$$dstImage\_Width = srcImage\_Width$$
$$dstImage\_Height = (srcImage\_Height + 1 - odd) / 2$$

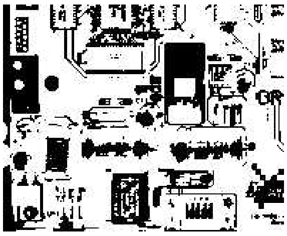
Flexible Masks in EasyImage



Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

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



Resulting image

EasyImage Functions that support flexible masks

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

Computing Image Statistics

EasyObject statistics are related to the objects in an image.

EasyImage statistics are related to whole images (global illumination / contrast, saturation, presence or absence of an object).

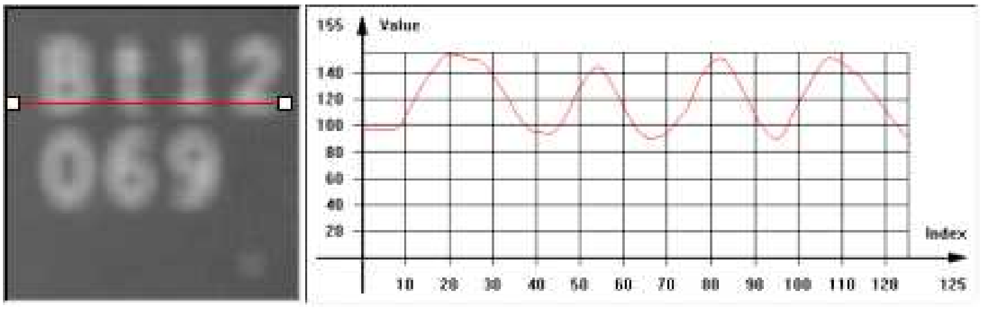
Sliding window (creates new image of avg or std deviation of gray-level values)

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

> **NOTE**
> The computing time of these functions does not depend on the window size.

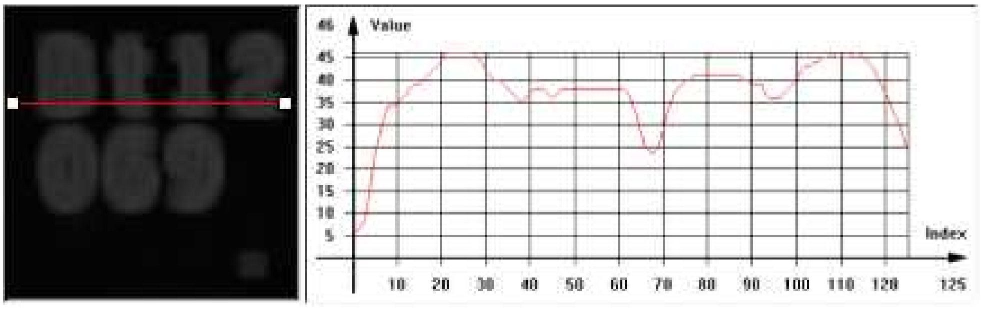
The result of the operation is another image.

The **local average**, `EasyImage::.LocalAverage`, corresponds to a strong low-pass filtering.



Sliding window average

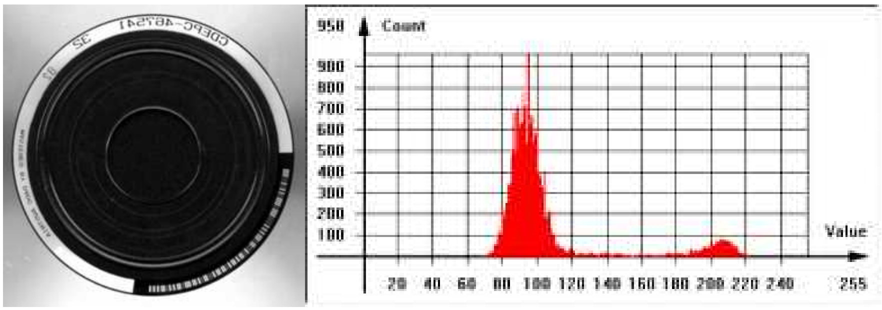
The **local standard deviation**, `EasyImage::.LocalDeviation` enhances the regions with a high frequency contents, such as noisy or textured areas.



Sliding window standard deviation

Histogram computation and analysis(and LUT creation)

A histogram is a statistical summary of an image: it shows the number of occurrences of every gray-level value in an image, and it's shape reveals characteristics of the image. For instance, peaks in the histogram curve correspond to dominant colors in the image. If the histogram is bi-modal, a large peak for the dark values corresponding to the background, and smaller peaks in the light values.



Typical image histogram

Histogram Computation

EasyImage::.Histogram computes the histogram of an image. It has an input mask argument. It supports flexible mask.
BW8, BW16 and BW32 source images are supported.

You can compute the cumulative histogram of an image, i.e. the count of pixels below a given threshold value, by calling EasyImage::.CumulateHistogram after EasyImage::.Histogram.

Histogram Analysis

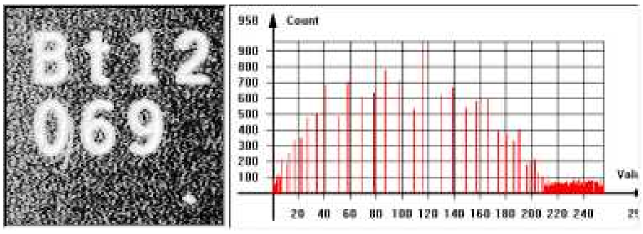
EasyImage::.AnalyseHistogram and EasyImage::.AnalyseHistogramBW16 provide statistics and thresholding values:

- Total number of pixels.
- Smallest and largest pixel value (gray-level range).
- Average and standard deviation of the pixel values.
- Value and frequency of the most frequent pixel.
- Value and frequency of the least frequent pixel.

Histogram equalization

EasyImage::.Equalize re-maps the gray levels so that the histogram fills in the whole dynamic range as uniformly as possible.

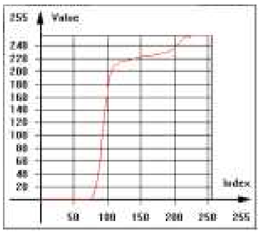
This may be useful to maximize image contrast, or reveal a lot of image details in dark areas.



Equalized image and histogram

Setup a lookup table

`EasyImage::.SetupEqualize` creates a LUT so you can work explicitly with the histogram and LUT vectors. It can be more efficient to keep the image histogram for other purposes (i.e statistics) and keep the equalization LUT to apply to other images.



Equalization lookup table

Image focus

Sharp focusing can be achieved if the `EasyImage::.Focusing` quantity is maximum for a given image. This function must be called multiple times with multiple images with a different focus for the basis of an "auto-focus" system.

`EasyImage::.Focusing` computes the total gradient energy of the image. You can then use this gradient as a measure of the focusing of an image.

The gradients of the image show the edges of the structures present in the image, with strong values if the image is well-focused and weaker values otherwise.

To compute the total gradient energy of the image, Open eVision:

a. Squares the pixel values of the horizontal and vertical gradient images.

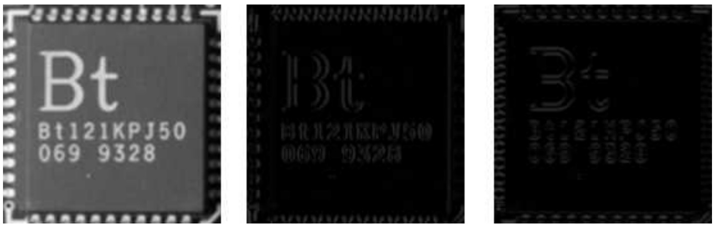
b. Averages the squared pixel values over both images.

c. Sums the averages.

d. Takes the square root of the resulting value.

> ✓ **TIP**
> The resulting value is maximum if the image is well-focused.

A well-focused image, with its (absolute-valued) horizontal and vertical gradients. The gradients show the edges of the structures with strong values. The total gradient energy for this image is 17.9.



A badly focused image, with its (absolute-valued) horizontal and vertical gradients. The gradients show the edges of the structures with weak values. The total gradient energy for this image is 7.9.

EasyImage statistics functions

Area (number of pixels with values above/on/between thresholds)

- `EasyImage::.Area` counts pixels with values above (or on) a threshold.
- `EasyImage::.AreaDoubleThreshold` counts pixels whose values are comprised between (or on) two thresholds.

Binary and weighted moments (object position and extent)

- `EasyImage::.BinaryMoments` computes the 0th, 1st or 2nd order moments on a binarized image, i.e. with a unit weight for those pixels with a value above or equal to the threshold, and zero otherwise. It provides information such as object position and extent.
- `EasyImage::.WeightedMoments` computes the 0th, 1st, 2nd, 3rd or 4th order weighted moments on a gray-level image. The weight of a pixel is its gray-level value. It provides information such as object position and extent.

Gravity center (average pixel coordinates above/on threshold)

- `EasyImage::.GravityCenter` computes the coordinates of the gravity center of an image, i.e. the average coordinates of the pixels above (or on) the threshold.

Pixel count (between 2 thresholds)

- `EasyImage::.PixelCount` counts the pixels in the three value classes separated by two thresholds.

Minimum, maximum and average gray-level value

- `EasyImage::.PixelMax` computes the maximum gray-level value in an image.
- `EasyImage::.PixelMin` computes the minimum gray-level value in an image.
- `EasyImage::.PixelAverage` computes the average pixel value in a gray-level or color image. For a color image, it computes the means of the three pixel color components, the variances of the components and the covariances between pairs of components.

Average, variance and standard deviation

- `EasyImage::.PixelStat` computes min, max and average gray-level values.

- `EasyImage::.PixelVariance` computes average and variance of pixel values.
- `EasyImage::.PixelStdDev` computes average and standard deviation of pixel values. For a color image, it computes the standard deviations and correlation coefficients (covariance over the product of standard deviations) of the pairs of pixel component values.

Number of different pixels by comparing 2 images

- `EasyImage::.PixelCompare` counts the number of different pixels between two images.

0.2. EasyColor - Pre-processing Color Images

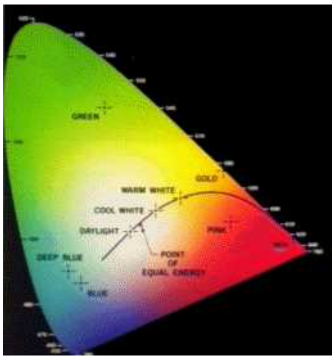
EasyColor makes color image processing as efficient as possible by detecting, classifying and analyzing objects. Several conversion functions mean that any color system can be processed.

Color definition and supported systems

What Is Color?

The human eye is sensitive to light:

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



Visible color gamut in the XYZ color space

There are three color systems:

- **Mixture** systems (RGB/XYZ) give the proportions of the three primaries to be combined.
- **YUV Luma/chroma** systems (XYZ/YUV) separate the achromatic (Y) and chromatic sensations (U & V). Used when a black and white image is required as well (television).

- **Intensity/saturation/hue** systems (RGB/XYZ/YUV) separate achromatic (black and white Intensity) from enhanced chromatic (color Saturation and Hue) sensations. Used to eliminate lighting effects, or to convert RGB images to another color system. More saturated colors are more vivid, less saturated ones are grayer.

In general:

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

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

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

Color Image Processing

A color image is a vector field with three components per pixel. All three RGB components reflected by an object have amplitude proportional to the intensity of the light source. By considering the ratio of two color components, one obtains an illumination-independent image. With a clever combination of three pieces of information per pixel, one can extract better features.

There are 3 ways to process a color image:

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

Supported color systems

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

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

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

Transform using LUTs (LookUp Tables)

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

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

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

"LUT for Gain/Offset (Color) " on page 81: `EasyImage::.GainOffset`,

"LUT for Color Calibration" on page 81: `Calibrate`,

"LUT for Color Balance" on page 82: `WhiteBalance`,

`ConvertFromRGB`, `ConvertToRGB`.

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

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

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

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

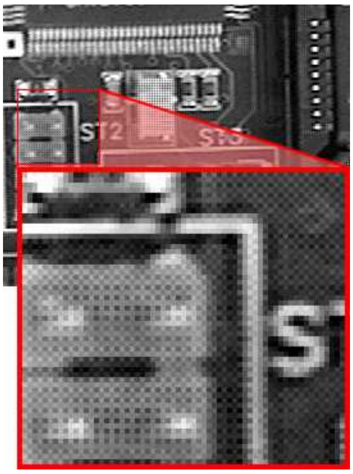
Discrete Quantized vs. Continuous Unquantized

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

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

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

Bayer transform



Bayer pattern encoded image

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

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

Transform YUV444 /YUV422

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

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

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

Merge, extract and color

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

Merge and extract components

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

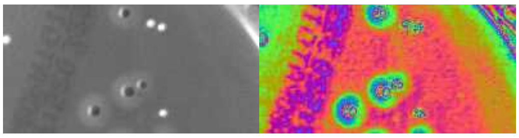
These operations can use a color LUT to transform on the fly, they could build an RGB image from lightness, saturation and hue planes.

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

Pseudo-color to transform gray-level images to color

The trick is to define a regular gamut of 256 colors and each color will be assigned to pixels with a corresponding gray-level value.

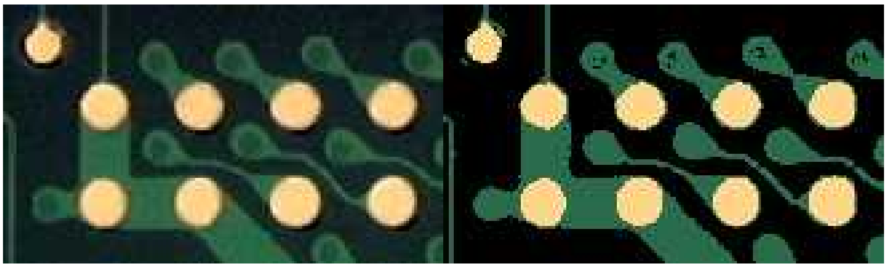
To define pseudo-color shades, you specify a trajectory in the color space of an arbitrary system. You can then pseudo-color using the drawing functions color palette (see Image and Vector Drawing) then save and/or transform it like any other color image.



Gray-level and pseudo-colored image

Separate color objects

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



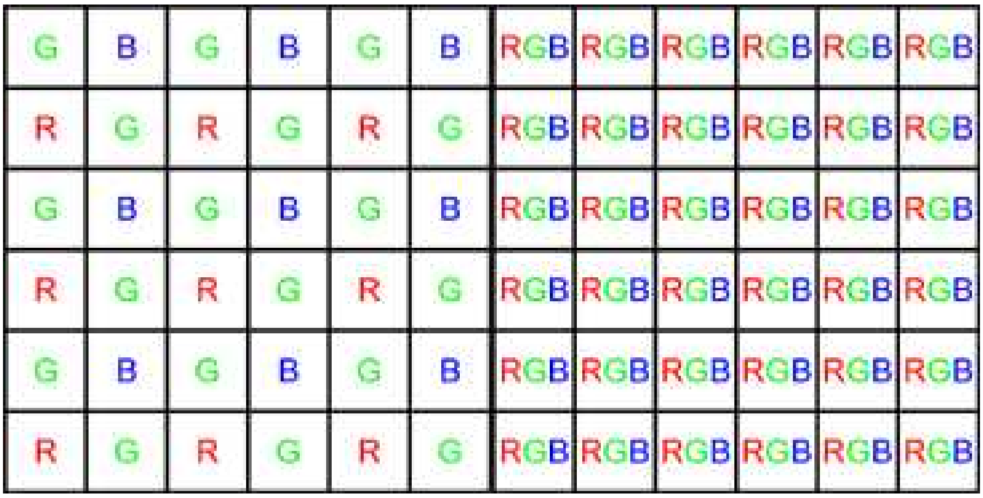
Raw image and segmented image (3 colors)

Bayer Transform

The Bayer pattern is a color image encoding format for capturing color information from a single sensor.

A color filter with a specific layout is placed in front of the sensor so that some of the pixels receive red light only, while others receive green or blue only.

An image encoded by the Bayer pattern has the same format as a gray-level image and conveys three times less information. The true horizontal and vertical resolutions are smaller than those of a true color image.



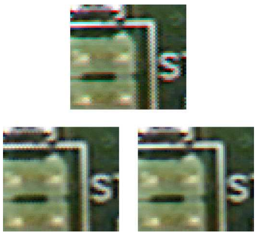
G	B	G	B	G	B	RGB	RGB	RGB	RGB	RGB	RGB
R	G	R	G	R	G	RGB	RGB	RGB	RGB	RGB	RGB
G	B	G	B	G	B	RGB	RGB	RGB	RGB	RGB	RGB
R	G	R	G	R	G	RGB	RGB	RGB	RGB	RGB	RGB
G	B	G	B	G	B	RGB	RGB	RGB	RGB	RGB	RGB
R	G	R	G	R	G	RGB	RGB	RGB	RGB	RGB	RGB

Bayer vs. true color format

NOTE
The Bayer pattern normally starts with a GB/RG block in the upper left corner. If the image is cropped, this parity rule can be lost, but parity adjustment is unnecessary when working on a Open eVision ROI.

The Bayer conversion method `EasyColor::.BayerToC24` transforms an image captured using the Bayer pattern and stored as a gray-level image, into a true color image. There are three ways to reconstruct the missing pixels. The more complex the interpolation, the slower the conversion. However, it is highly recommended to use interpolation.

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

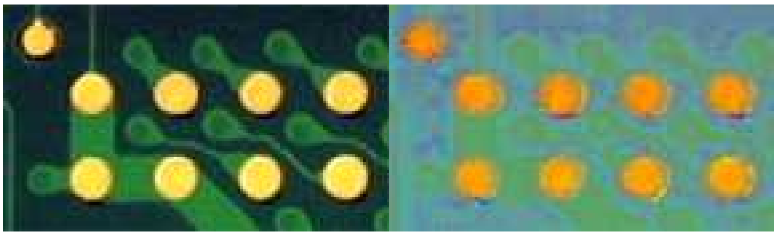


Converted images with no (top), standard (left) and improved interpolation method (right)

LUT for Gain/Offset (Color)

Separate gains and offsets can be applied to each of the three components of an image (contrast enhancement transform). The RGB image must be transformed to the targeted color space, gains and offsets applied, then transformed back to RGB.

- When applied to a mixture representation, all three gains and offset should vary in a similar way.
- When applied to luma/chroma representations, the gain and offset of the chromatic components should vary in a similar way.
- When applied to intensity/saturation/hue representation, it makes no sense to apply gain and offset to the hue component.



Enhanced saturation / Uniform lightness

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

LUT for Color Calibration

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

- Sample colors are the average color in a suitable ROI using `PixelAverage`.
- True color values are specified in the XYZ color system. Even though the reference colors are described by their XYZ coordinates, the image to be calibrated must contain RGB information.

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

LUT for Color Balance

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

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

Gamma Pre-Compensation

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

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

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

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

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

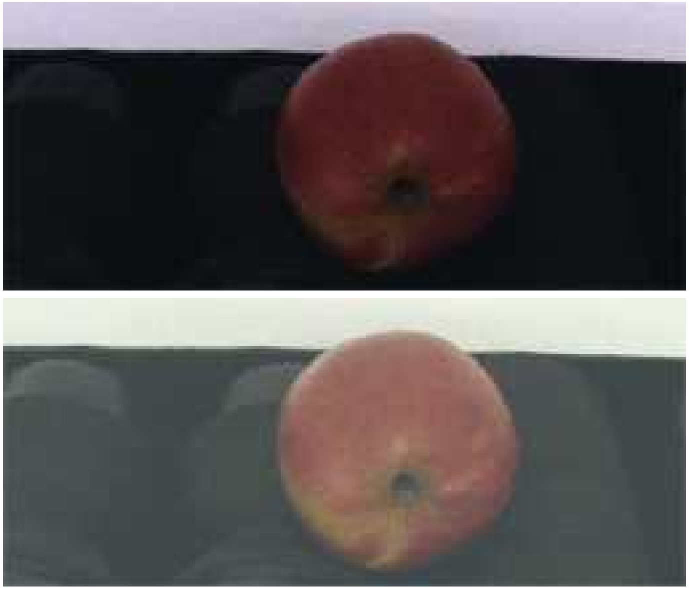
Gamma Pre-Compensation Cancellation

Many color cameras have a built-in gamma pre-compensation feature that can be turned off. If this feature cannot be turned off and is not desired, its effect can be canceled by applying the direct gamma transform. The following pre-defined gamma values are available for this purpose:

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

White Balance

A camera may exhibit color imbalance, i.e. the three color channels having mismatched gains, or the illuminant (the light sources) not being perfectly white. When this occurs, the white areas appear as an unsaturated color. The white balance correction automatically adjusts three independent gains so that the components of a white pixel become equal. This means that a white balance calibration step is required, during which a white surface must be shown to the camera and the corresponding color component are measured. `PixelAverage` can be used for this purpose.



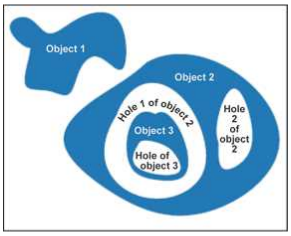
Raw image, and image with white balance and gamma pre-compensation

PART IV

MATCHING AND MEASUREMENT TOOLS

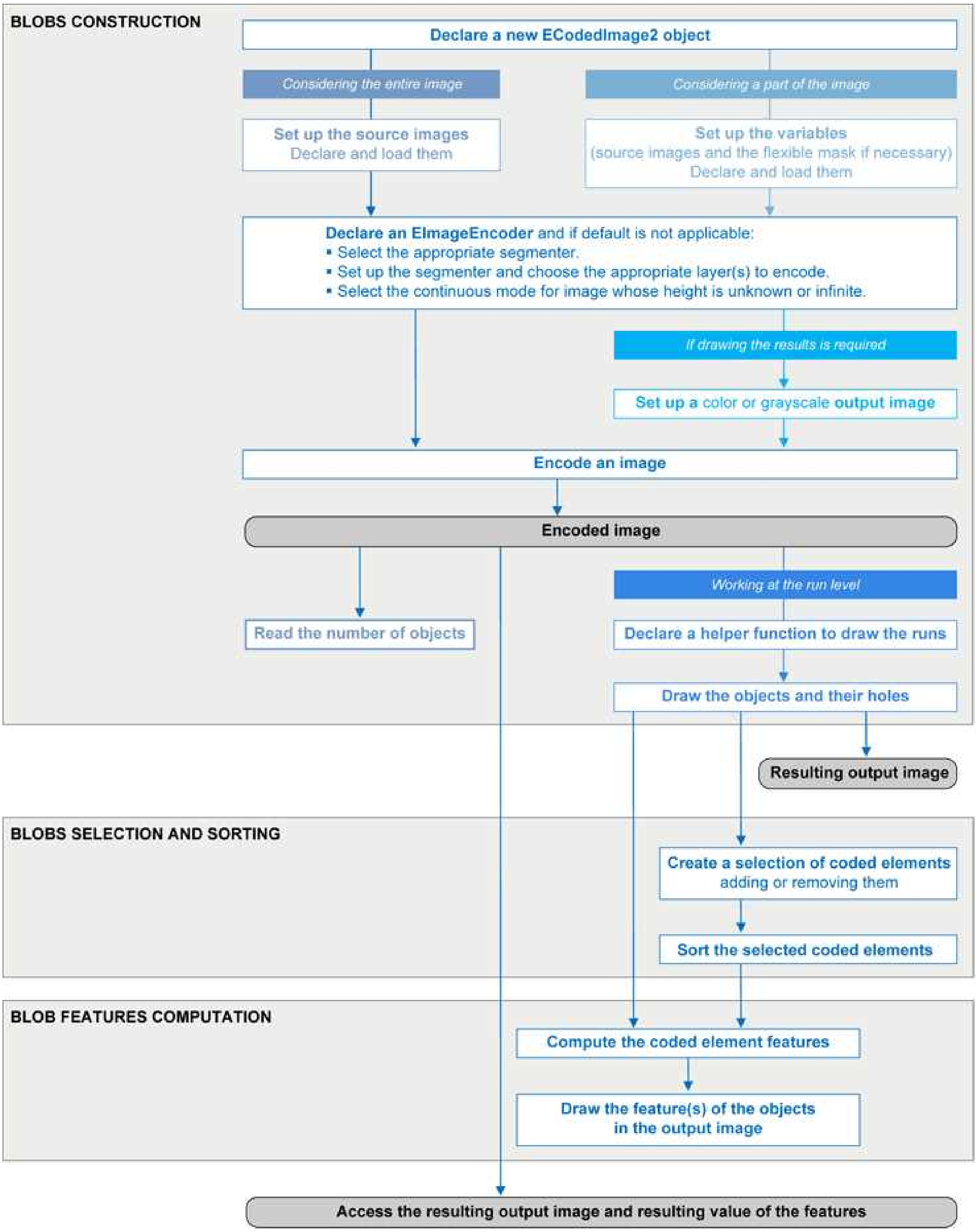
0.1. EasyObject - Analysing Blobs

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



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

Workflow


BLOBS CONSTRUCTION
Declare a new ECodedImage2 object
Considering the entire image
Considering a part of the image
Set up the source images
Declare and load them
Set up the variables
(source images and the flexible mask if necessary)
Declare and load them
Declare an EImageEncoder and if default is not applicable:
▪ Select the appropriate segmenter.
▪ Set up the segmenter and choose the appropriate layer(s) to encode.
▪ Select the continuous mode for image whose height is unknown or infinite.
If drawing the results is required
Set up a color or grayscale output image
Encode an image
Encoded image
Working at the run level
Read the number of objects
Declare a helper function to draw the runs
Draw the objects and their holes
Resulting output image
BLOBS SELECTION AND SORTING
Create a selection of coded elements
adding or removing them
Sort the selected coded elements
BLOB FEATURES COMPUTATION
Compute the coded element features
Draw the feature(s) of the objects
in the output image
Access the resulting output image and resulting value of the features

Blob Definition

A blob is a grouping of neighboring pixels of the same gray level range.
Blobs may be objects or holes in objects. EasyObject functions analyze both objects and holes.

When blobs are built, the inclusion relationship between holes and objects is computed.

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

Blobs are handled as independent entities:

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

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

Build Blobs

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

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

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

Functions

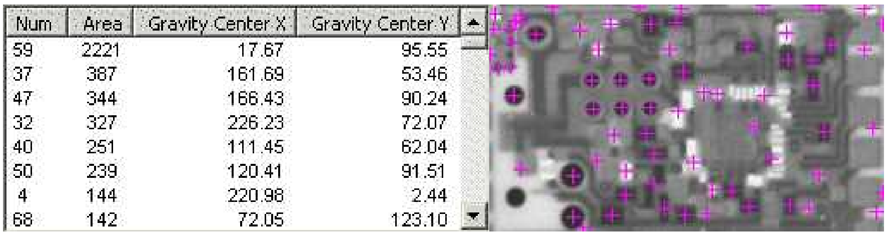
- Segmentation `GetSegmentationMethod` and `SetSegmentationMethod`
- Grayscale single threshold `EGrayscaleSingleThresholdSegmenter`
- Grayscale double threshold `EGrayscaleDoubleThresholdSegmenter`
- Color single threshold `EColorSingleThresholdSegmenter`
- Color range threshold `EColorRangeThresholdSegmenter`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

Pixel aggregation (encoder)

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

Extract objects (using geometric parameters)

Once an image has been encoded, the coded elements (objects or holes) are accessible through the abstract class `ECodedElement` which provides a large set of methods applicable to a particular coded element:



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

Features computation and display

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

Image Segmenters

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

1. Grayscale Single Threshold (default)

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

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

EasyObject provides 5 thresholding methods:

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

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

2. Grayscale Double Threshold

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

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

The **Low Threshold** and **High Threshold** are user-defined integer values, set with SetLowThreshold and SetHighThreshold methods, and got with GetLowThreshold and GetHighThreshold methods.

3. Color Single Threshold

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

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

The **Color Threshold** is a set of three **user-defined** integer values designating a color in the color space, set with SetThreshold method and got with GetThreshold method.

4. Color Range Threshold

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

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

The Low Threshold and High Threshold are each a set of three user-defined integer values designating a color in the color space, set with SetLowThreshold and SetHighThreshold methods and got with GetLowThreshold and GetHighThreshold methods.

5. Image Range

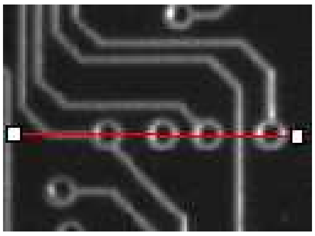
The following cases need a segmentation using **pixel-by-pixel thresholding** which gives an allowed range of values for each pixel:

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

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

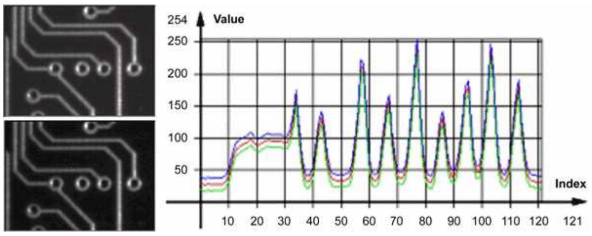
Preparing high and low reference images

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



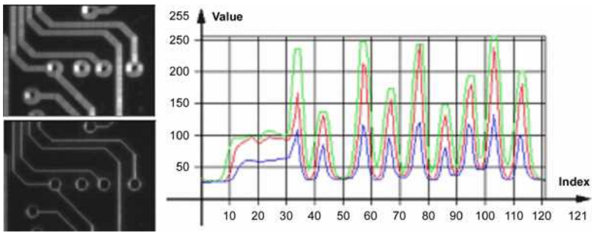
Source image

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



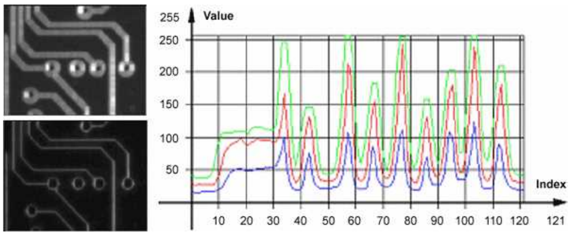
Gray-level tolerance margins

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



Geometric tolerance margins

Combining both kinds of tolerance margins gives the best results.



Combined margins

Image Segmenter

EImageRangeSegmenter and EReferenceImageSegmenter are applicable to BW8, BW16, and C24 images; and produce coded images with two layers.

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

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

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

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

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

- BW8: SetHighImageBW8GetHighImageBW8
- BW16 SetHighImageBW16GetHighImageBW16
- C24 SetHighImageC24GetHighImageC24

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

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

6. Labeled Image

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

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

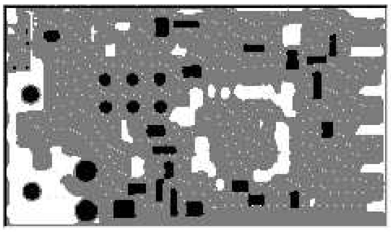
7. Binary Image

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

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

Image Encoder

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



Object building

Selecting the Layers to Encode

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

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

Enabling/disabling layer encoding for each layer individually

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

Two-layer segmenters

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

Three-layer segmenters

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

Manually Assigning a Layer Index to Each Layer Individually

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

Two-layer segmenters

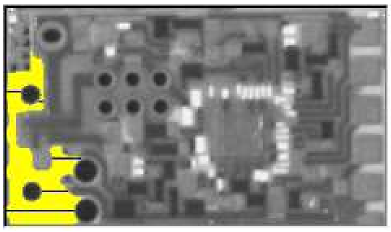
Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerIndex	IsBlackLayerIndex	**0**
White layer	SetWhiteLayerIndex	IsWhiteLayerIndex	**1**

Three-layer segmenters

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

Runs

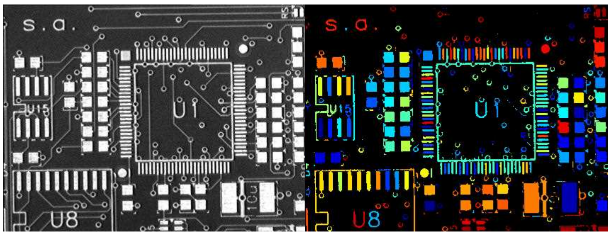
For the sake of computational efficiency, the objects are described as lists of runs. A run is a sequence of adjacent pixels that share homogeneous properties (such as being above a given threshold). These runs are merged in objects by the image encoder.



A single object with five enhanced runs

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

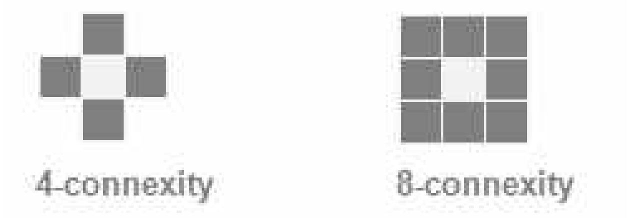
1. **Declare a new** `ECodedImage2` object.
2. **Declare an** `EImageEncoder` and, if applicable, select the appropriate segmenter. Setup the segmenter and choose appropriate layer(s) to encode.
3. **Setup an output image**.
4. **Encode the image**.
5. **Color the runs in the output image**. Iterate over the objects of a specific layer by constructing a loop and then a `RunsIterator` object. This iterator allows to browse runs of the considered object. Once the iterator has finished a run of the considered object, the inner loop processes the pixels spanned by this run in the output image.
6. **Select a specific layer**.



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

Connexity

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



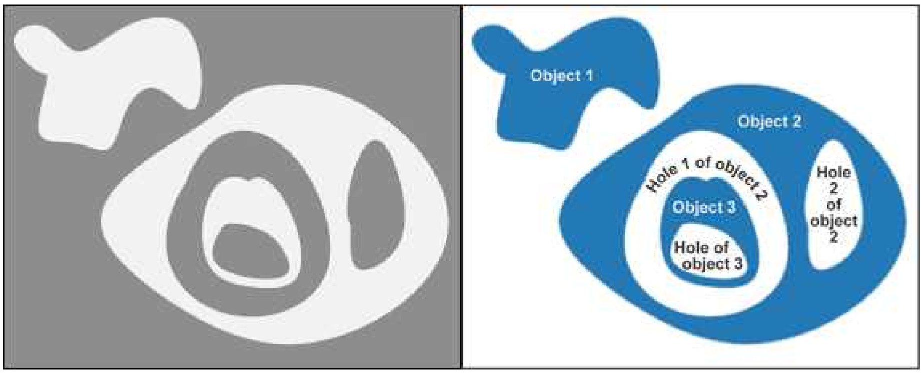
Multiple images can be encoded in continuous mode.

Holes Construction

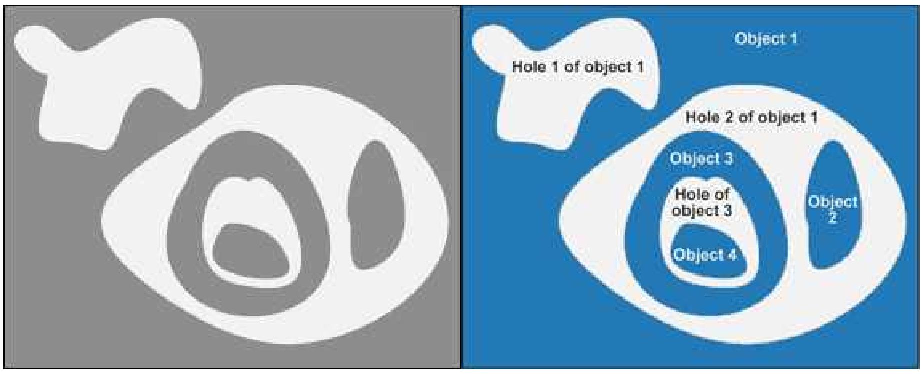
A hole is a set of connected pixels that are entirely surrounded by a parent object (4 or 8 pixels depending on the connexity mode).

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

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



Encoding the white layer (3 objects and 3 holes)

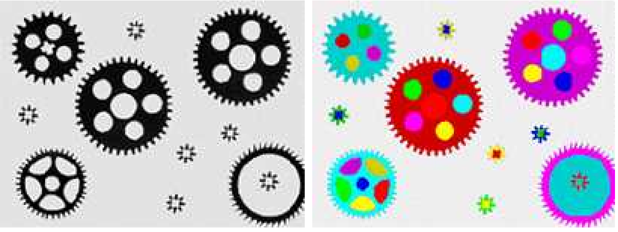


Encoding the black layer (4 objects and 3 holes)

How to Color holes

1. **Declare a new** ECodedImage2 object.
2. **Declare an** EImageEncoder and, if applicable, select and setup the appropriate segmenter, and choose the appropriate layer(s) to encode.
3. **Setup an output image**.
4. **Encode the image**.
5. **Declare a helper function to draw the runs.** A helper function (see also section Object Construction/Working at the Run Level) draws the runs in an output image, using, for example, a given color. This function can be shared for objects and holes.

6. **Draw the objects and their holes in the output image.** It is necessary to iterate over the objects of the chosen layer.
   a. The helper function draws the runs of each object (`DrawRuns`) using a specific color.
   b. The holes are iterated over the current object, and their runs are drawn.
   c. Each hole of an object is drawn with a different color computed in the global function (`GetFadedColor`) which returns a color that depends upon the hole index, for example a gradation of red to green colors.



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

Normal vs. Continuous Mode

Normal Mode (default)

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

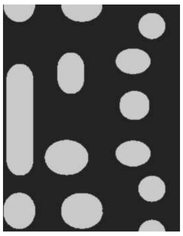
Continuous Mode

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

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

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

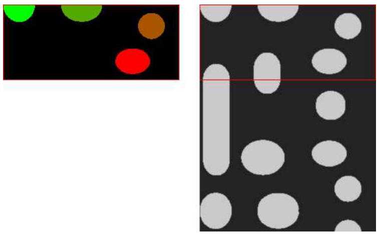
In this example, we generate a sequence of color images that exhibit objects encoded over successive chunks



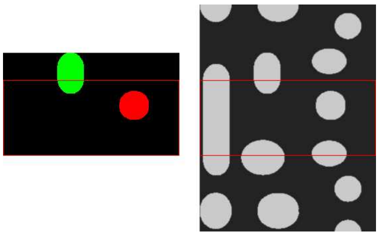
Original image

Three chunks of the image

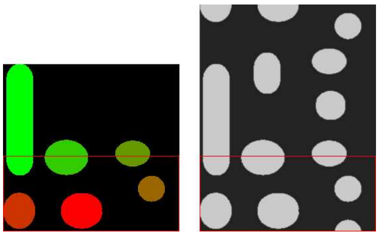
1. **Draw the objects encoded in a layer of a coded image.** This code is essentially the same as in "Browsing Runs" code snippet. The only difference is that an offset can be applied along the Y-axis.
2. **Define a function to draw the objects of a layer.** If a coded image contains objects that were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



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



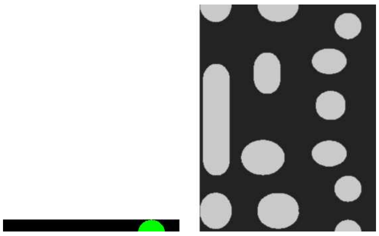
When count reaches 1, one of these two objects becomes completed,
which leads to the encoding of the following image.
Two other objects are not encoded yet at this time.
Here is the result of the encoding of the last chunk (count = 2).



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

Flushing Continuous Mode

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



Result of the flush

Selecting and Sorting Blobs

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

Create / Modify Selection

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

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

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

Clear selection

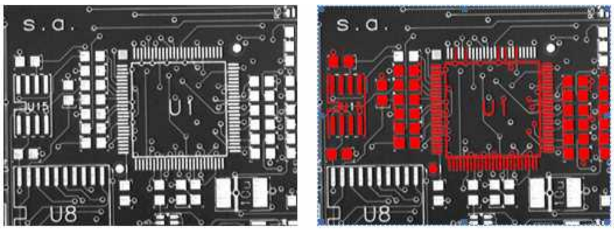
You can clear a previous selection using `EObjectSelection::.Clear`.

Sort selection

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

Example

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



Source image, and selection of objects

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

Advanced Features

Computable Features

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

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

Position

Limit
(top, bottom, left, right)
	`ECodedElement::.GetTopLimit`
	`ECodedElement::.GetBottomLimit`
	`ECodedElement::.GetLeftLimit`
	`ECodedElement::.GetRightLimit`

Gravity center
(X and Y)
	`ECodedElement::.GetGravityCenter`
	`ECodedElement::.GetGravityCenterX`
	`ECodedElement::.GetGravityCenterY`

Weight gravity center (X and Y)	`ECodedElement::.ComputeWeightedGravityCenter`

Gravity center and weight gravity center

The **gravity center** returns the abscissa of the gravity center of the coded element.

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

Extents

Area (pixel count)	`ECodedElement::.Area`
Feret box (center X and Y, height, width with distinct orientation angles at 22, 45, 68 degrees)	`ECodedElement::.ComputeFeretBox` ECodedElement::.GetFeretBox22Box ECodedElement::.GetFeretBox22Center ECodedElement::.GetFeretBox22CenterX

	ECodedElement::.GetFeretBox22CenterY ECodedElement::.GetFeretBox22Height ECodedElement::.GetFeretBox22Width ECodedElement::.GetFeretBox45Box ECodedElement::.GetFeretBox45Center ECodedElement::.GetFeretBox45CenterX ECodedElement::.GetFeretBox45CenterY ECodedElement::.GetFeretBox45Height ECodedElement::.GetFeretBox45Width ECodedElement::.GetFeretBox68Box ECodedElement::.GetFeretBox68Center ECodedElement::.GetFeretBox68CenterX ECodedElement::.GetFeretBox68CenterY ECodedElement::.GetFeretBox68Height ECodedElement::.GetFeretBox68Width
Bounding box (center X and Y, height, width)	ECodedElement::.GetBoundingBox ECodedElement::.GetBoundingBoxCenter ECodedElement::.GetBoundingBoxCenterX ECodedElement::.GetBoundingBoxCenterY ECodedElement::.GetBoundingBoxHeight ECodedElement::.GetBoundingBoxWidth
Min. enclosing rectangle (angle, center X and Y, heath, width)	ECodedElement::.MinimumEnclosingRectangle ECodedElement::.MinimumEnclosingRectangleAngle ECodedElement::.MinimumEnclosingRectangleCenter ECodedElement::.MinimumEnclosingRectangleCenterX ECodedElement::.MinimumEnclosingRectangleCenterY ECodedElement::.MinimumEnclosingRectangleHeight ECodedElement::.MinimumEnclosingRectangleWidth

Feret box

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

- **Bounding box** is the Feret box at 0°.
- **Minimum enclosing rectangle** is the Feret box with the minimum surface across all the possible angles.
- **Width** of a **FeretBox rectangle** is the length of the rectangle side that exhibits the smallest angle with the X-axis. This is NOT necessarily the smallest side!
- The **height** of a Feret box rectangle is the length of the other side of the rectangle.

Miscellaneous

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

Ellipse of inertia

Eccentricity of the ellipse of inertia	ECodedElement::.Eccentricity
Moment	ECodedElement::.GetCentralMoment ECodedElement::.GetMoment ECodedElement::.GetNormalizedCentralMoment

Ellipse (angle, height, width)	ECodedElement::.GetEllipseAngle ECodedElement::.GetEllipseHeight ECodedElement::.GetEllipseWidth
Second order geometric moments (Sigma: X, XX, XY, Y, YY)	ECodedElement::.GetSigmaX ECodedElement::.GetSigmaXX ECodedElement::.GetSigmaXY ECodedElement::.GetSigmaY ECodedElement::.GetSigmaYY

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

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

NOTE
Note. Formulas (N = area):

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

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

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

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

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

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

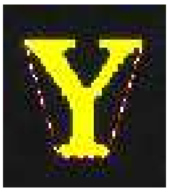
$$\mathrm{HEIGHT} = 4\sqrt{I_B}$$

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

Convex Hull

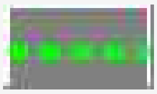
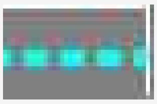
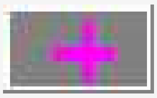
The convex hull of a shape is the convex polygon of minimum area that completely surrounds an object. The convex hull can be used to characterize the object footprint, as well as to observe concavities. Given that the number of vertices of the convex hull is variable, they are stored in a `EPathVector`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

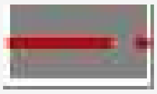
The corresponding function is `ECodedElement::.ComputeConvexHull`.



Graphic Representation

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

Objects	Graphic
Bounding box	
Convex hull	
Ellipse	
Feret box	
Feret box with an angle of 22°	
Feret box with an angle of 45°	
Feret box with an angle of 68°	
Gravity center	

Minimum enclosing rectangle	
Weighted gravity center	

Coordinate System and Conventions

Coordinate system

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

Angles

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

Evaluating the features

There is one property per feature, removing the need to access the feature through an **enum**.

Draw Coded Elements

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

To draw coded elements

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

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

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

Flexible Masks in EasyObject

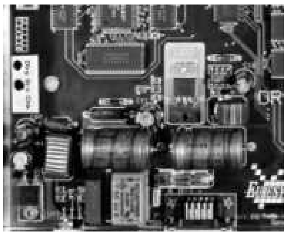
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

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

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

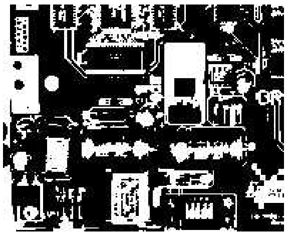
EasyObject functions that create flexible masks



Source image

1) ECodedImage2::.RenderMask: from a layer of an encoded image

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

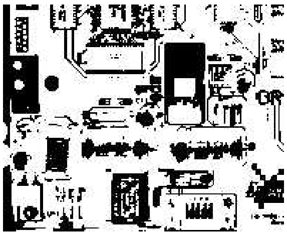


BW8 resulting image that can be used as a flexible mask

2) ECodedElement::.RenderMask: from a blob or hole

1. Select the coded elements of interest.

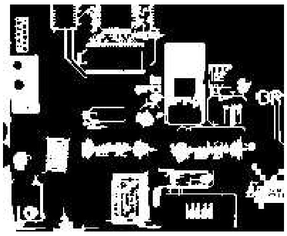
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::.RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

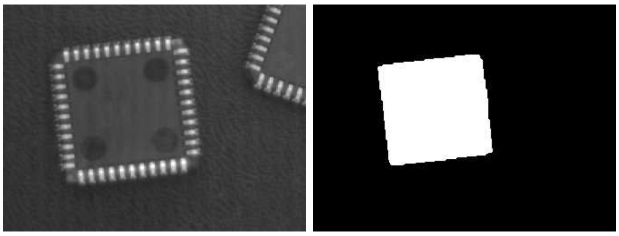
3) EObjectSelection::.RenderMask: from a selection of blobs

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



BW8 resulting image that can be used as a flexible mask

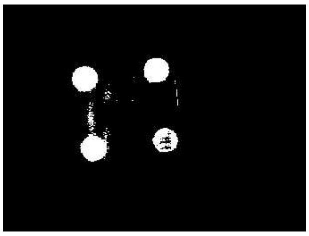
Example: Restrict the areas encoded by EasyObject



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

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

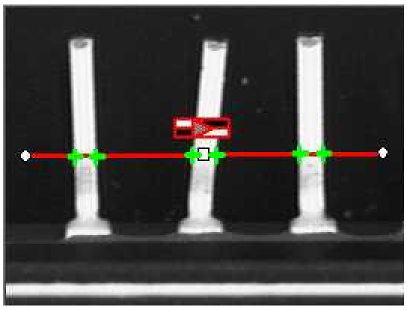
4. Encode the source image. Encoding a layer with just the area in the flexible mask is then pretty straightforward.
   We see that the circles are correctly segmented in the black layer with the grayscale single threshold segmenter:



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

0.2. EasyGauge - Measuring down to Sub-Pixel

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



Workflow

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

Uncalibrated Gauging: for a simple model

EasyGauge basic use is straightforward.

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

Uncalibrated gauging is easy to implement but has several drawbacks:

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

Calibrated gauging: for one or two simple measurement sites

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

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

Complex Gauging

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

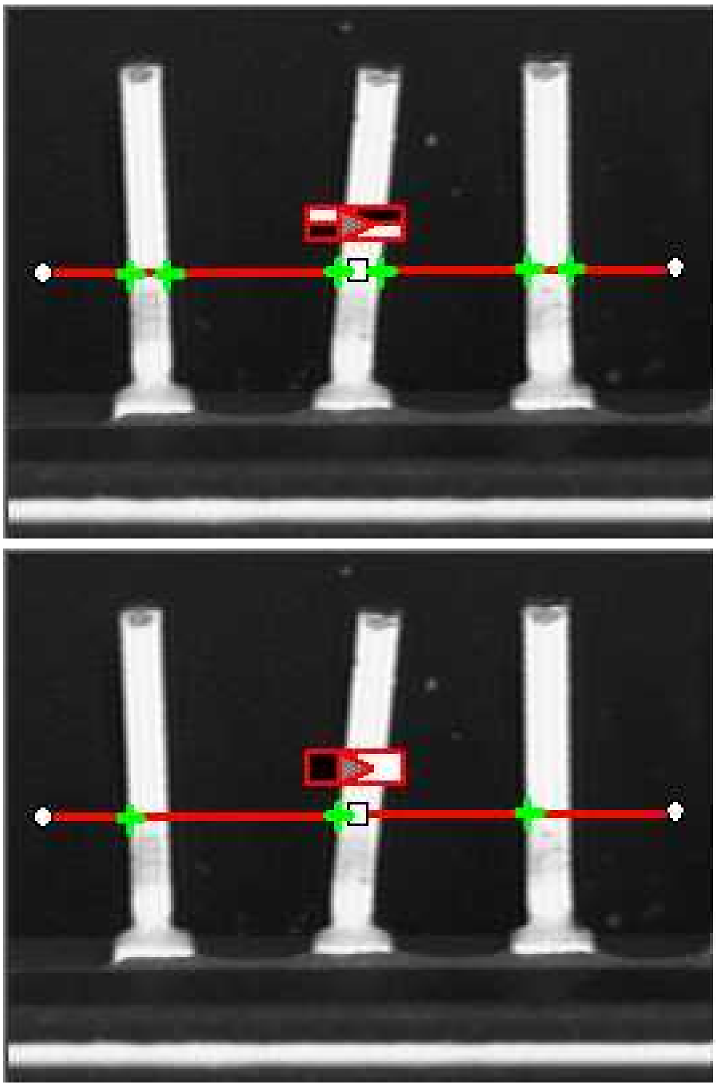
- **Attaching gauges to an** `EFrameShape` object moves the gauges with the frame (translation and/or rotation), the application program must adjust the frame position to track the inspected part.
- **Attaching gauges to another gauge** moves them according to the measured position of the supporting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

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

Gauge definitions

Point gauge

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



Point location. Contrast-based selection

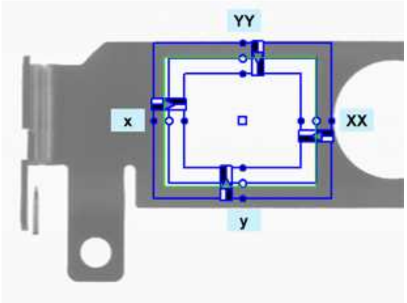
Rectangle Gauge

The placement of a rectangle gauge is defined by its nominal position (given by the coordinates of its center), its nominal size and its rotation angle.

Each side of a rectangle can have its own transition detection parameters, and can be set to active or inactive with the `ActiveEdges` property. When a side is active:

- setting the value of a parameter only applies to the currently active sides1.
- getting the value of a parameter yields a result only when the value of this property is the same for all active sides.
- only active sides are used for measurement and model fitting.

These rules allow to set different parameters for different sides, and measure parallel sides or a corner point instead of the whole rectangle. The four sides are denoted by letters "x", "y", "XX" and "YY" respectively.



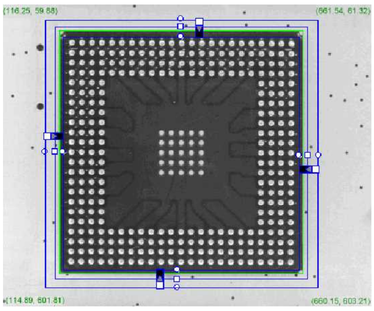
Naming conventions for the sides of a rectangle gauge

Usage

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

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

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



Locating a rectangle's corners

Wedge gauge

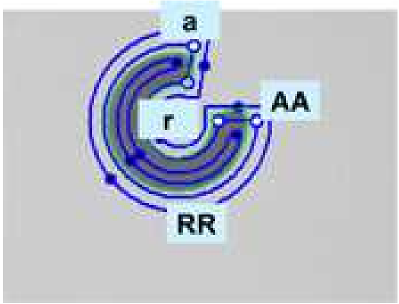
The placement of a wedge gauge is defined by its nominal position (given by the coordinates of its center), its nominal inner and outer radius (inner and outer diameter), its breadth (difference between radii), the angular position from where it extents and its angular amplitude.

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

Each side of a wedge can have its own transition detection parameters and can be set to active or inactive with the `ActiveEdges` property. When a side is active, this means that:

- setting the value of a parameter only applies to the currently active sides;
- getting the value of a parameter yields a result only when the value of this parameter is the same for all active sides;
- only active sides are used for measurement and model fitting.

So different sides can have different parameters, and you can measure parallel arcs or oblique sides, or a corner point, instead of the whole wedge. The four sides are denoted by letters "a", "r", "AA" and "RR" respectively.



Naming conventions for the sides of a wedge gauge

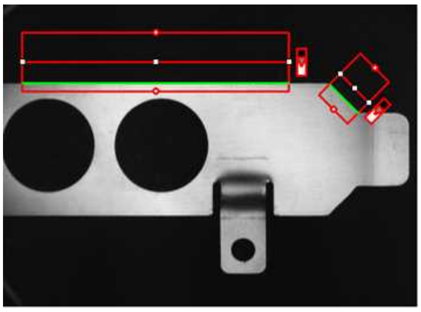
Usage

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

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

Line gauge

The placement of a line gauge is defined by its center coordinates, its length and its angle with respect to the X-axis. To constrain the line slope value, set `Angle` and `KnownAngle`.



Line fitting

Usage

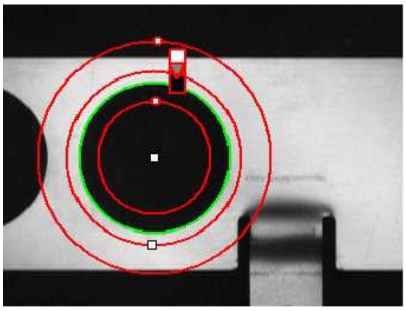
Define and position the gauge, then use `Measure` to fit the lines. To obtain the line properties, set the `ActualShape` property to **TRUE**to return the fitted line (**TRUE** value) (instead of the nominal line position **FALSE** value, default).

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

Circle gauge

The placement of a circle gauge is defined by its nominal position (given by the coordinates of its center), its nominal diameter (or radius), the angular position from where it extents and its angular amplitude.

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



Circle fitting

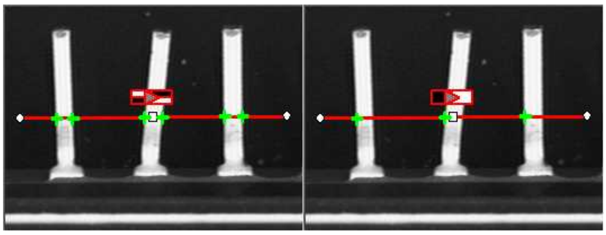
Usage

Once the gauge has been defined and positioned, use `Measure` to trigger the circle fitting operation. To obtain the measurement results, set the `ActualShape` mode to **TRUE**. The `ActualShape` mode determines whether an inquiry returns the fitted circle (**TRUE** value) or the nominal circle position (**FALSE** value, default). The requested information is then retrieved by means of the circle properties.

Alternatively, `MeasuredCircle` provides the results as an `ECircle` object.

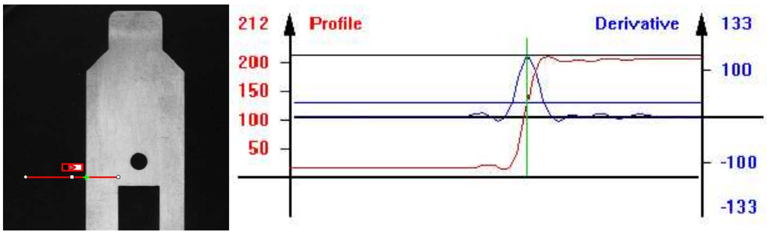
Find transition points using peak analysis

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



Point location. Contrast-based selection

Point Location principle



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

On a linear profile extracted from an image, an edge appears as a transition from dark to light (or vice versa). When plotting pixel values along the gauge, this transition appears as an S-shaped curve. The first derivative of this curve exhibits a peak around the transition point. The better the contrast, the sharper the transition and the higher the peak.

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

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

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

EPointGauge parameters

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

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

[2] Black / White / Black

GetMeasuredPoint returns an EPoint object which contains located point information. An integer index between 0 and GetNumMeasuredPoints-1 must be passed.

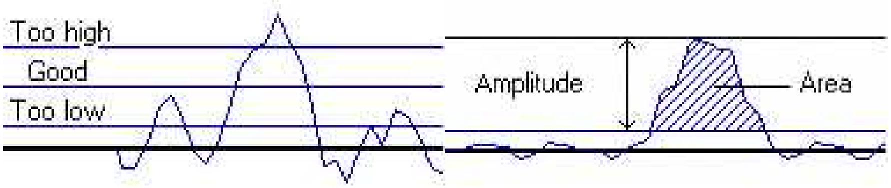
GetMeasuredPeak: Returns EPeak containing the peak's Area and Amplitude, and the delimiting coordinates along the probe segment (Start, Length and Center values).

Select Peaks to improve edge precision

The threshold level is very important:

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

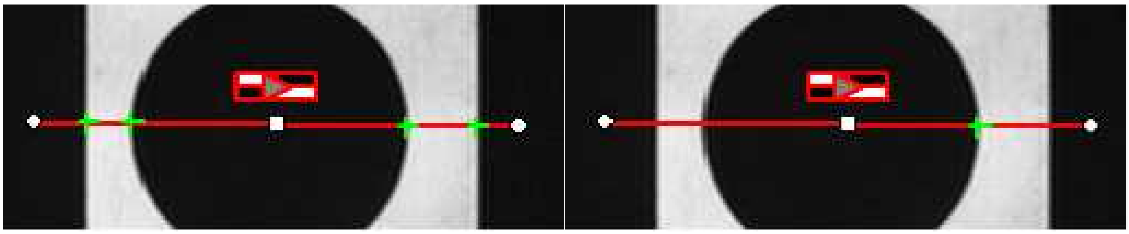
To resolve this dilemma, the EasyGauge peak selection mechanism can reject low contrast or false edges: transition strength is measured by peak *amplitude* and *area*. Every edge measurement determines peak amplitude and area. If either value falls below the minimum amplitude or minimum area, the peak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



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

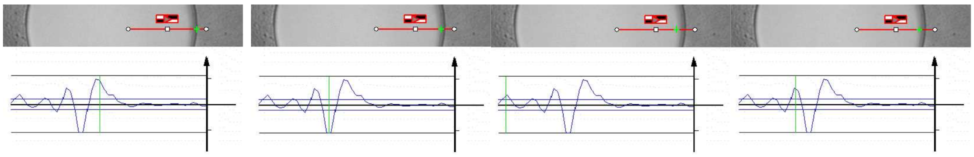
Multiple versus single transition

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



Multiple transition (left) versus single transition (right)

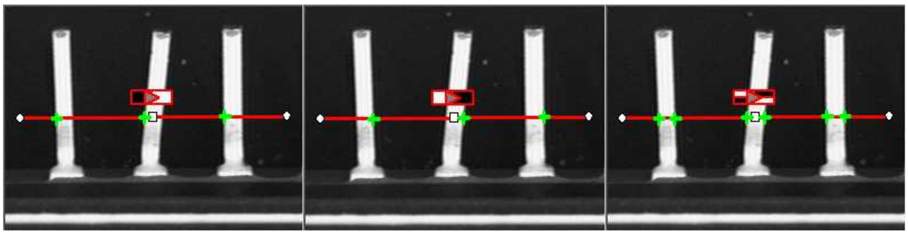
You can select the single most relevant transition based on 4 criteria: the highest peak, the peak with the largest area, the peak closest to the gauge center, or the N-th peak encountered starting from one tip of the gauge.



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

Positive or negative peak selection

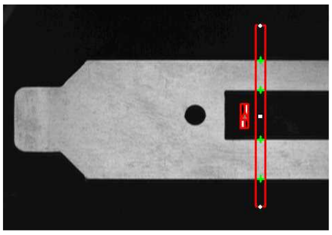
Peak selection can also be refined by choosing the transition polarity: White to Black or Black to White (i.e. positive or negative peak), or indifferent.



Black to white, white to black or indifferent polarities

Pre-filtering

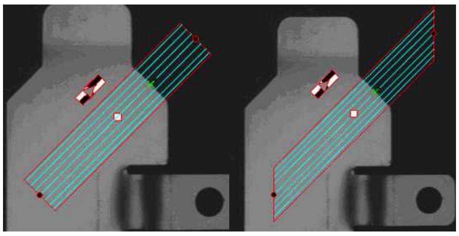
Pre-filtering the image locally can reduce noise effects.
Transverse (lengthwise) filtering averages several parallel lines when sampling the image.
Longitudinal (crosswise) uniform filtering can also be applied to the resulting profile curve.



Thick point gauge for filtering

Transverse Filtering

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



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

Point Probe Position

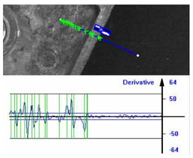
The expected **nominal** position of a point gauge is specified by its **center**, orientation **angle** with respect to the X-axis, and length **tolerance**that the point position can vary.

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

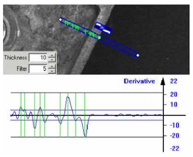
Tuning Point Measurement Parameters for unclear edges

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

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

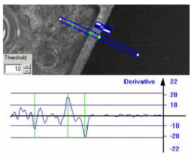


2. Decide whether noise reduction is required. Lay the gauge over the desired location and observe the profile curve and its derivative (play with the filtering parameters while looking at the plotted curve). The curve regularity gives an indication of the spread of the gray-level values.
When these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



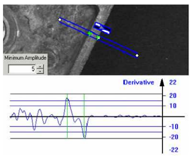
Thickness 10
Filter 5

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



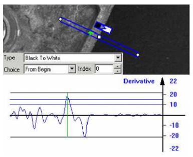
Threshold 10

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



Minimum Amplitude 5

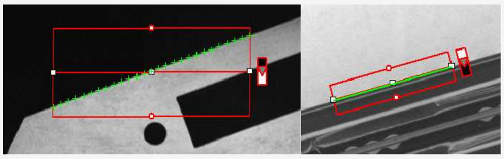
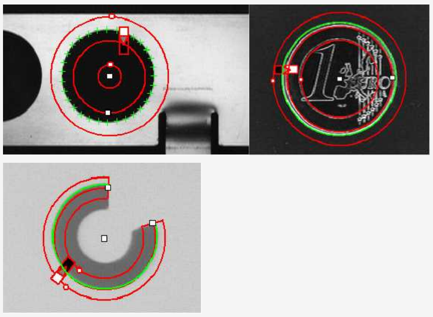
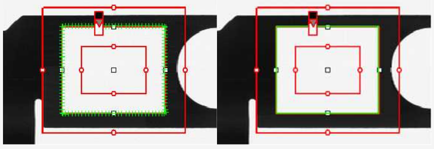
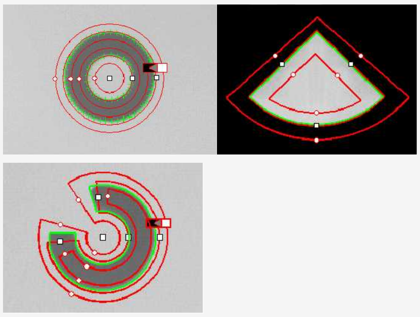
5. Choose whether all transition points are needed or just the most relevant. If all are required, they can be queried one after another. Otherwise, a point selection strategy should be chosen based on strength, order or transition polarity (black to white and/or conversely).



Type Black To White
Choice From Begin Index 0

Find shapes using geometric models

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

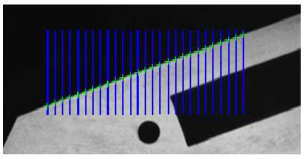
Line: Measures position and orientation of straight edges.	
Circle: Measures position and curvature of a circle or arc.	
Rectangle: Measures position, orientation and size of a rectangle.	
Wedge: Measures position, orientation and size of a ring/ disk sector / curvilinear rectangle.	

All gauge types share these common features:

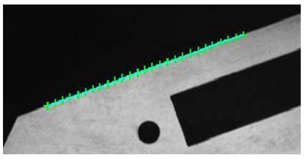
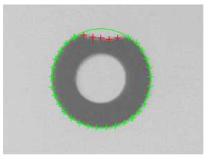
Point sampling

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

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



Sampling paths and sampled points

NumSamples returns the number of points sampled during the model fitting operation.	
Model fitting	
The model is adjusted to minimize error residue and provide the best edge parameter estimates. Rectangles and wedges have parallelism and concentricity constraints. Image shows sampled points and fitted line.	
Outlier rejection	
After model fitting, some points will be too far away from the fitted model and may harm location accuracy. EasyGauge can tag them as outliers to be ignored using the FilteringThreshold property. The outlier elimination process can be repeated several times using NumFilteringPasses. The number of valid sample points remaining after a model fitting operation is kept in NumValidSamples. The average distance of these points to the fitted model is returned by AverageDistance.	

Gauge Manipulation: Draw, Drag, Plot, Group

EasyGauge provides means to graphically interact with gauges to place and size them, combine them as a hierarchy of grouped items, and store/retrieve them and all working parameters to/from model files.

Draw

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

Drag

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

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

Plot

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

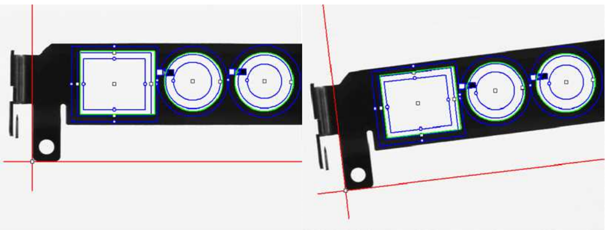
Group

Measurement gauges can be grouped (their relative placement remains fixed) to form a dedicated tool that can be moved (translated and rotated) to follow the movement of inspected items / probes before computing measurements.

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

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

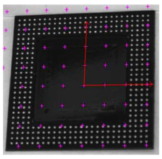
`Detach`, `DetachDaughters` dissociates the gauge or daughters from the mother.



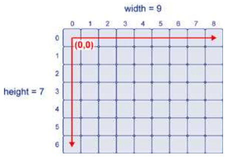
Calibration and Transformation

Field-of-view calibration

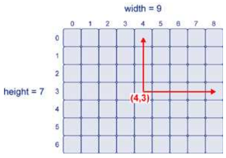
Calibration establishes the relationship between real-world point coordinates and image pixels. A simple calibration model computes faster, a repeatable part position is easier to locate.



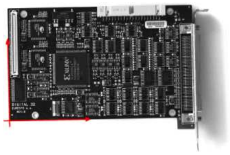
The Raw sensor coordinate system starts from upper left and extends rightwards and downwards.
The range of abscissae is **0** to **width-1** and the range of ordinates is **0** to **height-1** where integer coordinate values correspond to pixel centers.



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



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



Before World-to-Sensor Transform

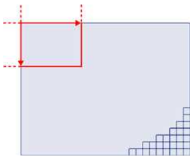
Before converting from world to sensor coordinates, sources of distortion should be eliminated:

- adjust sweep frequency or scanning speed to avoid non-square pixels.
- adjust optical alignment to minimize perspective effect. The field of view should be parallel to the sensor plane.

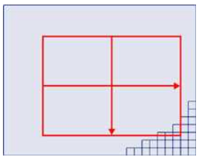
- use long focal distances and good quality lenses to minimize Optical distortion.
- use appropriate scale factor based on lens magnification, observation distance and focusing.
- minimize skew and translation effects by secure fixtures, and part-movement / acquisition-triggering synchronization.

Effects of World-to-Sensor Transform

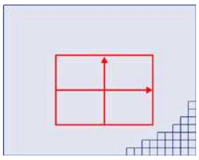
- **No calibration**. World and sensor coordinates are identical.



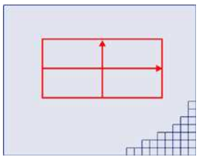
- **Translated calibration**: The coordinate origin can be moved. World coordinates correspond to pixel units.



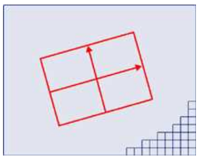
- **Isotropic scaling** (square pixels). A scale factor converts pixel values to physical measurements.



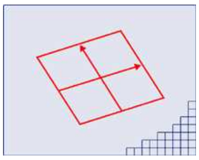
- **Anisotropic scaling** (non-square pixels). Uses two scale factors with pixel aspect ratio (X /Y) in the range [-4/3, -3/4] (or [3/4, 4/3]). Pixels are always displayed as square, so the image appears stretched.



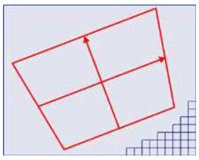
- **Scaled and skewed** (square pixels). Real-world axis aligns with rotated inspected part using translation, rotation and scaling.



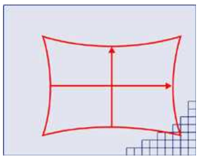
- **Scaled and skewed** (non-square pixels). Distortion is apparent. Occurs when camera scan speed does not match pixel spacing.



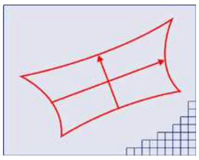
- **Perspective distortion** causes further away objects to look smaller; lines remain straight but angles are not preserved.



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



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



Calibration using EWorldShape

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

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

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

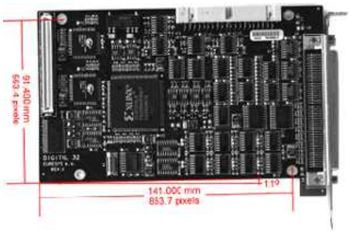
There are several ways to obtain the calibration coefficients:

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

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

1. Take a picture of the part to be inspected or a calibration target (e.g. rectangle).
2. Locate feature points such as corners in the image (by the eye) and determine their coordinates in pixel units —let `(i,j)`.
3. Use the Euclidean distance formula to derive the calibration coefficient: $C = \frac{\sqrt{(i_1 - i_0)^2 + (j_1 - j_0)^2}}{D}$ where `C` is a calibration coefficient, in pixels per unit, and `D` is the world distance between the corresponding points, in units.
4. For non-square pixels repeat this operation for pairs of horizontal and vertical points.

To estimate a skew angle, apply this formula to two points on the X-axis in the world system:
$\theta = \arctan \frac{j_1 - j_0}{i_1 - i_0}$



Estimating scale factors and skew angle

When the calibration coefficients are available, use `SetSensor` to adjust them and set the calibration mode, or set them individually using: `SetSensorSize`, `SetFieldSize`, `SetResolution`, `SetCenter`, `SetAngle`.

Pass a set of reference points (landmarks) to a calibration function

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

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

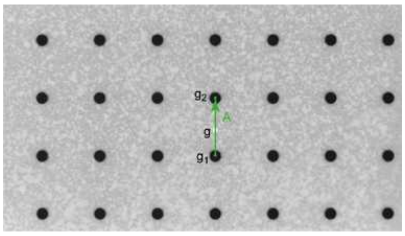
Analyze a Calibration target

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

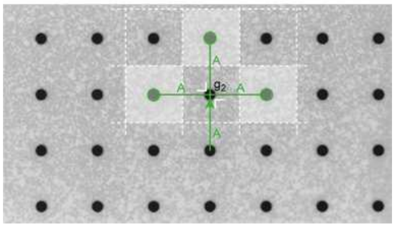
Open eVision relies on the use of a specific target holding a rectangular grid of symmetrical dots (of any shape)with no other object on the grid.

Dot Grid based calibration example

1. Grab an image of the calibration target in such a way that it covers the whole field of view (or restrict the image of view to an ROI where only dots are visible).
2. Apply blob analysis to extract the coordinates of the centers of the dots, as can be done by EasyObject.
3. Pass all points detected to AddPoint (sensor coordinates only).
4. Call RebuildGrid to reconstruct a grid to calibrate a field of view using an iterative algorithm which computes the world coordinates of each dot.
   a. The grid points nearest to the gravity center (g) of grid points are selected (g_1 and g_2) to form the first reference oriented segment, of length A.



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

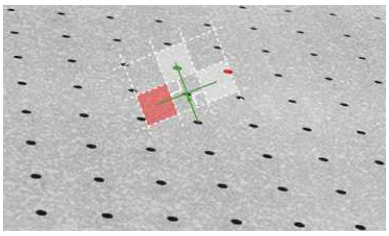


c. The 3 perpendicular segments are the references of the next iterative searches. The algorithm goes back to step 2.

5. Call `Calibrate`the landmark approach.

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

1. A tolerance area does not contain a neighboring point (red square in the figure).
2. A tolerance area contains more than one neighboring point.
3. The point in the tolerance area is not the correct one. For instance, the point might be diagonally connected (red point in the figure).



Advanced Features

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

Sensor Width and Height

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

Field-of-View Width and Height

The field-of-view (f-o-v) width and height give the actual image size, in length units, i.e. the size of the rectangle corresponding to the image edges in the world space. These values are related to the pixel resolution by the following equations:

$$\text{f-o-v width} = \text{pixel width} * \text{sensor width}$$
$$\text{f-o-v height} = \text{pixel height} * \text{sensor height}$$

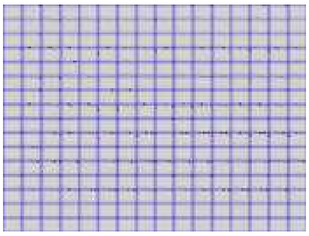
or

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

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

`Scale`

Ratio



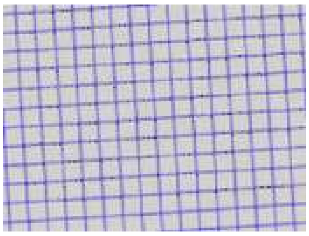
Anisotropic aspect ratio

Center Abscissa and Ordinate

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

Skew Angle

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

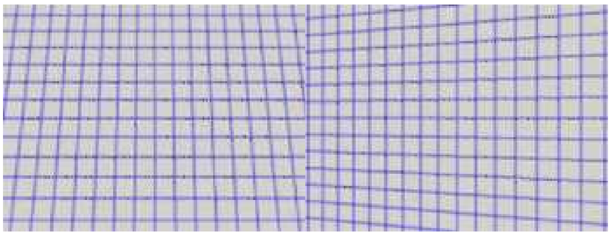


Skew angle

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

X and Y Tilt Angles

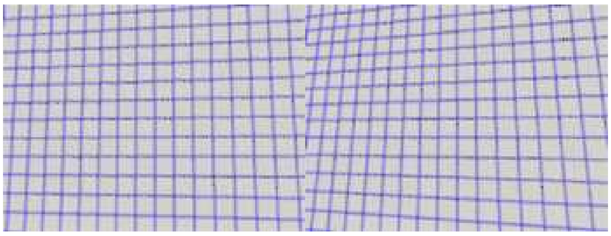
The X and Y tilt angles describe the viewing plane direction. They correspond to the required rotations around X and Y axis that bring the Z axis parallel to the optical axis.



Tilt X and tilt Y angles

Perspective Strength

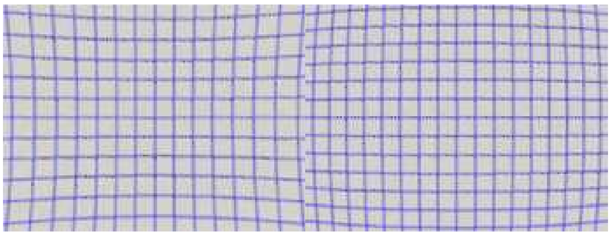
The perspective strength gives a relative measure of the perspective effect. The shorter the focal length, the larger the value.



Weak and strong perspective

Distortion Strength

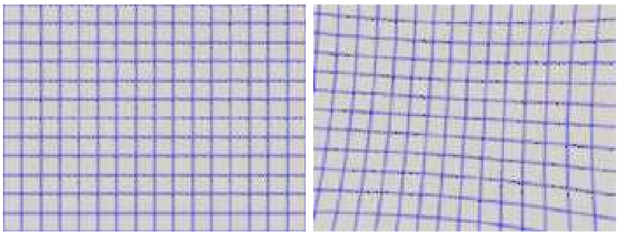
Distortion strength and `GetDistortionStrength2` give a relative measure of radial distortion in the image corners, i.e. the ratio of image diagonal length with and without distortion.



Positive and negative distortion

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

Effect of the Calibration Coefficients



No calibration coefficient: All coefficients combined.

Unwarp an image

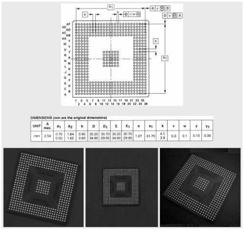
An **EWorldShape** object manages a field-of-view calibration context. Such an object is able to represent the relationship between world coordinates (physical units) and sensor coordinates (pixels), and account for the distortions inherent in the image formation process.

Image calibration is an important process in quantitative measurement applications. It establishes the relation between the location of points in an image (pixel indices) and the actual positions of those points in the real world, on the inspected item.

Calibration can be setup by providing explicit calibration parameters of the calibration model, or a set of known points (landmarks), or a calibration target.

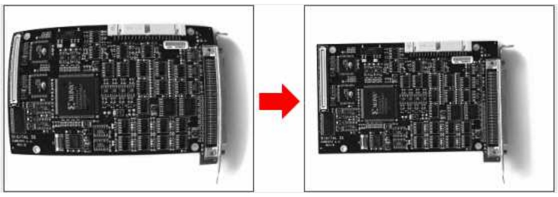
The goal of calibration is twofold:

- To gain independence with respect to the viewing conditions (part placement in the field of view, lens magnification, sensor resolution, ...), letting you describe the inspected item once for all using absolute measurements.



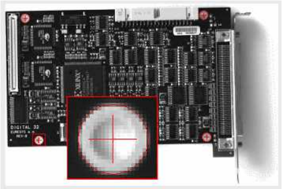
Single model versus multiple viewing conditions

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



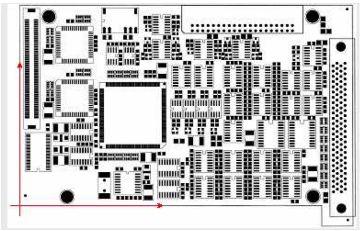
Removal of image distortion

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



Feature point in sensor space

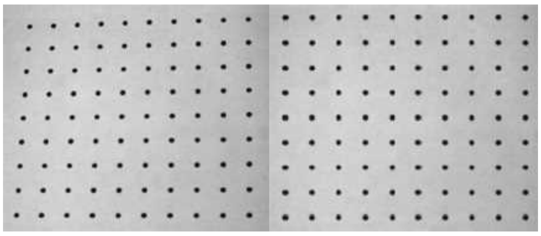
The world coordinates describe the location of points on the inspected item are expressed in an appropriate length measurement unit.
The world coordinates are actual dimensions, usually gathered from design drawings or by mechanical measurements.
They require a reference frame to be defined.



Reference frame in world space

Unwarp

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

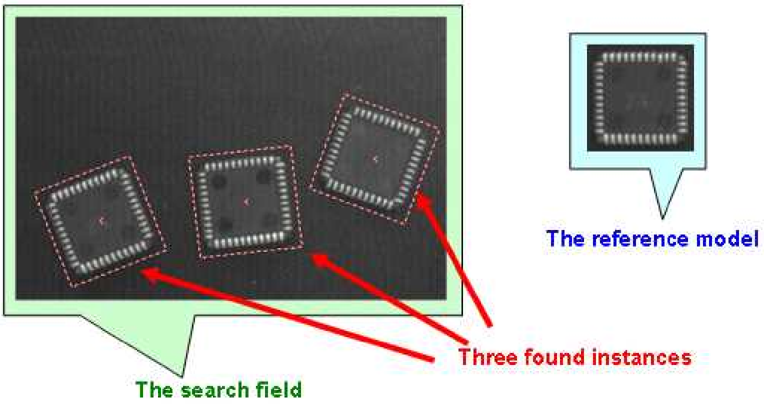


Distorted vs. Unwarped image

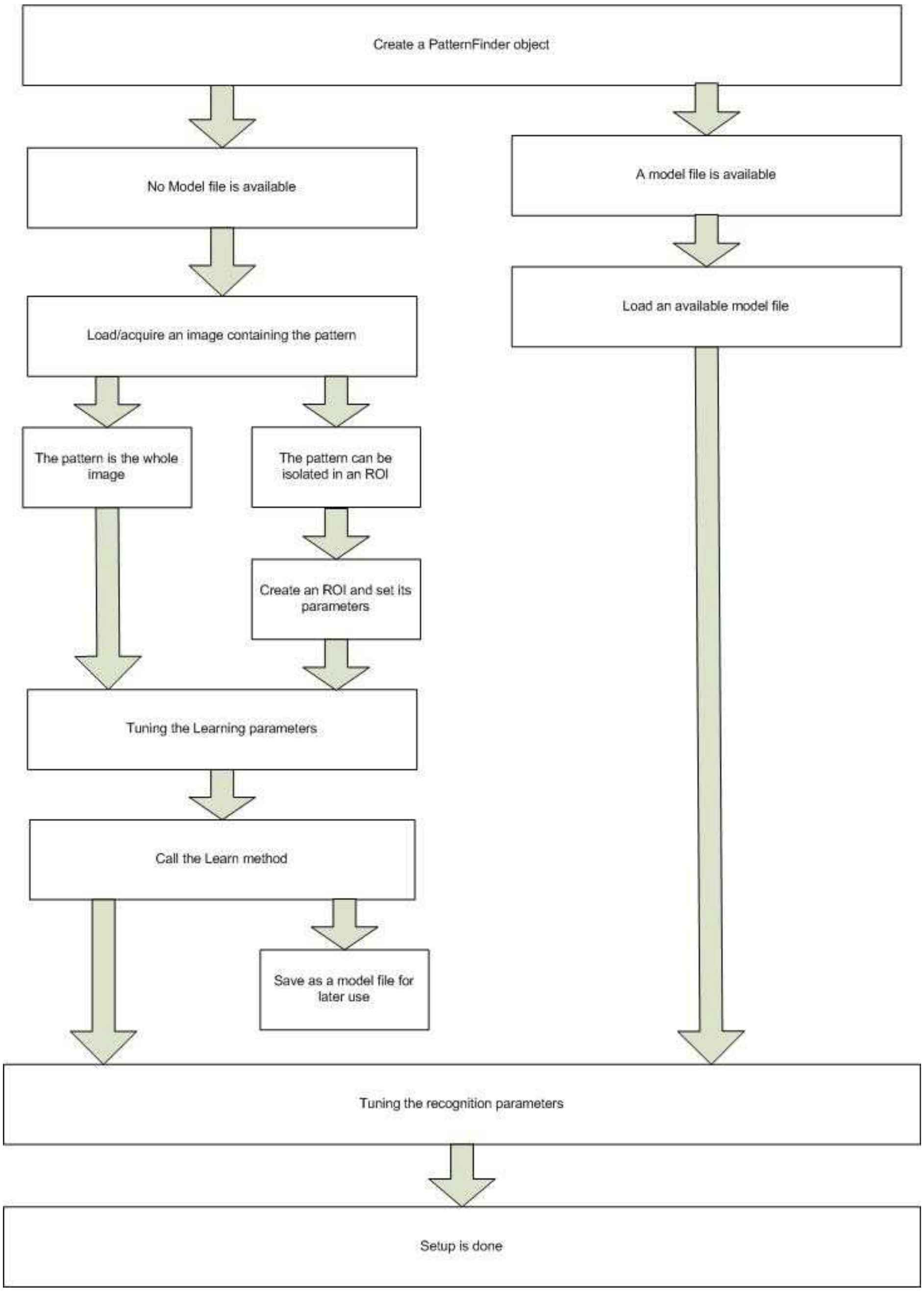
0.3. EasyFind - Matching Geometric Patterns

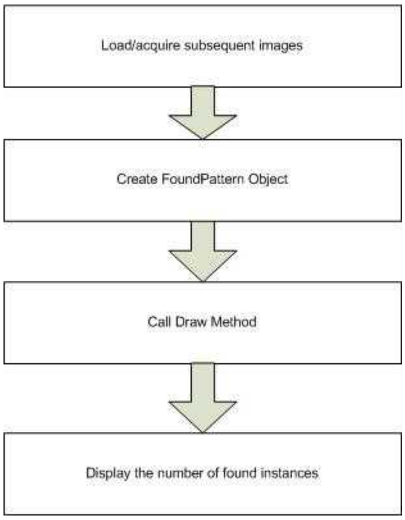
Workflow

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


The reference model
Three found instances
The search field

W orkflow

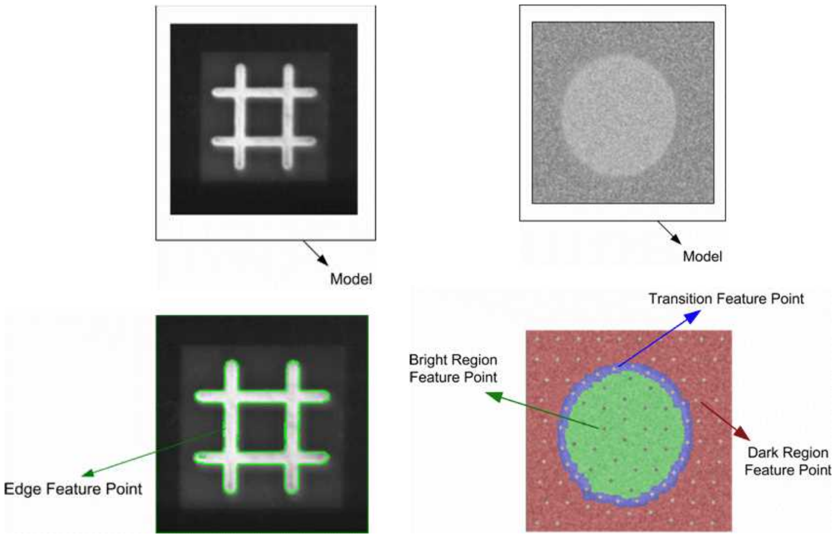

Create a PatternFinder object
No Model file is available
A model file is available
Load/acquire an image containing the pattern
Load an available model file
The pattern is the whole image
The pattern can be isolated in an ROI
Create an ROI and set its parameters
Tuning the Learning parameters
Call the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


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

Feature points definition

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

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

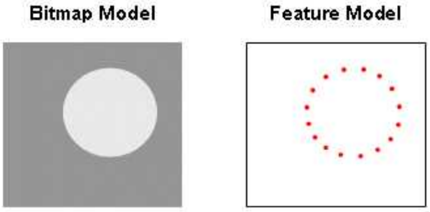


- **Edge feature points**: an abrupt change of gray level between two regions indicates an edge at this location in the search fields.
- **Transition feature points**: a smooth change of gray level between two regions indicates a transition area in their neighborhood (represented by dots in the blue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

Learning Process

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

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



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

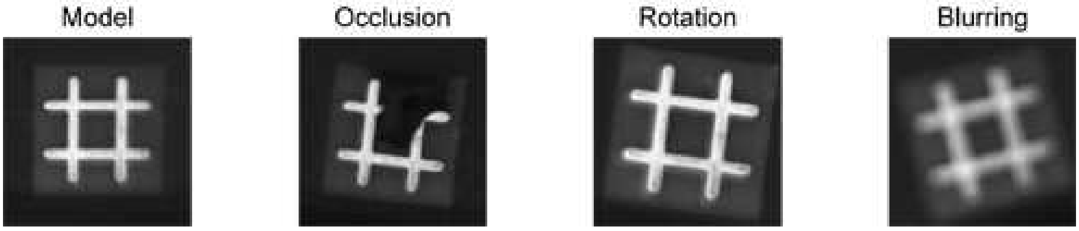
Consistent Edges

Models must be well contrasted with sharp edges. They should be substantially different from the rest of the expected search fields. Can be scaled or rotated, very robust to: blurring, noise, occlusion, illumination variation (point-by-point scores improves robustness and computation time of the finding phase).

Good for models with consistent edges, sharp contrast transitions, regions delineated by well defined edges that are in approximately the same place for each instance in all search fields.

Choose the similar points with `EPatternFinder::.ContrastMode` property:

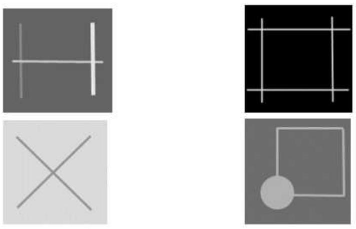
- □ PointByPointNormal: if points share the same contrast polarity.
- □ PointbyPointInverse: if points exhibit opposite contrast polarity.
- □ PointByPointAny: regardless of their respective contrast polarity.

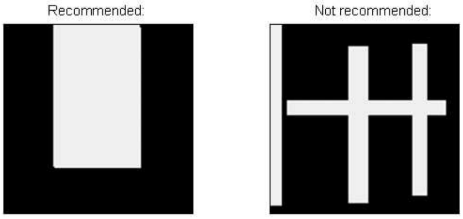


Thin Structures (defined by edge feature points)

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

Good for models containing thin elements.

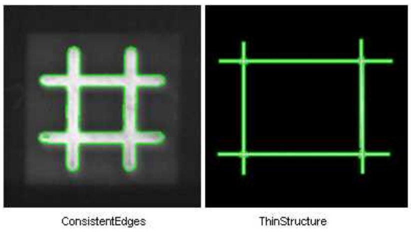




Check the Learned model is correct

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

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

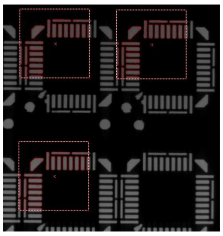


Finding Process

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

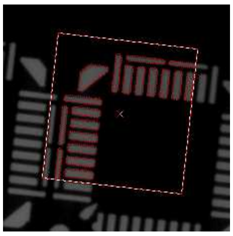
Maximum number of expected instances

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



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

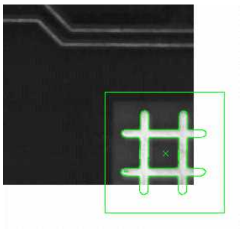
Ranges have a bias and a tolerance. For instance, for an angle bias of 20° and an angle tolerance of 5°, EasyFind returns instances with an angle between 15° and 25° with respect to the learned model (20° ± 5°).



Advanced Features

Find partial patterns

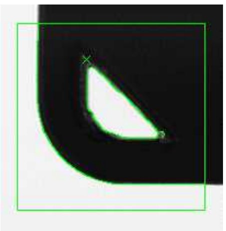
EasyFind can locate instances of thin structures and consistent edges that are partially out of the search field, if the extension of search field is set to > 0 pixels.



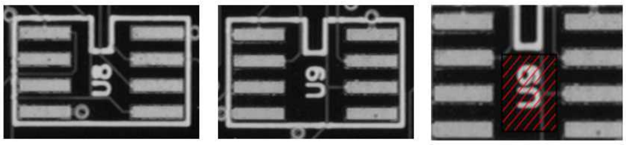
Tune Parameters

These parameters can be tuned for all models:

- □ Adjust the Light Balance using model drawing to preview the model so that it fits the useful parts of the pattern, then learn the model again.
- □ Set the gray-level threshold (this overrides light balancing) and learn the model again.
- □ Move the pivot to a specific place in the model like a corner or a hole. The pivot is the location returned by EasyFind when it finds an instance (the center by default).



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

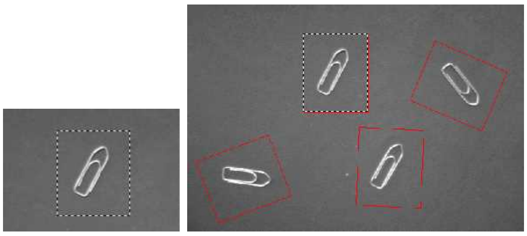


- Thin Structures may benefit from tuning these parameters:
  - □ Automatic (thin elements with the same contrast between them and their neighboring regions)
  - □ Thin elements darker than the neighborhood
  - □ Thin elements brighter than the neighborhood

0.4. EasyMatch - Matching Area Patterns

EasyMatch learns a pattern and finds exact matches:

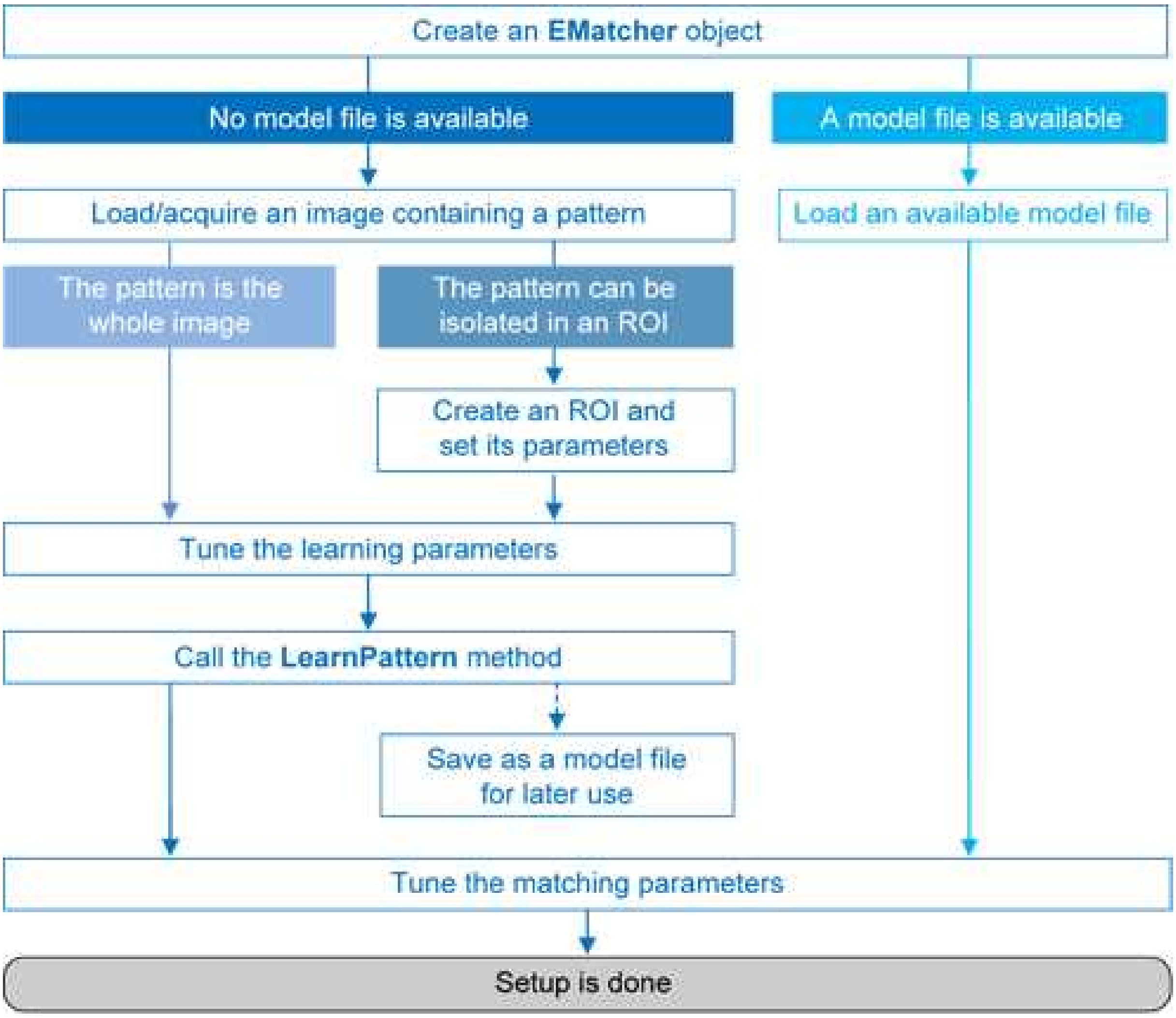
1. The pattern is learned by defining an ROI that contains the object to be matched.
   This ROI is created after iteratively learning from several images which contain the object.
2. The parameters are tuned to ensure the pattern is found reliably.
3. Images can now be searched for one or more occurrences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



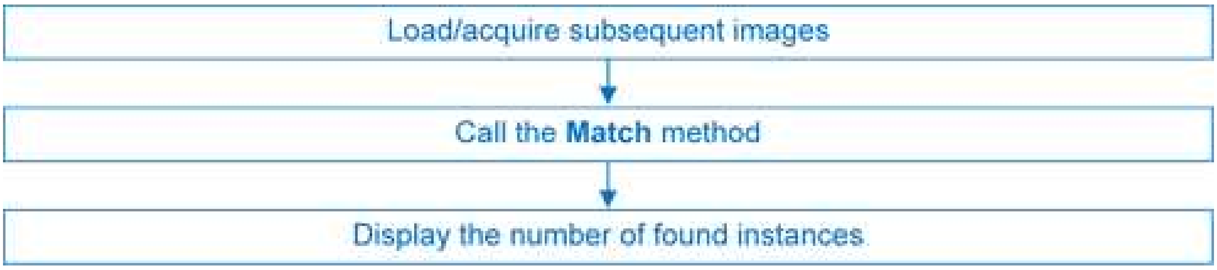
Learning and Matching a pattern

Workflow

Learning workflow


Create an EMatcher object
No model file is available
A model file is available
Load/acquire an image containing a pattern
Load an available model file
The pattern is the whole image
The pattern can be isolated in an ROI
Create an ROI and set its parameters
Tune the learning parameters
Call the LearnPattern method
Save as a model file for later use
Tune the matching parameters
Setup is done

Matching workflow


Load/acquire subsequent images
Call the Match method
Display the number of found instances

Learning Process

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

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

Best pattern characteristics

- **repeatable**, you need to know if it can translate or rotate or scale.
- **represent the object to be located.**

  It should:

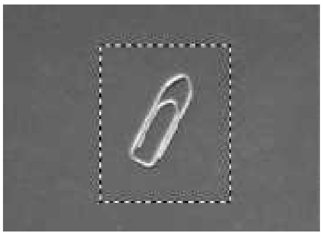
  - □ Keep the same appearance whatever the lighting conditions.
  - □ Remain at a fixed location with respect to the part.
  - □ Be rigid and not change shape.
- **exhibit good contrast in small and large scale.** It should be distinctly visible from a distance, and on a reduced image.
- **not be invariant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

Customize Parameters

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

- `DontCareThreshold`: If don't care areas are required, the corresponding pixels must hold a value below the `DontCareThreshold`.
  If all the background can be ignored, merely adjusting the `DontCareThreshold` to the right thresholding value can do.
  Otherwise, when the don't care area is unrelated to the threshold pattern image, the `DontCareThreshold` should be set to 1 and all pixels belonging to the don't care area should be set to black (value 0).
- `MinReducedArea`: To improve time performance, EasyMatch sub-samples the pattern. This parameter stipulates the minimum size of the pattern (as its area in pixels) during sub-sampling. The smaller the value, the faster the matching process, but, set too low, it decreases the matching process reliability.
  The value of `MinReducedArea` is computed automatically if `AdvancedLearning` is enabled (default behavior). Setting explicitly `MinReducedArea` will disable AdvancedLearning. A value of 64 is usually a good compromise.
- `AdvancedLearning`: If the pattern is defined as a ROI of an image, `AdvancedLearning` optimizes learning parameters, such as `MinReducedArea`, by using the whole image context. `AdvancedLearning` is enabled by default, as it leads to better results in case of tiled or periodic images. If `MinReducedArea` is set explicitly, `AdvancedLearning` is disabled. Please note that as `AdvancedLearning` changes the number of pixels in the pattern, it can have a significant impact on the matching process duration.

- FilteringMode: If the image has sharp gray-level transitions, it is better to choose a low-pass kernel instead of the usual uniform kernel.



Learning a pattern

Matching Process

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

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

You can set these parameters:

□ Rotation range: MinAngle, MaxAngle.

□ Scaling range: MinScale, MaxScale.

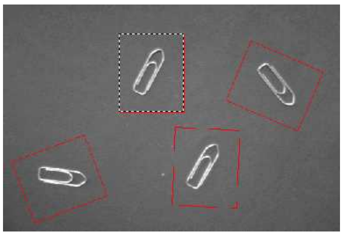
□ Anisotropic scaling range: MinScaleX, MaxScaleX, MinScaleY, MaxScaleY.

The following functions return the result of the matching:

□ NumPositions returns the number of good matches found. A good match is defined as having a score higher than prescribed value (the MinScore threshold value).

□ GetPosition returns the coordinates of the N-th good match. The positions are sorted by decreasing score.

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



Matching a pattern

Advanced Features

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

- Learning time:
  - □ Optimize number of searches: Searching all positions takes too long, so a sequence of searches is performed at various scales (reductions). The coarsest reduction is quick and approximate. Subsequent reductions work in a close neighborhood to improve location, drastically reducing the number of positions to be tried. The location accuracy is given by 2^K, where K is the reduction number.
  - □ MinReducedArea. Indicates how small the pattern can be made for rough location.
- Matching time:
  - □ Correlation mode (way to compare the pattern and the image): CorrelationMode. **Can be standard, offset-normalized, gain-normalized and fully normalized:** the correlation is computed on continuous tone values. Normalization copes with variable light conditions, automatically adjusting the contrast and/or intensity of the pattern before comparison.
  - □ Contrast mode (way to deal with contrast inversions): ContrastMode. Lighting effects can cause an object to appear with inverted contrast, you can choose whether to keep inverted instances or not, and whether to match positive occurrences only, negative occurrences only or both.
  - □ Maximum positions (expected number of matches): MaxPositions, MaxInitialPositions. You can compel EasyMatch to consider more instances than needed at the coarse stage using the MaxInitialPositions parameter (this number is progressively reduced to reach MaxPositions in the final stage).
  - □ Minimum score (under which match is considered as false and is discarded): MinScore, InitialMinScore.
  - □ Sub-pixel accuracy: Interpolate. The accuracy with which the pattern is measured can be chosen (the less accurate, the faster). By default, the position parameters for each degree of freedom are computed with a precision of a pixel. Lower precision can be enforced. One tenth-of-a-pixel accuracy can be achieved.
  - □ Number of reduction steps: FinalReduction. Can speed up matching when coarse location is sufficient, range [0...NumReductions-1].
  - □ Non-square pixels: GetPixelDimensions, SetPixelDimensions. When images are acquired with non-square pixels, rotated objects appear skewed. Taking the pixel aspect ratio into account can compensate for this effect.
  - □ "Don't care" pixels (ignored for correlation score) below the DontCareThreshold value. When the pattern is inscribed in a rectangular ROI, some parts of the ROI can be ignored by setting the pixels values below a threshold level. The same feature can be used if parts of the template change from sample to sample.

0.5. Golden Template Validation (EChecker)

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

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

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

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

Training

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

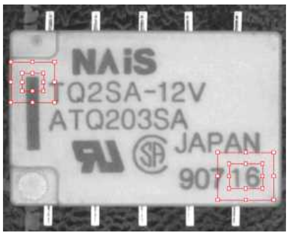
Two modes of operation are provided, depending on whether reference images are stored on disk or are acquired and processed on the fly.

These operations cannot be used during on-line operation.

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

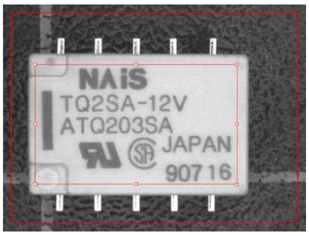
1. On the first reference image, one or two ROIs are placed to define the location patterns (fiducial marks or landmarks).
   These ROIs are surrounded by two others to define the possible movement freedom, and are used as search areas for pattern matching. ROIs can be placed interactively using dragging handles.
   EChecker manages the dragging operations.
   - Draw graphically renders the ROIs and handles.
   - HitTest detects the presence of the cursor over one of the handles.
   - Drag moves a handle.
   These functions operate on all three ROIs (pattern, search and inspected). You will quickly notice that:

- Dragging a pattern ROI causes the corresponding search ROI to adjust automatically so tolerances remain constant.
- Dragging a search ROI causes its size to adjust symmetrically with respect to the pattern.
- Adjusting a search area also sets the inspected ROI to the largest available space in the image.



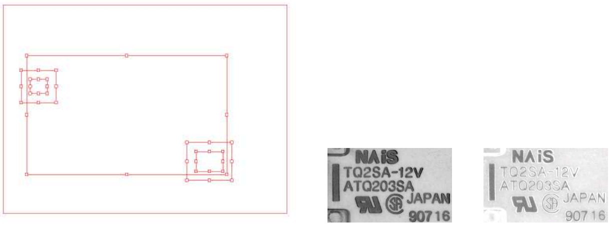
Pattern ROIs and search ROIs

- If the training images are on disk, the list of file names can be registered and used later to execute learning in a single command (*batch learning*, as opposed to *on-the-fly learning*).
- Another ROI is defined to delimit the area to be inspected. This area must only include pixels of the rigid part (that moves with the fiducial marks), and not the background.



Inspected ROI

- All images are processed and are averaged using Statistical training. It uses realignment to deal with displacement of the inspected part in the field of view, and gray-level normalization to deal with global illumination changes.
- Ideally at least 16 images should be used in the learning passes to create the low and high threshold images. The user can strengthen or loosen the acceptance intervals using a global tolerance parameter. Call `Learn(`ELearningMode_Ready`)`. Property `RelativeTolerance` adjusts the acceptance ranges.
- The model can be saved to a single file including all relevant information, i.e. placement of the various ROIs, fiducial pattern images, gray-level normalization parameters and the threshold images.



Components of an EChecker model

Check the reference images for reliability.

After the alignment ROIs have been set, they should be checked for reliability of fiducial location.

The best way is to load the training images, display them and locate the patterns in them, by means of member function Register. If location fails, different corrective actions can be taken, depending on the problem:

- The choice of pattern is poor. Define other ROIs with more stable contents.
- The search areas are too tight, so that occurrences of the pattern are found along edges. Enlarge the search area.
- The image is insufficiently representative (it has defects). It is better to withdraw it from the learning set.

NOTE
The first call to member Register invokes the Learn members of the alignment EasyMatch contexts, i.e. training on the patterns is achieved. Unless member Learn(ELearningMode_Reset) is called, these patterns will be used for all subsequent alignment operations. The first image to be used serves both as a reference for defining the alignment pattern and contrast measurement. It is called the *mother image*.

If the learning images are saved on disk: Every time a file is successfully loaded and is accepted as a reference image, the EChecker object can add the file path name to a list. Later, all files can be processed in a single command.

More learning

After the ROI placement and pattern learning have been performed (Register operations), training still requires two passes:

- The **"average" pass** is needed to compute an ideal, noise-free, image that reveals the central tendency of the part image.
  For each image, realign and normalize (Register). If the operation is successful (good pattern location), call Learn(ELearningMode_Average) for immediate processing (on-the-fly learning), or
  AddPathName for deferred processing (batch learning).
- The **"deviation" pass** measures variations around the average image.
  For each image, realign and normalize (Register). If the operation was successful, call
  Learn(ELearningMode_RmsDeviation) (enhances the large deviations), or
  Learn(ELearningMode_AbsDeviation) for immediate processing (more robust and is recommended in most circumstances).

In principle, the images shown during those two passes should be the same.
If you do not want to archive them, two distinct sets of images can be used (on-the-fly learning). These sets need not even be of the same size.

A learning set size of at least 16 images is recommended.

Alternatively, calling BatchLearn will perform the two required passes for all images added to the file list.

Adjust thresholding

Learn(ELearningMode_Ready). Property RelativeTolerance can be adjusted to adjust the acceptance ranges.

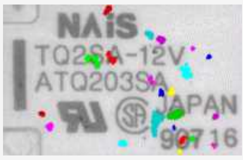
Inspection

Inspection is straightforward: the sample image is realigned with the model file and gray-level normalized.

NOTE
The inspected ROI must be positioned on the mother image. It can be interactively positioned in the same way the pattern and search areas are. This ROI can be set at the same time as the others. Changing the search tolerance will reset the inspected ROI to the largest available area.
EChecker can tune the global RelativeTolerance property at inspection time, thus changing lower and upper threshold images used by EasyObject (legacy).
This is the only EChecker parameter that can be changed after learning.

Inspected image, ...

... high and low threshold images, ...



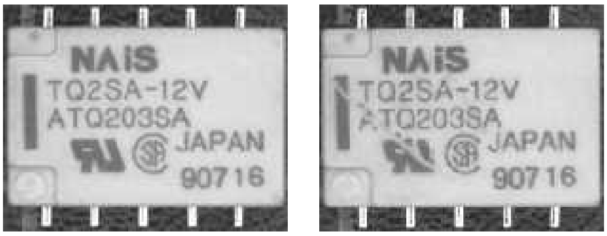
... and detected blobs.

The resulting image is passed to an ECodedImage object for blob analysis; which groups neighboring pixels to form blobs, discards the smaller blobs (usually noise), measures geometric characteristics (location, size, orientation, ...) and others, see EasyObject.

This image is then compared to the lower and upper reference images (a comparison is made pixel by pixel between the sample and the template to detect pixels that fall out of acceptance intervals). Defective pixels are handled by standard EasyObject functions.

Compare using EChecker

EChecker finds visible differences between templates and samples, and reports them in a defect map which highlights significant differences. EasyObject blob analysis tools can then locate the defects and qualify them in terms of extent, orientation, lightness and so on. It is ideally suited when inspected items are rigid (shape does not vary) and illumination is uniform, ensuring repeatable visual appearance of the image, so that point to point image comparison makes sense.

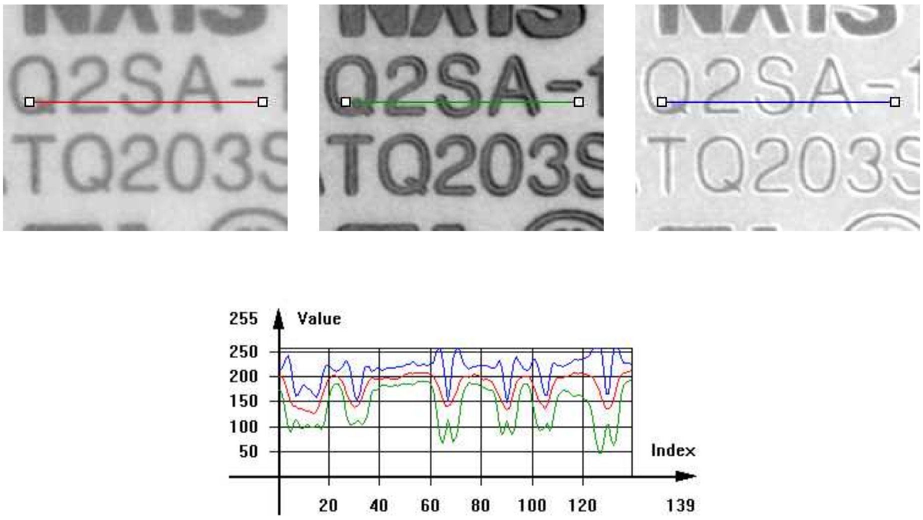


Reference image vs. sample image with defects

Statistical training

Grabbing several reference images optimizes assessment of normal gray-level variations and acceptance intervals:

- consecutive images of the same part without any change (static test) gives a gray-level distribution that corresponds to noise distribution.
- consecutive images of different defect-free parts reveals variations due to the parts.



Accepted gray-level ranges

Image Comparison

The best way to compare images is using `EChecker` to combine a set of images and determine the range of acceptable values at each pixel.

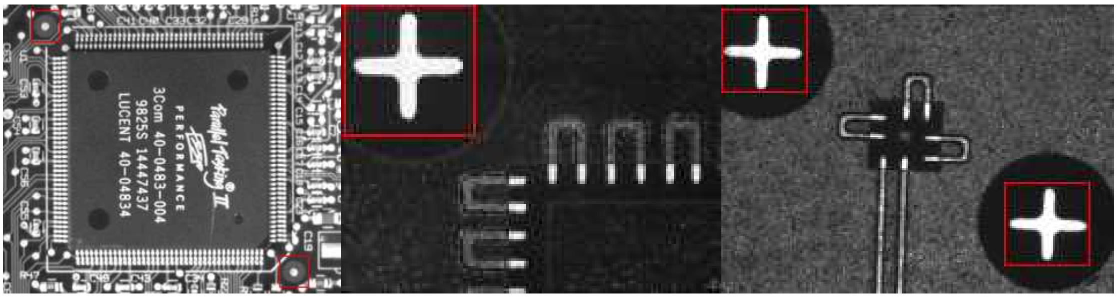
Images can also be compared by subtracting 2 images (using Arithmetic and Logic functions) to get an absolute value, then thresholding this difference to highlight the non-zero pixels where the images differ.

EChecker performs:

alignment

Pattern matching measures movement of the inspected part in terms of translation, rotation and/or scaling (as described in the EasyMatch chapter).
When the patterns have been located, the image is realigned so that the inspected part is brought to the same position as in the reference images.
A single matching pattern easily handles translation.
Two matching patterns provide better accuracy for rotation measurement.

When fiducial marks are available, they can be used as landmarks for accurate and repeatable location.
`EChecker` hosts one or two matching contexts internally.

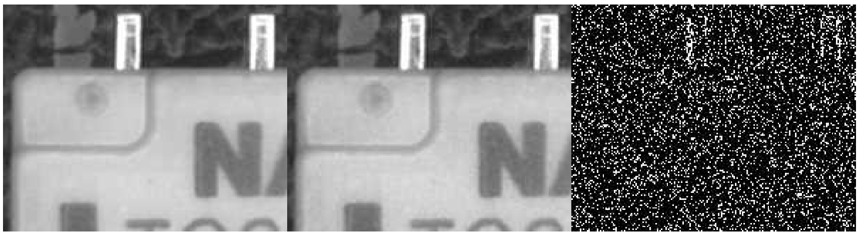


Realignment using fiducial marks

The alignment method has three shortcomings:

1) Unavoidable variations

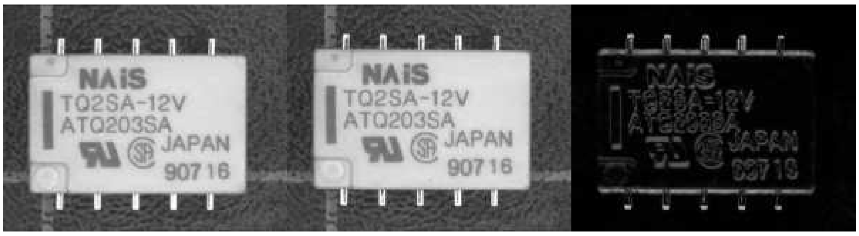
Tolerance must be introduced as noise can make identical images have slight variations in color or shape.



Comparing two noisy images for strict equality.
Black pixels of the result image are non equal pixels.

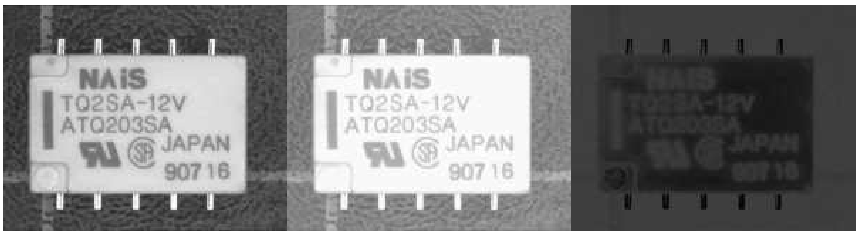
2) Placement of inspected parts is rarely repeatable

Slight misplacement means the compared pixels no longer correspond, the resulting effect is especially noticeable around edges.



Comparing misaligned images

3) Lighting cannot be separated from parts



Comparing images in different lighting conditions

PART V

TEXT AND CODE READING TOOLS

0.1. EasyBarCode - Reading Bar Codes

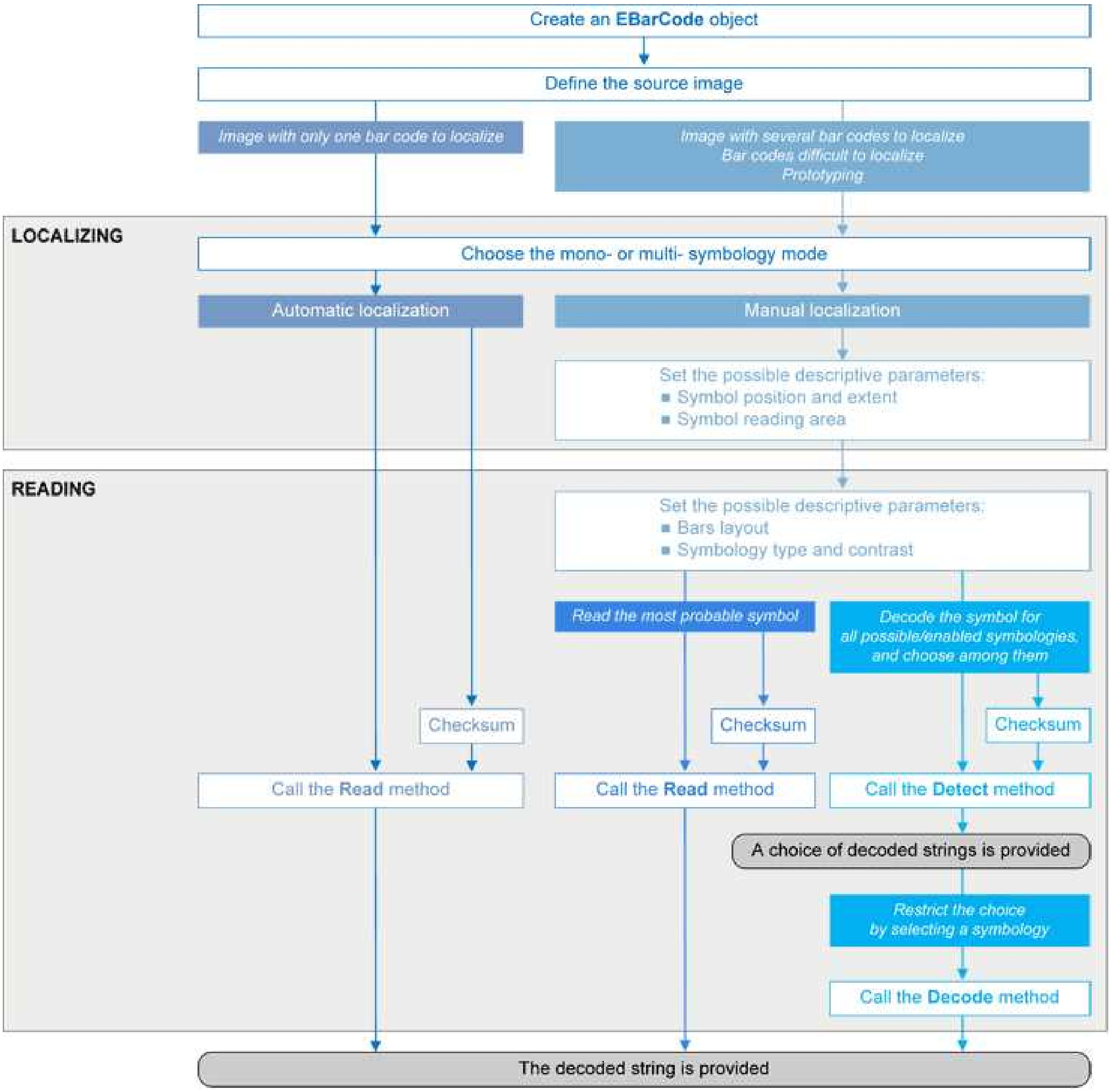
Reading Bar Codes

4 012345 987652

Bar code (EAN 13 symbology)

EasyBarCode can locate and read bar codes automatically.
Location can be performed manually for prototyping or when automatic mode results are unsatisfactory.

Workflow


Create an EBarCode object
Define the source image
Image with only one bar code to localize
Image with several bar codes to localize
Bar codes difficult to localize
Prototyping
LOCALIZING
Choose the mono- or multi- symbology mode
Automatic localization
Manual localization
Set the possible descriptive parameters:
■ Symbol position and extent
■ Symbol reading area
READING
Set the possible descriptive parameters:
■ Bars layout
■ Symbology type and contrast
Read the most probable symbol
Decode the symbol for all possible/enabled symbologies, and choose among them
Checksum
Checksum
Checksum
Call the Read method
Call the Read method
Call the Detect method
A choice of decoded strings is provided
Restrict the choice by selecting a symbology
Call the Decode method
The decoded string is provided

Bar code definition

A bar code is a 2D pattern of parallel bars and spaces of varying thickness that represents a character string. It is arranged according to an encoding convention (**symbology**) that specifies the character set and encoding rules.

- The bar code may be black ink on white background or inverted (white ink on black background).
- The bar code should be preceded and followed by a quiet zone of at least ten times the module width (smallest bar or space thickness).
- Bars should be surrounded below and above by a quiet zone of a few pixels.
- Bar and space widths must be greater than or equal to 2 pixels.

symbologies

A symbology defines the way a bar code is encoded.

Symbologies can be enabled in `StandardSymbologies` or `AdditionalSymbologies` parameters.

The standard symbologies are enabled by default:

- Code 39
- Code 128
- Code 2/5 5 Interleaved
- Codabar
- EAN 13*
- EAN 128
- MSI
- UPC A*
- UPC E

NOTE
* EAN 13 and UPC A only differ by the layout of surrounding digits.

Additional symbologies that are supported:

- ADS Anker
- Binary code
- Code 11
- Code 13
- Code 32
- Code 39 Extended (a super-set of Code 39)
- Code 39 Reduced (a subset of Code 39)
- Code 93
- Code 93 Extended
- Code 412 SEMI
- Code 2/5 3 Bars Datalogic
- Code 2/5 3 Bars Matrix
- Code 2/5 5 Bars IATA
- Code 2/5 5 Bars Industry
- Code 2/5 5 Compressed
- Code 2/5 5 Inverted
- Code BCD Matrix
- Code C.I.P
- Code STK

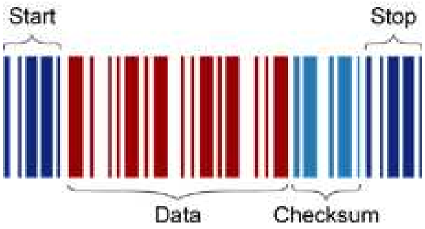
- EAN 8
- IBM Delta Distance A
- Plessey
- Telepen

Checksum

A checksum character enables the reader to check the barcode validity depending on the symbology:

- The checksum may be mandatory and must be checked by the reader.
- The checksum may be mandatory but may not need to be checked.
- The checksum and its verification may both be optional.

`VerifyChecksum` enables or disables (default) checksum verification.



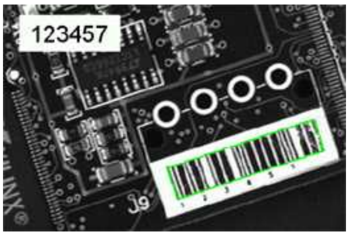
Bar code structure (Code 39)

Read a bar code

The **Automatic** mode reading algorithm locates a bar code in the field of view and Reads it.
If several bar codes are present, only one is located, like a straightforward hand-held bar code reader.

Before reading, the decoding symbologies must be specified in the StandardSymbologies, or AdditionalSymbologies properties.

Mono-symbology mode reads the bar code using the expected symbology type(s) and reports the encoded information (if readable) or the reason for failure (if not readable). There is only one interpretation for the character string.



Decoded bar code

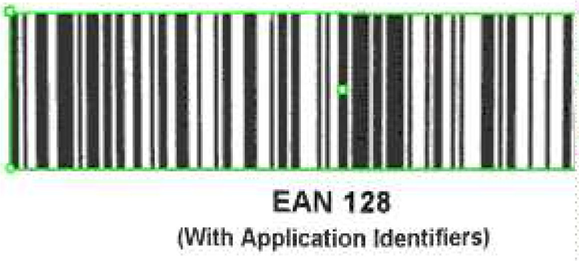
Note: When the bar code contains \0x00 characters, the `std::.string::.c_str` method should not be used (since C-strings are terminated by the \0x00 character). An iterator over the characters should be used instead of a C-string.

Advanced features

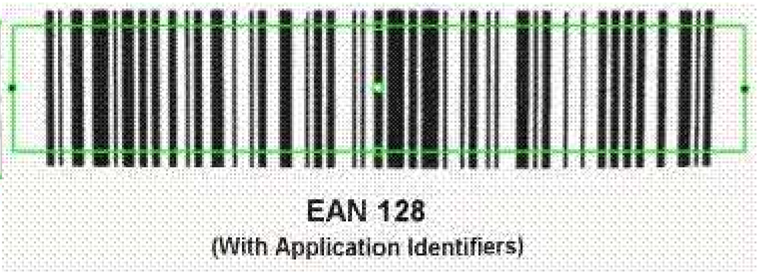
Locate and Read bar code manually

If automatic localization fails or for prototyping purposes, the user can provide the **bar code position** and **reading area** to manually locate the code.

- **Bar code position** can be provided graphically by a bounding box around the bar code or by its parameters. If several symbols appear in the image, they can be processed one after the other.
- The **reading area** of the bar code is the area that is read. It should be wider than the bar code bounding box width, and less high than the bar code bounding box height. It may also be rotated relatively to the bar code bounding box, to take into account slanting bars (Advanced mode!).



Bounding box — graphical appearance (manual location)



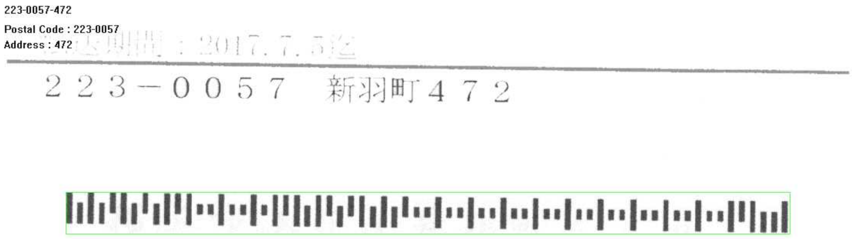
Reading area — graphical appearance (manual location)

Read all interpretations (multi-symbology mode)

Use `Detect` to report the number of possible symbologies in the `NumEnabledSymbologies` property, and list the data contents by decreasing likeliness.

Then call the `Decode` method in a loop, using `GetDecodedSymbology` to walk through the list of successful symbologies in decreasing order of likelihood.

Reading Mail Bar Codes



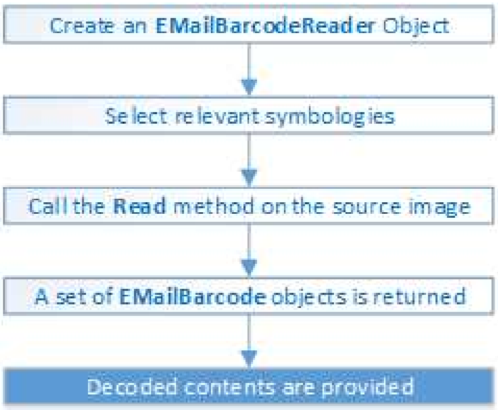
Mail bar code example

Specifications

The Mail Bar code Reader:

- Detects and decodes postal 4-state bar codes.
- Supports multiple mail bar codes in an image.
- Supports various symbologies.
- Supports the 4 main bar code orientations, with a tolerance of 3°.
- Detects bars that are at least 3 pixels wide.

Workflow

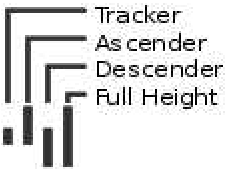

Create an EMailBarcodeReader Object
Select relevant symbologies
Call the Read method on the source image
A set of EMailBarcode objects is returned
Decoded contents are provided

4-state bar codes

A 4-state bar code is a special kind of bar code where data is encoded on the height and position of the bars rather than their width.

Each bar can have one of 4 possible states:

- □ Short and centered
- □ Medium and elevated
- □ Medium and lowered
- □ Full height



Mail bar code symbologies

The symbology of a mail bar code specifies how to decode the bar code and how to interpret its contents.

Every country uses its own flavor of mail bar code, or symbology. Some countries, like the US, even use multiple symbologies.

As of now, the Open eVision Mail Bar code Reader supports the following symbologies:

- □ US: PLANET, POSTNET and Intelligent Mail
- □ Japan: Japan Post

Mail bar code orientation

The Open eVision Mail Bar code Reader is designed to be used in mail-handling machines. As such it is optimized to handle the 4 main orientations you encounter in such machines:

- □ No Rotation: The mail barcode is horizontal and read from left to right
- □ Rotated 90° to the right: The mail barcode is vertical and read from top to bottom
- □ Rotated 90° to the left: The mail barcode is vertical and read from bottom to top
- □ Rotated 180°: The mail barcode is upside down, horizontal, and read from right to left.

For each of these orientations, an additional rotation of -3 to 3 degrees is allowed.

Checksum

Some symbologies specify the presence of a checksum in the bar code data.

This checksum is an additional character computed from all others encoded characters. It enables the reader to check the decoded character string coherence.

- The Mail Bar code Reader allows the user to verify or not the checksum for all enabled symbologies.
- By default, checksum is not controlled.
- To enable or disable checksum verification for all enabled symbologies, set the `ValidateChecksum` property.

Reading the mail bar codes in an image

To read all the mail barcodes in a given image:

1. Create an `EMailBarcodeReader` object.
2. Optionally, select the relevant symbologies using the `ExpectedSymbologies` property.

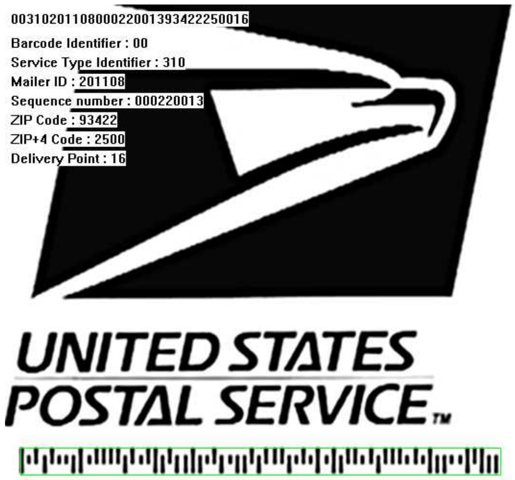
   By default, Mail Bar code Reader will consider all supported symbologies.
3. Optionally, select the relevant orientations using the `ExpectedOrientations` property.

   By default, Mail Bar code Reader will test all supported orientations.
4. Call `Read` on the source image or ROI.

Each mail bar code detected is returned as an `EMailBarcode` object.

5. Each `EMailBarcode` objects contains the following information:
   - □ The decoded string, using the `Text` property.

- □ The decoded string, split up in semantic parts, using the `ComponentStrings` property.
- □ The bar code orientation, using the `Orientation` property.
- □ The bar code position, using the `Position` property.



```
0031020110800022001393422250016
Barcode Identifier : 00
Service Type Identifier : 310
Mailer ID : 201108
Sequence number : 000220013
ZIP Code : 93422
ZIP+4 Code : 2500
Delivery Point : 16
```

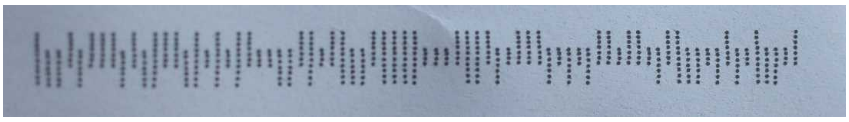
US Intelligent Mail bar code with highlighted position and decoded information

Advanced parameters

The advanced parameters of the `EMailBarcodeReader` object are:

- `EnableDottedBarcodes` activates the support for dotted barcodes (barcodes whose bars are printed with dots).

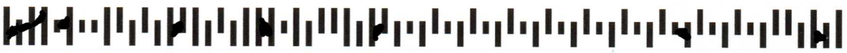
  By default, this property is set to false.



Dotted Mail Barcode

- `EnableClutteredBarcodes` activates the support for cluttered barcodes (barcodes in which some bars are connected).

  By default, this property is set to true.



Cluttered Mail Barcode

- `ValidateChecksum` activates the validation of the bar codes checksums, if present.

  By default, this property is set to false.

0.2. EasyMatrixCode - Reading Matrix Codes

EasyMatrixCode vs EasyMatrixCode2

Starting with release 2.5, Open eVision introduces a new data matrix code reading class, named `EasyMatrixCode2`.

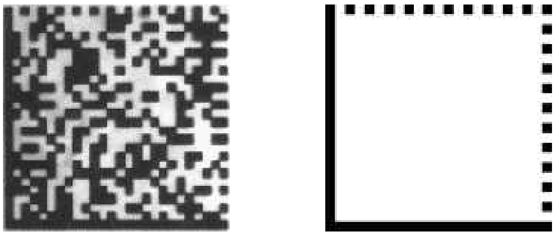
Compared to `EasyMatrixCode`, it offers the following benefits:

- Ability to read multiple data matrix codes in an image.
- Support for asynchronous processing.
- Improved consistency of reading and grading results.
- Improved consistency of processing time.
- Improved handling of deformed data matrix codes.

EasyMatrixCode

Specifications

Reference | Code Snippets



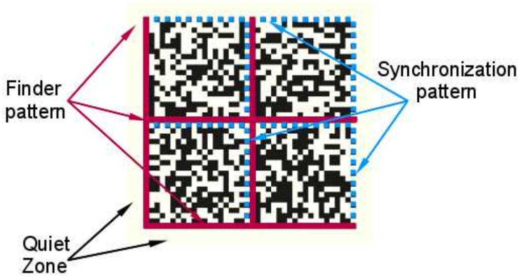
ECC 200, 26x26 cells data matrix code (left) and finder pattern (right)

In a single read operation, EasyMatrixCode locates, unscrambles, decodes, reads and grades the quality of grayscale 2D data matrix codes of any size, contrast, location and orientation (even viewed from the back on a transparent medium), providing they meet the following specifications:

- Minimum cell (= module) size: 3x3 pixels
- Maximum stretching ratio (ratio between cell width and height): 2
- Minimum quiet zone (blank zone around the matrix code) width: 3 pixels

Data Matrix Code Definition

- A data matrix code is a two-dimensional rectangular array of black and white cells which conveys a string of characters (digits, letters and special characters).
  - □ It is encoded to achieve maximum packing.
  - □ Each cell corresponds to a bit of information.
  - □ Additional redundant bits allow error correction for robust reading of degraded symbols.
- A data matrix code is located using the **Finder pattern**:
  - □ The bottom and left edges of a Data Matrix code contain only black cells.
  - □ The top and right edges have alternating cells.

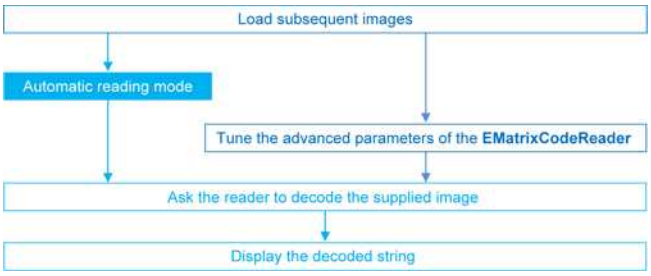


- A data matrix code is characterized by:
  - □ Its logical size (number of cells).
  - □ Its **encoding type**: ECC 000 (odd symbol sizes, deprecated) or ECC 200 (even symbol sizes)..

NOTE
The data matrix code definition is provided by ISO/IEC and approved as standard ISO/IEC 16022.

Workflow

Reference | Code Snippets


Load subsequent images
Automatic reading mode
Tune the advanced parameters of the EMatrixCodeReader
Ask the reader to decode the supplied image
Display the decoded string

Reading a Matrix Code

Reference | Code Snippets

You can read the matrix code in an image automatically, using the `Read` method.

This method returns an `EMatrixCode` instance that contains the following information about the found data matrix code:

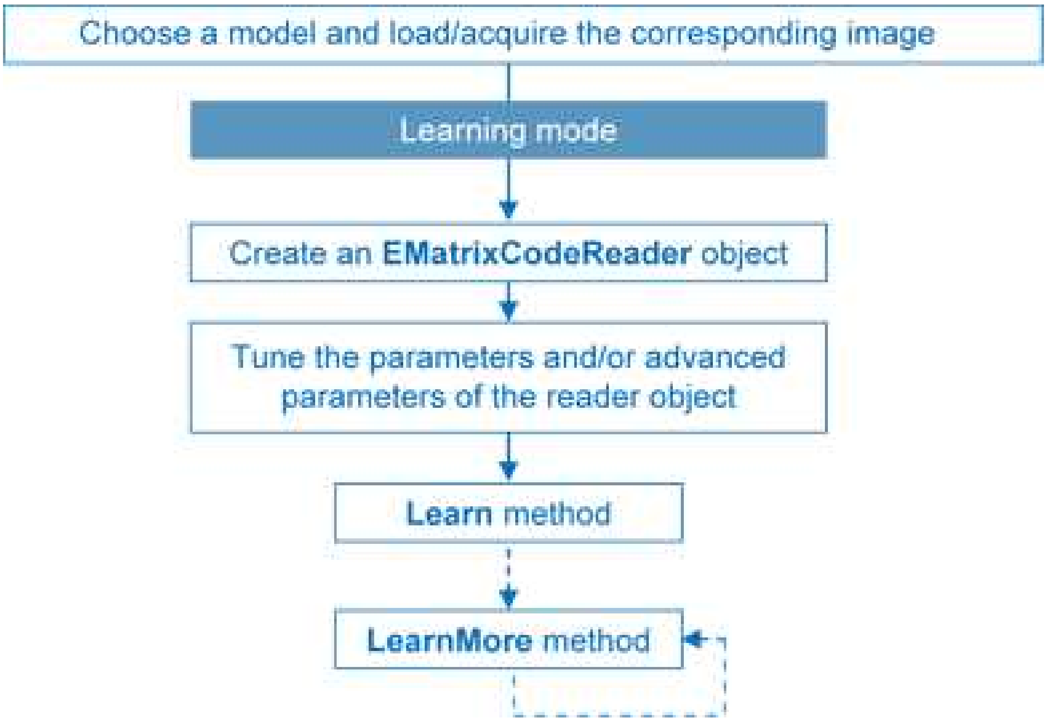
- Its decoded string,
- Its position in the image,
- Its logical size,
- Its encoding type,
- Its grading results,
- Methods to draw the data matrix code on the source image.

Learning a Matrix Code

Reference | Code Snippets

To search for specific features and speed up your processing, learn a Matrix code model.

Workflow


Choose a model and load/acquire the corresponding image
Learning mode
Create an EMatrixCodeReader object
Tune the parameters and/or advanced parameters of the reader object
Learn method
LearnMore method

1. Load the image of the matrix code you want to learn.
2. Learn the model:
    - Use the `Learn` method with `Contrast`, `Family`, `Flipping`, `Logical Size` parameters.
    - If you need to learn several matrix codes, use `LearnMore` and pass additional sample images.
    - Call `Learn` to replace `EMatrixCodeReader` parameters (calling `Learn` several times does not accumulate results, while `LearnMore` does).
3. Tune `search parameters` to be efficient and either:
    - Read only matrix codes that match a sample matrix code,
    - Or read only matrix codes that have the same properties (`Contrast`, `Family`, `Flipping`, `Logical Size`) as the learned one,
    - Or disregard a search parameter of the learned matrix code `SetLearnMaskElement`, for example to read only unflipped matrix codes. Just remove the default parameters, then add new ones.
4. Ask `EMatrixCodeReader` to decode the supplied image.
5. Display the `decoded string`.
6. Save the state of the reader object using `Save`.

Restoring the state of an EMatrixCodeReader

To restore the state of an `EMatrixCodeReader` and use it to read a matrix code:

1. `Load` an image.
2. Restore the reader state from the given file using `Load`.
3. `Read` the image.
4. Display the `decoded string`.

Computing the Print Quality

Reference | Code Snippets

To compute the print quality indicators as defined by BC11, ISO 15415, ISO/IEC TR 29158 (formerly known as AIM DPM-1-2006) and SEMI T10-0701 standards, retrieve the grades with the `GetIso15415GradingParameters`, `GetIso29158GradingParameters` and `GetSemiT10GradingParameters` accessors of the `EMatrixCode` class.

NOTE
The print quality of the matrix codes is computed during the `Read` operation, only if the `ComputeGrading` parameter is set to `true`.

Using GS1 Data Matrix Codes

Reference | Code Snippets

EasyMatrixCode is able to find and decode GS1-compliant data matrix codes.

The GS1 standard adds semantic identifiers to the contents of a data matrix code. These identifiers are interpreted in an easy and consistent way.

The structure of GS1-compliant content is as follows:

]d2[GS1]{Id1}{Value1}[GS1]{Id2}{Value2}...

where:

- "]d2" is the string identifying a GS1-compliant stream,
- [GS1] is the GS1 escape character (0x1d),
- {Id} is an application identifier,
- {Value} is the value associated to that identifier.

Example

The string:

]d2[GS1]11180112[GS1]15190101

is interpreted as follows:

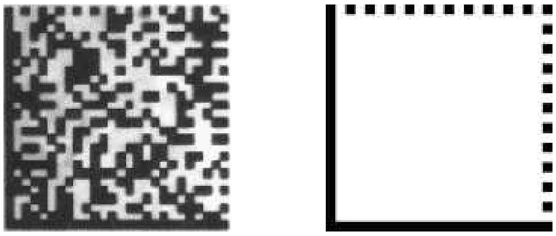
- It contains two GS1 parts: 11180112 and 15190101.
- The first (11180112) is composed of the identifier 11 and the value 180112, meaning that the product has a production date (the meaning of identifier 11) of January 12th, 2018.
- The second (15190101) is composed of the identifier 15 and the value 190101, meaning that the product has a best before date (the meaning of identifier 15) of January 1st, 2019.

TIP
For more information, see https://www.gs1.org/

EasyMatrixCode2

Specifications

Reference | Code Snippets



ECC 200, 26x26 cells data matrix code (left) and finder pattern (right)

In a single read operation, EasyMatrixCode2 locates, unscrambles, decodes, reads and grades the quality of grayscale 2D data matrix codes of any size, contrast, location and orientation (even viewed from the back on a transparent medium), providing they meet the following specifications:

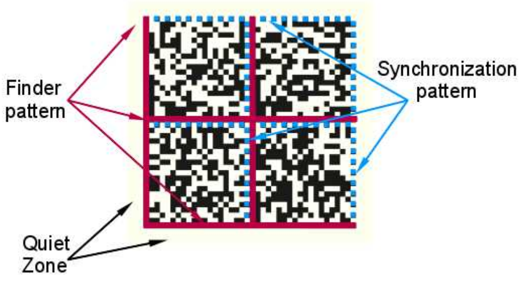
- □ Minimum cell (= module) size: 3x3 pixels
- □ Minimum quiet zone (blank zone around the matrix code) width: 1 pixel

All the functionality of EasyMatrixCode2 is available for testing in Open eVision Studio, except for the StopProcess method (for asynchronous processing).

> **NOTE**
> The relevant classes of the EasyMatrixCode2 library are stored in the name space “EasyMatrixCode2”.

Data Matrix Code Definition

- A data matrix code is a two-dimensional rectangular array of black and white cells which conveys a string of characters (digits, letters and special characters).
  - □ It is encoded to achieve maximum packing.
  - □ Each cell corresponds to a bit of information.
  - □ Additional redundant bits allow error correction for robust reading of degraded symbols.
- A data matrix code is located using the **Finder pattern**:
  - □ The bottom and left edges of a Data Matrix code contain only black cells.
  - □ The top and right edges have alternating cells.

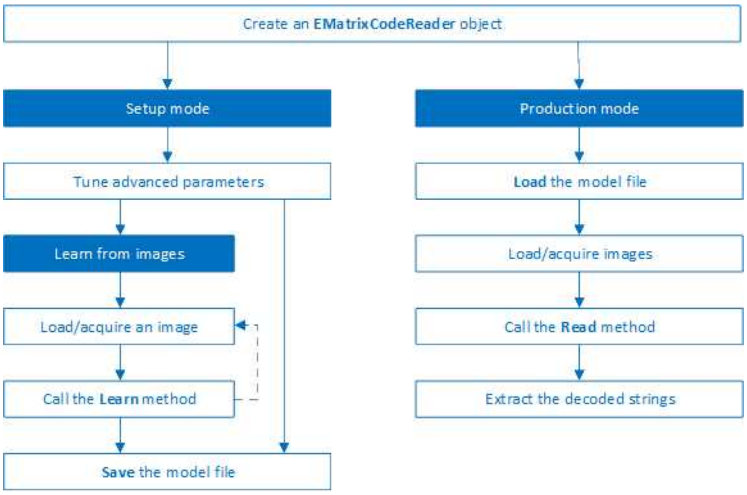


- A data matrix code is characterized by:
  - □ Its logical size (number of cells).
  - □ Its **encoding type**: ECC 000 (odd symbol sizes, deprecated) or ECC 200 (even symbol sizes)..

NOTE
The data matrix code definition is provided by ISO/IEC and approved as standard ISO/IEC 16022.

Workflow

Reference | Code Snippets



1. Load the image.
2. Read the data matrix codes in the image using `EMatrixCodeReader.Read()`.
3. Loop on found data matrix codes.
4. Display the decoded text.

Reading a Matrix Code

Reference | Code Snippets | dedicated code snippet: Reading Matrix Codes from an Image

You can read the matrix code in an image automatically as follows:

a. Create an `EMatrixCodeReader` object.

b. Call the `Read` method to detect and decode the matrix codes in the image.

c. Call the `GetReadResults` accessor to retrieve the decoded `EMatrixCode` instances.

The `EMatrixCode` instances contain the following information for each found data matrix code:

- □ Its decoded string,
- □ Its position in the image,
- □ Its logical size,
- □ Its encoding type,
- □ Its grading results,
- □ Methods to draw the data matrix code on the source image.

Learning a Matrix Code

Reference | Code Snippets | dedicated code snippet: Reading with Prior Learning

To improve the processing times of the `Read` method, learn a matrix code model from representative images as follows:

1. Load the image of the matrix code you want to learn from.
2. Call the `Learn` method to learn from the image.
3. Repeat with additional images if necessary.
4. Save the `EMatrixCodeReader` state to the disk with the `Save` method.

The `Learn` method re-orders the internal processing structure used to detect and decode the matrix codes in such a way that the learned codes are found faster.

> **TIP**
> The user-defined advanced parameters (`MaxNumCodes`, `Timeout`, `ReadMode` and `ComputeGrading`) are not affected by the `Learn` method .

If the `Learn` method is not able to detect any code in the image, it throws an exception.

> **TIP**
> The internal processing structure is not affected in this situation.

Restoring the state of an EMatrixCodeReader

- To restore a previously saved `EMatrixCodeReader` state , call the `Load` method.
- To restore the default state of an `EMatrixCodeReader` instance, call the `ResetLearning` method.

Computing the Print Quality

Reference | Code Snippets | dedicated code snippet: Inspecting Print Quality Grades

To compute the print quality indicators as defined by BC11, ISO 15415, ISO/IEC TR 29158 (formerly known as AIM DPM-1-2006) and SEMI T10-0701 standards, retrieve the grades with the `GetIso15415GradingParameters`, `GetIso29158GradingParameters` and `GetSemiT10GradingParameters` accessors of the `EMatrixCode` class.

NOTE
The print quality of the matrix codes is computed during the `Read` operation, only if the `ComputeGrading` parameter is set to `true`.

Using GS1 Data Matrix Codes

Reference | Code Snippets

EasyMatrixCode2 is able to find and decode GS1-compliant data matrix codes.

The GS1 standard adds semantic identifiers to the contents of a data matrix code. These identifiers are interpreted in an easy and consistent way.

The structure of GS1-compliant content is as follows:

]d2[GS1]{Id1}{Value1}[GS1]{Id2}{Value2}…

where:

- "]d2" is the string identifying a GS1-compliant stream,
- [GS1] is the GS1 escape character (0x1d),
- {Id} is an application identifier,
- {Value} is the value associated to that identifier.

Example

The string:

]d2[GS1]11180112[GS1]15190101

is interpreted as follows:

- It contains two GS1 parts: 11180112 and 15190101.
- The first (11180112) is composed of the identifier 11 and the value 180112, meaning that the product has a production date (the meaning of identifier 11) of January 12th, 2018.
- The second (15190101) is composed of the identifier 15 and the value 190101, meaning that the product has a best before date (the meaning of identifier 15) of January 1st, 2019.

TIP
For more information, see https://www.gs1.org/

Asynchronous Processing

Reference | Code Snippets

EasyMatrixCode2 supports asynchronous processing. This means that you can launch multiple processing threads in parallel, each reading the matrix codes in its own image.

From the main thread, to manually stop the `Read` method in any of these processing threads at any time, use the `StopProcess` method.

When you manually stop the `Read` method:

- □ The search for matrix codes stops immediately, whether it has found matrix codes in the image or not.
- □ To retrieve all matrix codes found before the manual stop, use the `GetReadResults` accessor.

Advanced Parameters

Reference | Code Snippets

Tune the following parameters to optimize the performance of **EasyMatrixCode2**.

- The `MaxNumCodes` parameter:
  - □ Tells the `EMatrixCode2Reader` the number of codes that can be in the image.
  - □ Affects the computational time of the `Read` method.
  - □ Is set to 1 by default. This means that the `EMatrixCodeReader` only detects a single matrix code per image.
  - □ If set to 0, tells the `EMatrixCodeReader` to find as many codes as possible in the image.
- The `Timeout` parameter:
  - □ Limits the amount of time that the `Read` and `Learn` methods may take to process a single image.
  - □ Is defined in microseconds.
  - □ Is set, by default, to a value that exceeds one hour.
- The `ReadMode` parameter affects the behavior of the `Read` method:
  - □ The setting `EReadMode_Speed` results in the shortest processing times and the `Read` method stops as soon as one of the following is true:
- The method has found `MaxNumCodes` codes.
- The method reaches the `Timeout` time limit.
- The `Read` process is completely finished.
  - □ The setting `EReadMode_Quality` results in the best grading results and the `Read` method keeps trying to improve its detection until one of the following is true:
- The method reaches the `Timeout` time limit.
- The `Read` process is completely finished.

- The ComputeGrading parameter:
  - Determines if the Read method computes the grading properties of the EMatrixCode object.
  - Is set to False by default.

After the tuning:

  - Use the Save method to store the state of the EMatrixCodeReader on the disk.
  - Use the Load method, at any time, to restore the saved state.

TIP
The Save and Load methods also store the effects of Learning.

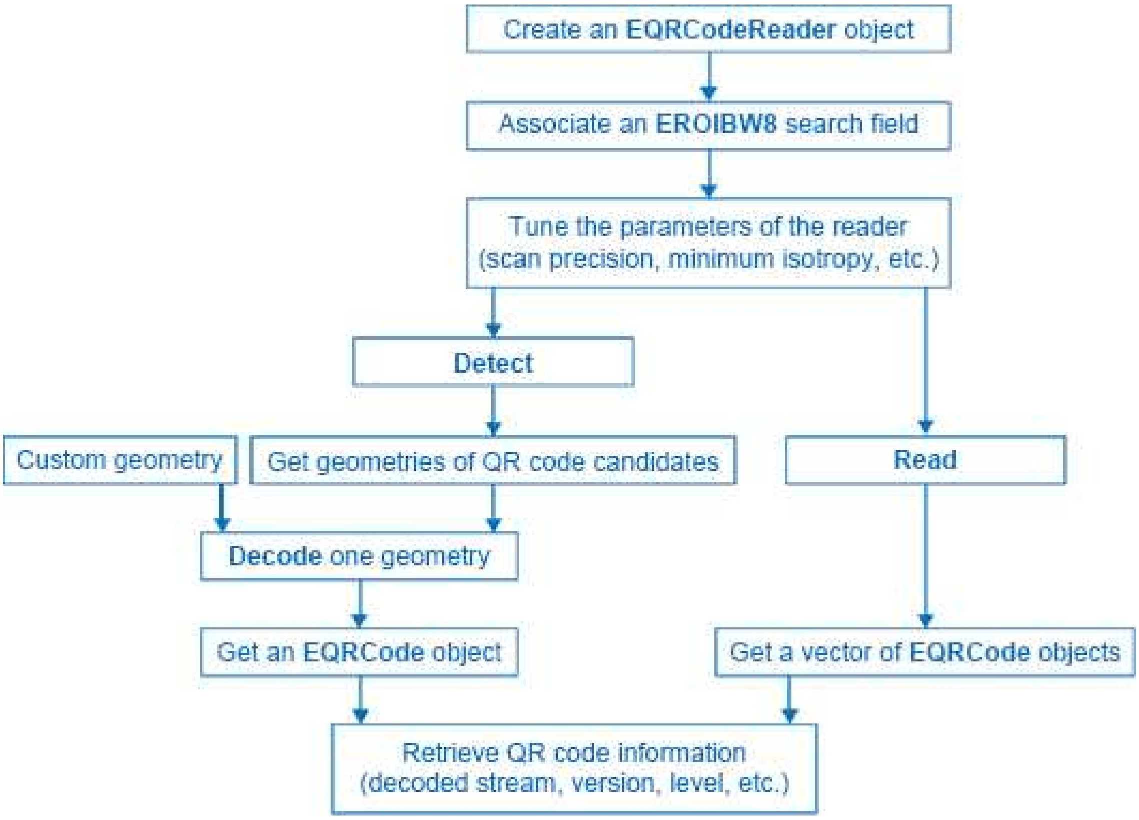
0.3. EasyQRCode - Reading QR Codes

Reading QR Codes

EasyQRCode detects QR (Quick Response) codes in an image, decodes them, and returns their data.

Error detection and correction algorithms ensure that poorly-printed or distorted QR codes can still be read correctly.

Workflow


Create an EQRCodeReader object
Associate an EROIBW8 search field
Tune the parameters of the reader (scan precision, minimum isotropy, etc.)
Detect
Custom geometry
Get geometries of QR code candidates
Read
Decode one geometry
Get an EQRCode object
Get a vector of EQRCode objects
Retrieve QR code information (decoded stream, version, level, etc.)

QR code definition

A QR code is a square array of dark and light dots. One dot (or "*module*") represents one bit of information.

QR codes contain various types of data and can be different models, versions, and levels. They always contain a message, metadata about alignment, size, format, and error correction bits. They comply with the international standard ISO/IEC 18004 (1, 2 and 2005).

QR code structure

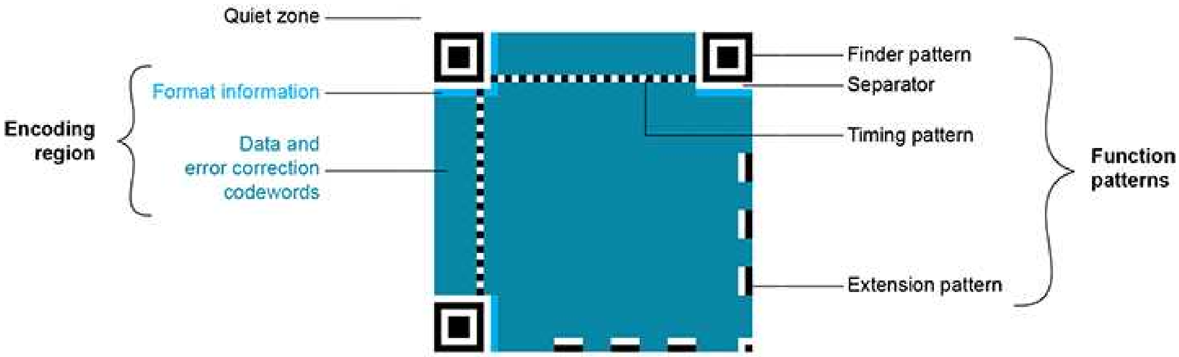
The QR code symbol consists of an *encoding region*, containing data and error correction codewords, and of *function patterns*, containing symbol metadata and position data.

A QR code must be structured with the following elements:

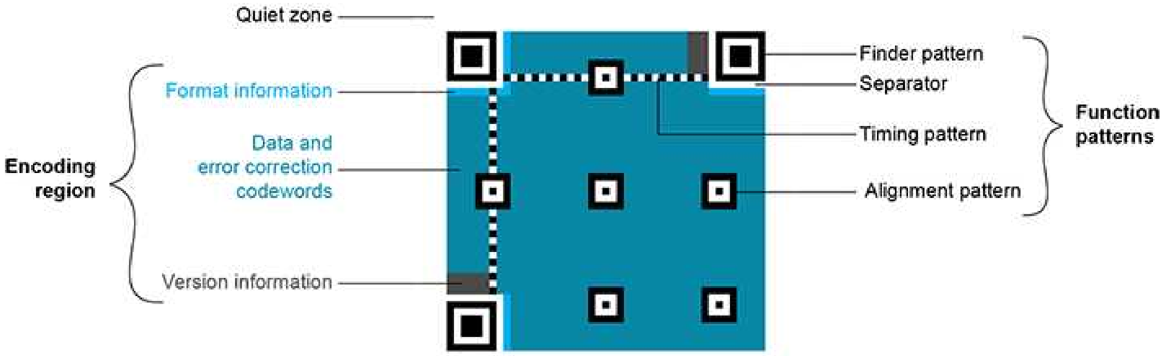
- *Quiet zone*: blank margin around the QR code
- *Finder patterns*: recognizable zones identifying a QR code
- *Extension patterns*: markers for the alignment of the QR code (model 1)
- *Alignment patterns*: markers for the alignment of the QR code (models 2 and 2005)
- *Timing Patterns*: data giving the module size (in pixels)
- *Format information*: zones providing the QR code level

- □ *Version information*: data giving the QR code size, for instance 25 x 25 modules (models 2 and 2005)
- □ *Data contents and error correction codewords*: the primary information carried by the symbol, with additional information for error correction

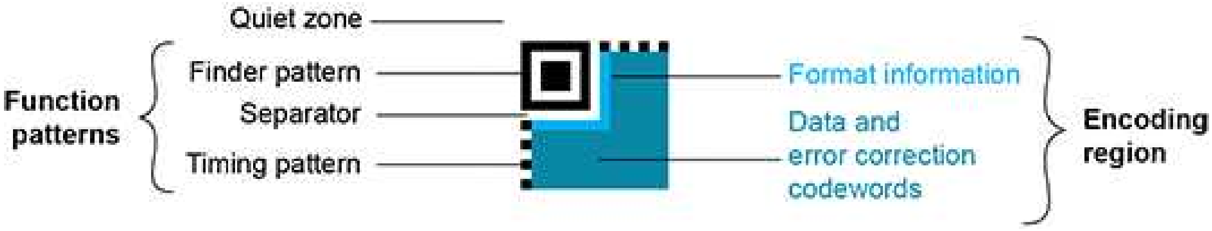
Variants of this structure exist, according to the model, format, or version of the QR code. For instance, model 1 QR codes do not feature alignment patterns but extension patterns. Micro QR codes include only one finder pattern, and no alignment pattern.



Structure of a model 1 QR code symbol



Structure of a QR code 2005 symbol



Structure of a Micro QR code symbol

QR code subtypes

A QR code can be one of the following subtypes:

- *Basic*: the default subtype.
- *ECI* (Extended Channel Interpretation): the ECI subtype provides a consistent method to embed interpretation information of data in the QR code. The ECI protocol is defined in the AIM Inc. International Technical Specification.
- *GS1*: the data contained in the QR code are formatted in accordance with the GS1 General Specification.
- *AIM*: the data contained in the QR code are formatted in accordance with a specific industry application previously agreed with AIM Inc. The application indicator value is embedded in the QR code data.

Data types

The QR code data can be any mix of these types:

- *Numeric data* (0-9)
- *Alphanumeric data* (0-9, A-Z, /,$, %...)
- *Byte data* (possibly ECI-encoded)
- *Kanji characters*

Byte data interpretation

In a QR code, the byte data can represent any information. Their interpretation depends on the subtype of the QR code:

- Basic subtype:
  - If some byte data are present in the QR code, you need to know how to interpret them.
  - Use the `EByteInterpretationMode` enum to select the corresponding byte interpretation mode (see the retrieving decoded data section in "Detecting and Decoding QR Codes" on page 173 for more details).
- ECI-encoded byte data:
  - The ECI subtype provides an ECI table indicator.
  - This indicator defines the character set to use to interpret the byte data.
  - **EasyQRCode** currently supports the UTF8 conversion table (ECI table indicator 26).

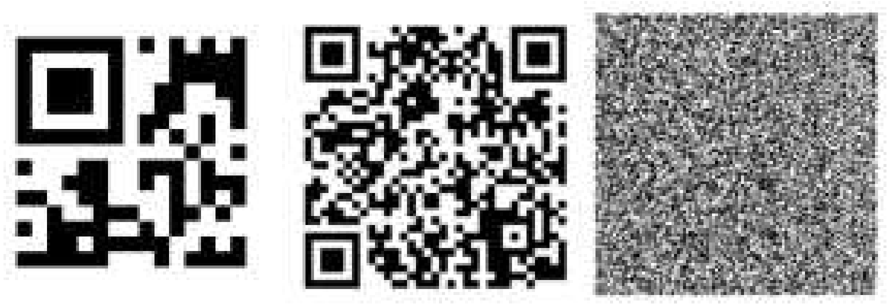
Models (Standards)

- *Model 1*: original QR code international standard, with versions ranging from 1 to 14. Note that the "version" of a QR code is the symbol size (in number of modules). It does not relate to the version of the standard, which is called the "model".
- *Model 2*: improvement of model 1. It provides versions from 1 to 40. It defines alignment patterns to improve reading of distorted QR codes, or QR codes printed on curved surfaces.
- *Model 2005*: improvement of model 2, including white-on-black QR codes, and mirror symbol orientation.

- *Micro QR codes*: (not yet supported) smaller QR codes, from version *M1* to version *M4*. They have been introduced to save printing space.

Versions (Symbol Size)

- *QR codes*: from version *1* (21 x 21 modules) to version *40* (177 x 177 modules), with an increment of +4 x +4 modules (version 2: 25 x 25 modules, version 3: 29 x 29 modules, ..., version 39: 173 x 173 modules).
- *Micro QR codes*: (not yet supported) version *M1* (11 x 11 modules), version *M2* (13 x 13 modules), version *M3* (15 x 15 modules), version *M4* (17 x 17 modules).



Examples of QR codes
From left to right:
Micro QR code, version M3, 15 x 15 modules,
Model 2 QR code, version 4, 33 x 33 modules, 67-114 characters,
Model 2 QR code, version 40, 177 x 177 modules, 1852-4296 characters

Levels (Error Correction)

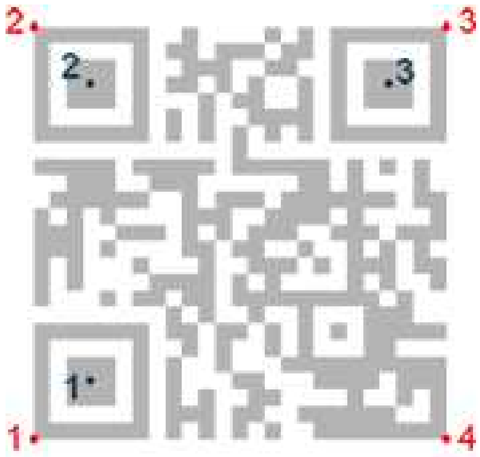
QR codes contain error correction data. The standard offers the following levels of error correction:

- *L*: (low) about 7% of codewords can be restored
- *M*: (medium) 15%
- *Q*: (quality) 25%
- *H*: (high) 30% (not available for Micro QR codes)

QR code geometry

When the QR code reader finds an array of dots that could match a QR code, it returns the "geometry" of this QR code candidate.

A `QR code geometry` is a set of points. It contains the coordinates of the `corners` of the QR code `quadrangle` (bottom left, top left, top right, bottom right), and the coordinates of the `finder pattern centers` (bottom left, top left, top right).



QR code geometry

Read a QR code

Reading a QR code returns information about `QR codes` for which `detection` and `decoding` were successful.

This is equivalent to detecting and decoding all QR codes in the given search field (see advanced features).

Detecting and Decoding QR Codes

Detect a QR code

1. Set a `search field` on an `EROIBW8` image.
2. If needed, tune the parameters to restrict the number of operations to process.
3. The QR code reader scans the image and searches for 3 finder patterns that could match a QR code, with the following requirements:
   - Minimum quiet zone (blank zone around the QR code) width: 3 pixels.
   - Minimum module size: 3 x 3 pixels.
   - Minimum isotropy: 0.5.
   - Maximum corner deformation: 15° (corner angles can range from 75° to 105°).
4. The reader returns QR code candidates, or the result of a `detection`, as a vector of geometries.

   The QR code reader uses the `gravity center` of the `QR code geometries` to sort this vector in line then columns order starting from the top left corner of the image.

Decode a QR code

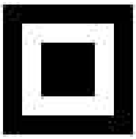
1. The QR code reader decodes a QR candidate and returns the `QR code`: `model`, `version`, `level`, `geometry` and the `decoded data` as described below.
2. The reader can report the amount of `unused error correction`.
   - Close to 1, very few errors were corrected when decoding the data. The decoding is highly reliable, and the QR code is of good quality.
   - Close to 0, many errors were corrected when decoding the data. The decoding is reliable, but the QR code quality is poor.
   - -1, error correction failed. Decoding was not performed.

Tune the search parameters

`Scan precision`: You can change the scan precision to scan the search field with:

- A fine precision (recommended for small QR codes)
- A coarse precision (recommended for medium to large QR codes)

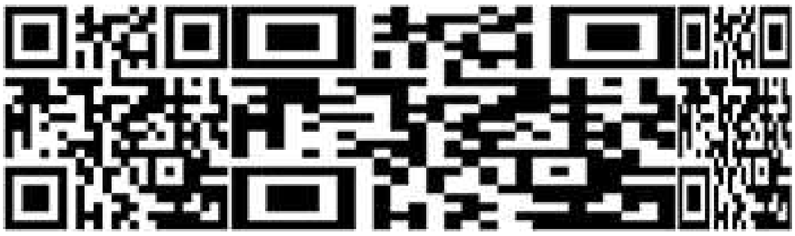
`Minimum score`: The QR code reader searches for this QR code finder pattern:



- A perfect match returns a pattern finder score of 1.
- Less accurate matches return lower scores.
- The minimum score allowed by default is 0.65 - you can tune this.

`Minimum isotropy`: The isotropy of a QR code represents its rectangular deformation.

- Perfectly square QR codes have an isotropy of 1 (short side divided by long side, whether the rectangle is vertical or horizontal).
- EasyQRCode can detect rectangle QR codes with an isotropy down to 0.5.
- The default `minimum isotropy` is 0.8, it can be tuned from 0 to 1.



Square and rectangular QR codes (isotropy = 1, 0.5, and 0.5 from left to right)

`Model` and version: The QR code reader searches for QR codes of all models, and all versions.

- □ You can shorten the process by specifying the QR code `model(s)` and a range of versions (from `1` to `40`) to be searched for.

Retrieve the decoded data

Retrieving methods

To retrieve the decoded data, you can (in growing complexity order):

1. Use the `GetDecodedString` method of an `EQRCode` object.
    - □ This method returns an UTF-8 formatted string that contains the concatenated data of the QR code.
    - □ It can take an `EByteInterpretationMode` as argument.
2. Use the `GetDecodedString` method of the `EQRCodeDecodedStreamPart` objects.
    - □ This method is called on a part and returns an UTF-8 formatted string that contains the data of this part.
    - □ It can take an `EByteInterpretationMode` as argument.
    - □ Concatenate the decoded string of each part.
3. Use the `GetDecodedData` method of the `EQRCodeDecodedStreamPart` objects.
    - □ This method is called on a part and returns a vector of bytes that contains the data of this part.
    - □ Interpret the data according to the `coding mode` of the QR code and the `encoding` of each part.
    - □ Concatenate the interpreted data of each part.

Interpreting the encoded data

The QR code data can be encoded in either alphanumeric, numeric or byte modes. If a QR code contains bytes, the interpretation mode of these bytes can be embedded in the QR code through the ECI protocol or you must specify or know it.

Use the dedicated `EByteInterpretationMode` for this purpose:

- `EByteInterpretationMode_Hexadecimal`
    - □ Converts all bytes to their hexadecimal values (2 characters per byte).
    - □ The escape character 0xEFBFBD surrounds the converted byte parts.
    - □ This mode overrides the ECI table indicator if it is present.

- EByteInterpretationMode_UTF8
  - □ Converts all bytes to UTF-8 if possible.
  - □ The GetDecodedString method throws an EException if the data are not UTF-8 compatible.
- EByteInterpretationMode_Auto
  - □ Converts all bytes in the best possible way following the ECI protocol.

The decoded string returns the concatenated data of the QR code in UTF-8 format:

- □ If bytes are present in the QR code data without ECI, specify the byte interpretation mode when you call the GetDecodedString method.
- □ If bytes are present in the QR code data with ECI encoding, use the corresponding byte interpretation table (currently, only table ECI 26: UTF-8).
- □ The hexadecimal byte interpretation mode does not throw an exception and returns all bytes parts present in the data in their hexadecimal form (2 characters per byte) surrounded by the 0xEFBFBD escape character.
- □ See the code snippet Retrieving Information of a QR Code.

The decoded stream class consists of:

- □ A coding mode (basic, ECI, FNC1/GS1 or FNC1/AIM).
- □ An application indicator (if the coding mode is FNC1/AIM, otherwise 0).

The decoded data:

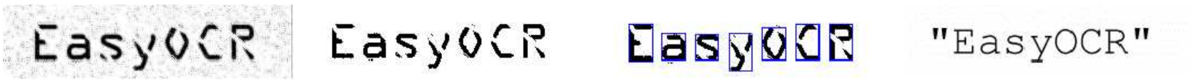
- □ Is accessible from each part of the decoded stream.
- □ Is interpreted according to its encoding (numeric, alphanumeric, byte or Kanji) and the ECI table indicator (if the coding mode is ECI, otherwise -1).
- □ Can be the raw bit stream (the bit data after unmasking and error correction, but before decoding as a vector of bytes).
- □ Can be the corresponding decoded string (specify a byte interpretation mode if the encoding is byte without ECI coding mode or if the ECI table is not supported).
- □ See also the code snippet Retrieving the Decoded Data (Advanced).

0.4. EasyOCR - Reading Texts

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

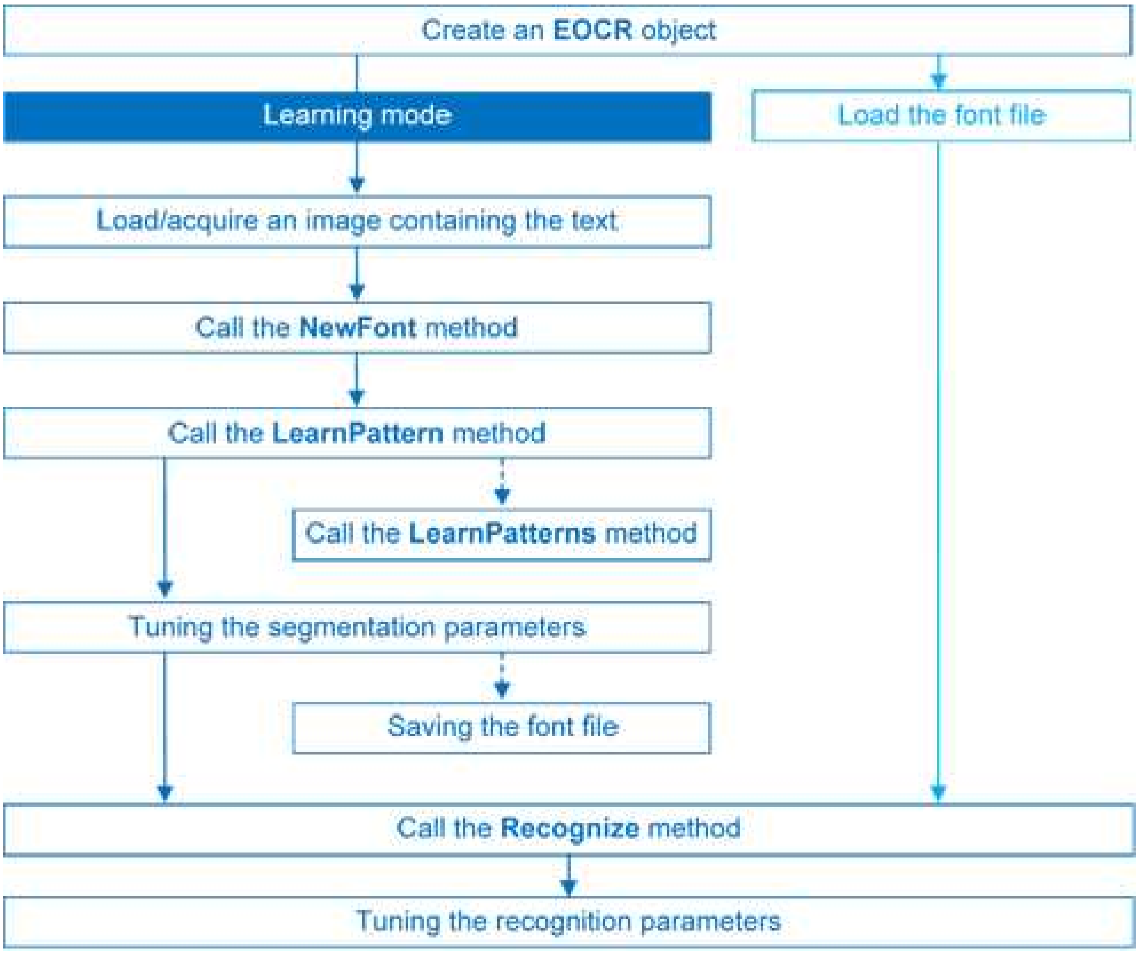
It uses font files (pre-defined OCR-A, OCR-B and Semi standard fonts, or other learned fonts) with a template matching algorithm that can recognize even badly printed, broken or connected characters of any size.

There are 4 steps to recognizing characters:



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

Workflow


Create an EOCR object
Learning mode
Load the font file
Load/acquire an image containing the text
Call the NewFont method
Call the LearnPattern method
Call the LearnPatterns method
Tuning the segmentation parameters
Saving the font file
Call the Recognize method
Tuning the recognition parameters

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:

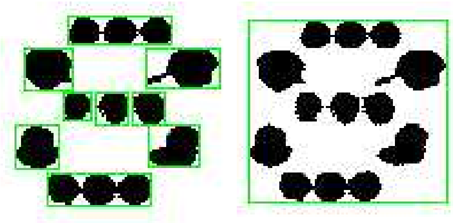
1. NewFont clears the current font.
2. LearnPattern or LearnPatterns adds the patterns from the source image to the font. Patterns are ordered by their index value, as assigned by the FindAllChars process. The patterns in a font are stored as a small array of pixels, by default 5 pixels wide and 9 pixels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

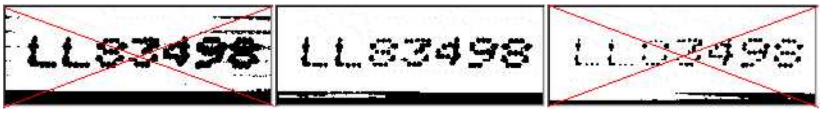
2. Filters remove very large and very small unwanted features.

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.

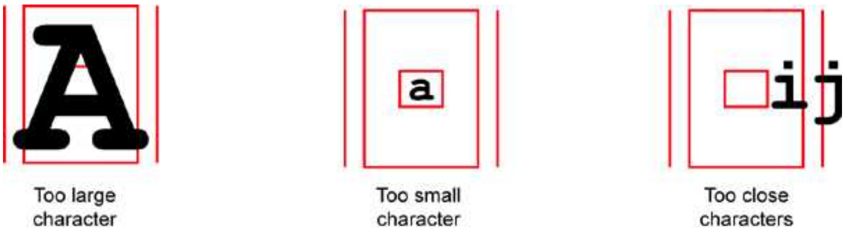
Steps 2 to 4 can be repeated at will to process other images or ROIs. The Recognize method can be used as well.

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.

0.5. EasyOCR2 - Reading Texts (Improved)

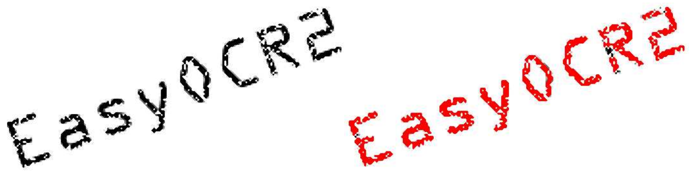
Reference | Code Snippets

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.

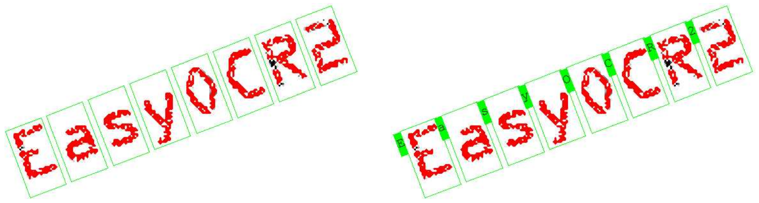
It uses an innovative segmentation method to detect blobs in the image, and then places textboxes over the detected blobs following a user-defined topology (number of lines, words and characters in the text). These methods support text rotation up to 360 degrees, can handle non-uniform illumination,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

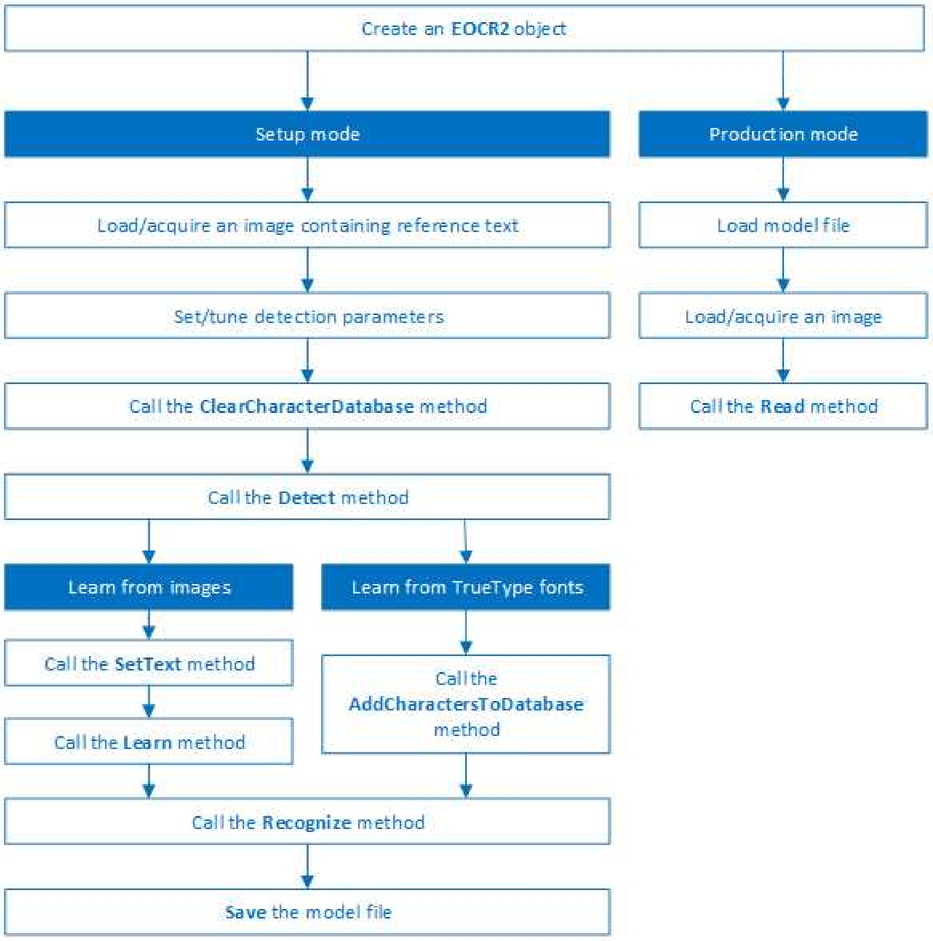
EasyOCR2 vs EasyOCR

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:

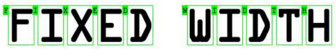
a. Set the required detection parameters.

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

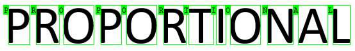
c. Call Detect to extract the text from the image.

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.

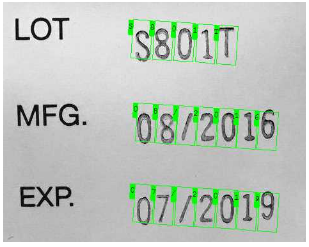
See example in code snippet: Detecting Characters



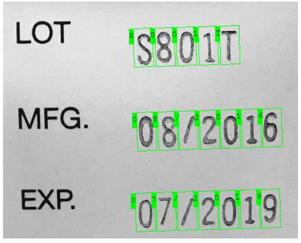
An example of a fixed-width font, processed with the detectionMethod ‘EOCR2DetectionMethod_FixedWidth’



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:

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

  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:

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

- □ 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

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

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.

PART VI

DEEP LEARNING INSPECTION TOOLS

0.1. Deep Learning Tools - Inspecting Images with Deep Learning

Purpose and Workflow

Tools

The deep learning tools are based on deep convolutional neural networks (CNNs):

- **EasyClassify** classifies images into a predefined set of classes. Use this tool to identify a product in an image or to detect if the product is good or defective.
- **EasySegment** detects and segments defects in images. This tool works in an unsupervised way. This means that it is trained on good products only.

  As you build only a model of what a good product is and not a model of what a defective product is:

  - □ The advantage is that the tool can detect and segment defects that are not in your dataset or that are unexpected.
  - □ The drawback is that the type of defects that the tool can detect and segment is more limited than when you build an explicit model of the defects.

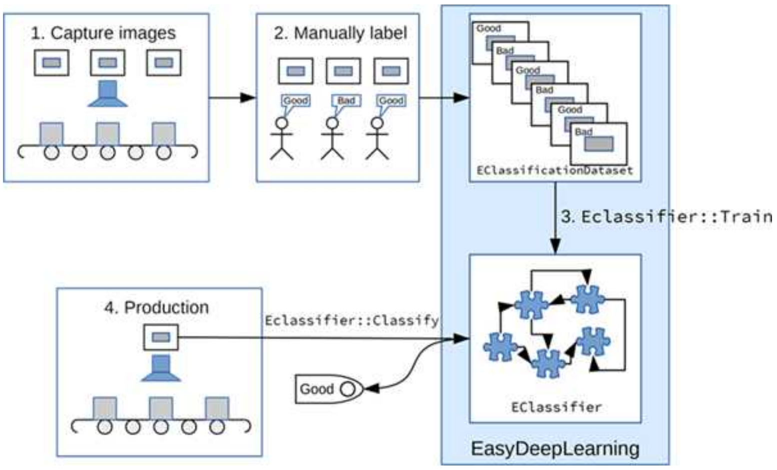
By opposition to traditional machine vision techniques, the deep learning tools do not require an explicit model of what to recognize and/or segment inside an image. Instead, they learn this model from a set of example images. Thus the deep learning tools can solve machine vision problems where an explicit model is too complex to build.

Specifications

	EasyClassify	EasySegment (unsupervised mode)
Minimum image size	128 × 128	64 × 64
Maximum image size	1024 × 1024	10 000 × 10 000
Best image size	256 × 256 - 600 × 600	n.a.
Number of channels	1 or 3 (grayscale and color images)	
Bit depth	8 bits, 16 bits	
Number of labels	2 - 1000	2 (good and defective)
Minimum number of images per label	2	1 for the good label 0 for the defective label
Supported image format	bmp, png, jpeg, j2k, tiff	

TIP
To accelerate computations, we strongly recommend running the deep learning tools on a recent NVIDIA GPU. Refer to the section "Hardware Support (CPU/GPU)" on page 195 for installing the required NVIDIA CUDA and deep learning library.

Workflow



To create an application based on the deep learning tools:

1. Capture a dataset of images representative of the problem you want to solve.
   - The capture conditions must be as close as possible of the production conditions.
   - Preferably, all images should have the same resolution.
   - The number of images needed to obtain a good performance depends on the complexity of the task and the tool used.
     Please refer to the specifications of each tool for the constraints on the resolution and the number of images.
2. Manually label the images in the dataset with the different categories you want to recognize.

   These categories depend on the tool:
   - **EasyClassify**:
     - Each image must correspond to one and only one category.
     - There must be at least 2 categories.
   - **EasySegment**:
     - A single category for images of good samples.
     - As many categories as you want (including none) for images of defective samples.

   Use the `EClassificationDataset` class to compile your labeled images.
3. Choose the deep learning tool that suits your need.

   All deep learning tools are child classes of the `EDeepLearningTool` class:
   - **EasyClassify**: `EClassifier` class

- □ **EasySegment**: EUnsupervisedSegmenter class

4. Train the deep learning tool on the dataset.
5. Apply the trained tool in production.

Each tool returns a specific object.

Tools and Resources

Deep Learning Studio

Deep Learning Studio is a graphical user interface that:

- □ Creates datasets and labeling images,
- □ Configures and visualizes the data augmentation transformations,
- □ Trains a deep learning tool,
- □ Analyzes the performance of the tool,
- □ Applies the tool to new images.

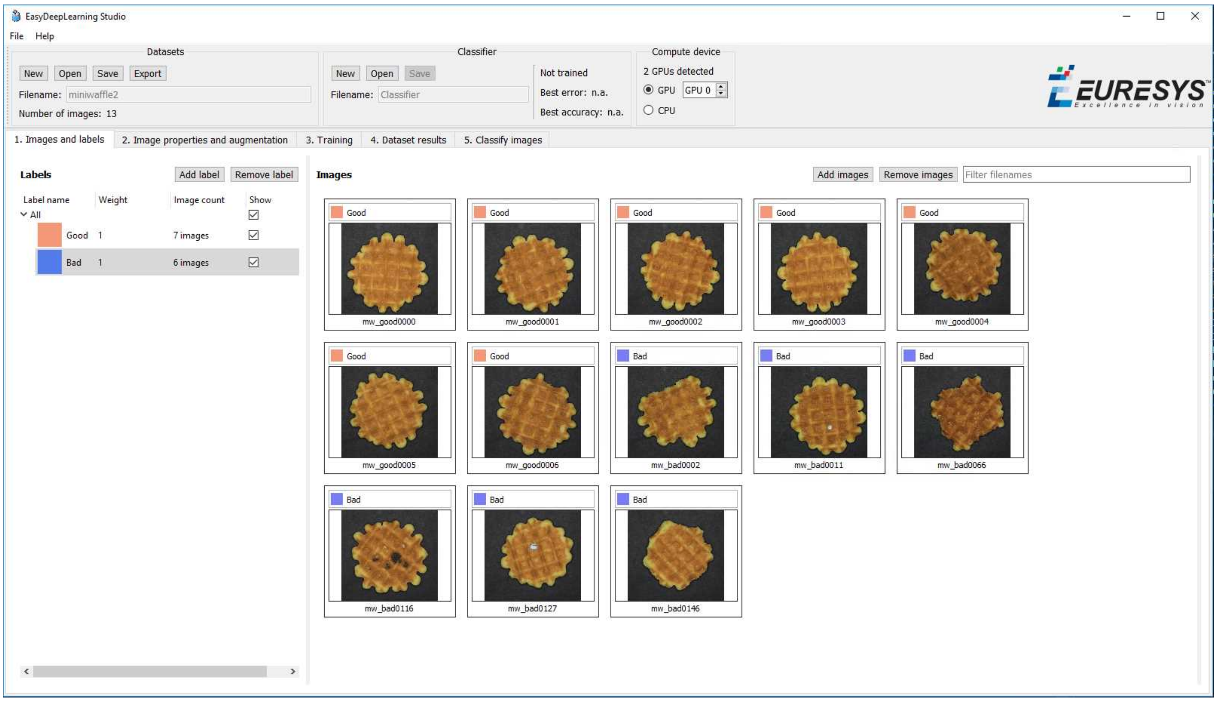
TIP
The **Deep Learning Studio** is available in the installation folder of Open eVision.

Resources and code snippets

- The sample dataset called "MiniWaffle" illustrated below is available in the folder `Sample Images/Deep Learning/EasyClassify/MiniWaffle` of the installation folder. This dataset contains images of good and bad mini waffles.
- Some sample programs in the folder `Sample Programs` show how to train and use a deep learning tool.
- Some code snippets are also provided for illustration and reference.

Workflow illustration with Deep Learning Studio

You can use **Deep Learning Studio** to perform steps 2 and 3 of the process described in section "Purpose and Workflow" on page 190.



Filename: miniwaffle2

Number of images: 13

Filename: Classifier

Not trained

Best error: n.a.

Best accuracy: n.a.

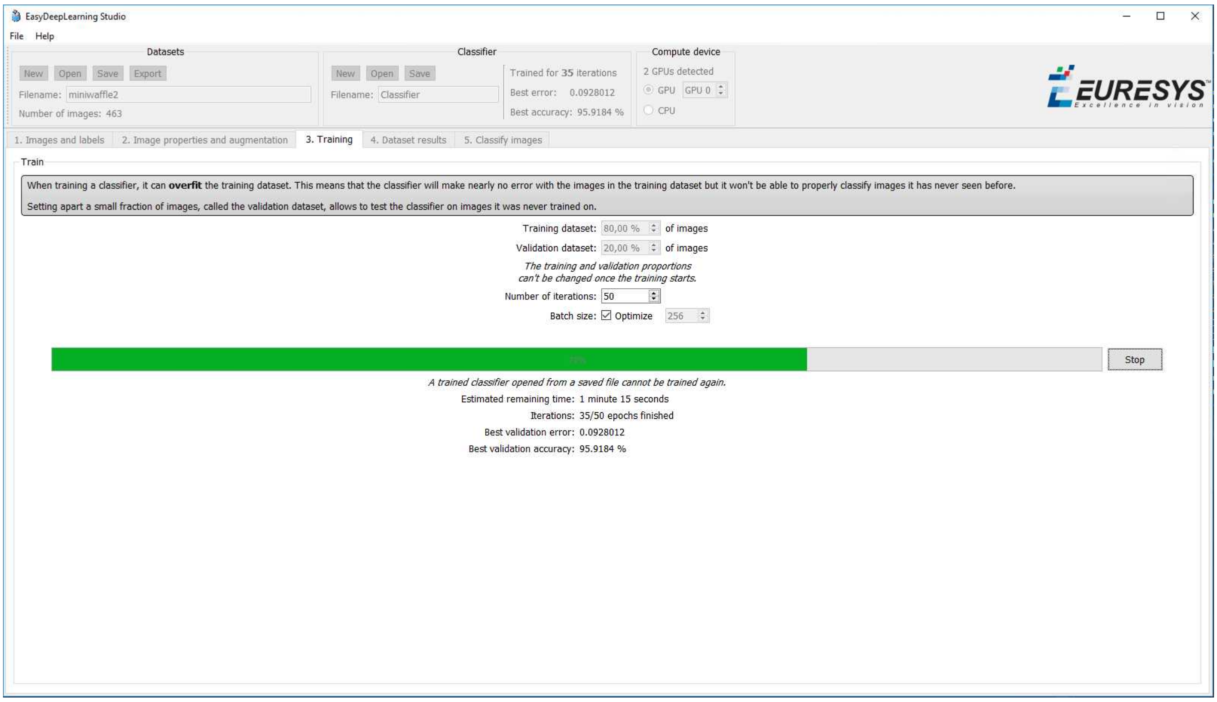
2 GPUs detected

● GPU GPU 0

○ CPU

Label name	Weight	Image count	Show
All			☑
Good	1	7 images	☑
Bad	1	6 images	☑

Manual labeling of images (step 2) and creating the dataset (steps 3a and 3b)



Filename: miniwaffle2

Number of images: 463

Filename: Classifier

Trained for **35** iterations

Best error: 0.0928012

Best accuracy: 95.9184 %

2 GPUs detected

● GPU GPU 0

○ CPU

When training a classifier, it can **overfit** the training dataset. This means that the classifier will make nearly no error with the images in the training dataset but it won't be able to properly classify images it has never seen before.

Setting apart a small fraction of images, called the validation dataset, allows to test the classifier on images it was never trained on.

Training dataset: 80,00 % of images

Validation dataset: 20,00 % of images

The training and validation proportions can't be changed once the training starts.

Number of iterations: 50

Batch size: ☑ Optimize 256

A trained classifier opened from a saved file cannot be trained again.

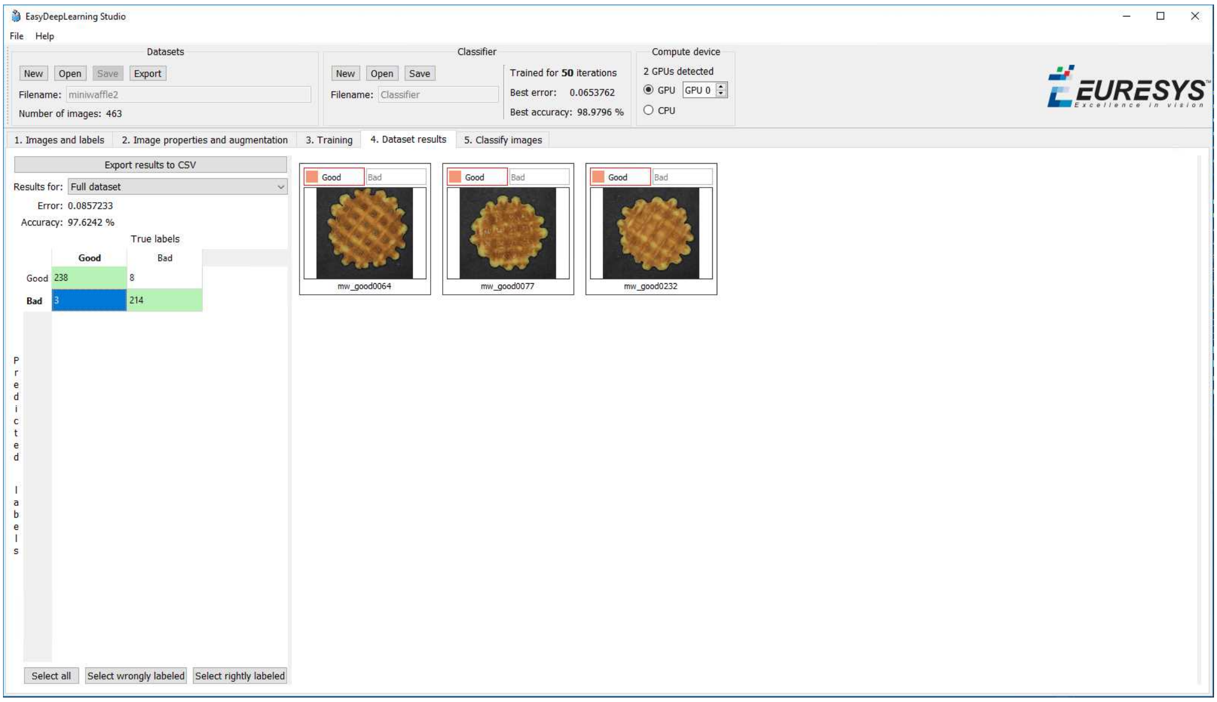
Estimated remaining time: 1 minute 15 seconds

Iterations: 35/50 epochs finished

Best validation error: 0.0928012

Best validation accuracy: 95.9184 %

Splitting the dataset (step 3c) and starting the training (steps 3d and 3e)



Filename: miniwaffle2

Number of images: 463

Filename: Classifier

Trained for **50** iterations

Best error: 0.0653762

Best accuracy: 98.9796 %

2 GPUs detected

● GPU GPU 0

○ CPU

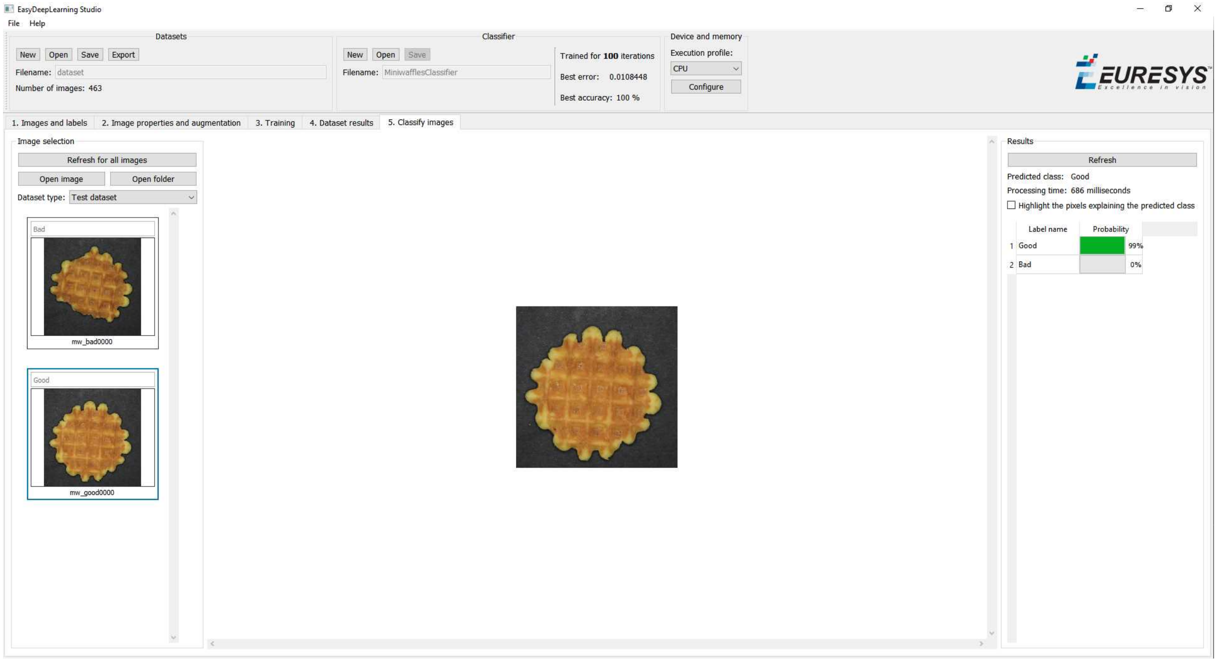
Results for: Full dataset

Error: 0.0857233

Accuracy: 97.6242 %

		True labels	
		Good	Bad
Predicted labels	Good	238	8
	Bad	3	214

Analyzing the performance (step 3f)



Filename: dataset

Number of images: 463

Filename: MiniwafflesClassifier

Trained for **100** iterations

Best error: 0.0108448

Best accuracy: 100 %

Execution profile: CPU

Dataset type: Test dataset

Predicted class: Good

Processing time: 686 milliseconds

□ Highlight the pixels explaining the predicted class

	Label name	Probability
1	Good	99%
2	Bad	0%

Classifying images (step 4)

Hardware Support (CPU/GPU)

Using a CPU

- Deep learning algorithms perform a lot of computations and can be very slow to train on a CPU.

  For example, for **EasyClassify**, on a high-end Intel Core i9-7900X CPU with a single thread, with no data augmentation:

  - □ The training can process up to 0.5 MegaPixels/second.
  - □ The validation and classification can process up to 1.5 MegaPixels/second.

- Use the `EDeepLearningTool::SetEnableGPU(false)` method to use the CPU with the deep learning tools.

✓ **TIP**
The deep learning tools support CPU processing for both 32-bit and 64-bit applications. However, the memory of a 32-bit application is limited to 2 GB and this can slow the training or the classification of large images.

Using an NVIDIA CUDA® GPU

- Using a recent NVIDIA GPU greatly accelerates the processing speeds.

  For **EasyClassify**, on a NVIDIA GeForce 1080Ti, with no data augmentation:

  - □ The training can process up to 50 MegaPixels/second.
  - □ The validation can process up to 160 MegaPixels/second.
  - □ The classification of a single image can process up to 55 MegaPixels/second (equivalent to more than 800 256 x 256 grayscale images/second).

✓ **TIP**
Please be aware that the actual speed varies with the input image format, the data augmentation, the batch size and the GPU model.

1. To use an NVIDIA GPU with the deep learning tools, install the following NVIDIA libraries on your computer:

   - □ NVIDIA CUDA® Toolkit version v10.0 (https://developer.nvidia.com/cuda-toolkit)
   - □ NVIDIA CUDA® Deep Neural Network library (cuDNN) v7 (https://developer.nvidia.com/cudnn)

2. According to the installation location:
   - □ If you install the NVIDIA CUDA® Toolkit in its default location (`C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0`), a deep learning tool automatically finds what it needs.
   - □ Otherwise, copy the DLLs `cusolver64_100.dll`, `curand64_100.dll`, `cufft64_100.dll` and `cublas64_100.dll` in the Open eVision DLL folder (its default location is `C:\Program Files (x86)\Euresys\Open eVision X.X\Bin64\`).
3. Install the NVIDIA CUDA ® Deep Neural Network library (cuDNN) that comes as a zip archive:
   a. Unzip the files.
   b. Copy the unzipped files to the NVIDIA CUDA® Toolkit installation directory as indicated in https://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html#installwindows.
   c. If the NVIDIA CUDA® Toolkit is not installed in its default location, copy the DLL file `cudnn64_7.dll` in the Open eVision DLL folder (its default location is `C:\Program Files (x86)\Euresys\Open eVision X.X\Bin64\`).
4. Use the method `EDeepLearningTool::SetEnableGPU(true)` to use the GPU with the deep learning tools.

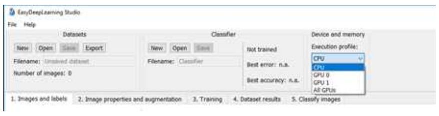
Using multiple GPUs

You can use multiple GPUs for the training and the batch classification.

- In the API, to set the list of GPUs, use the `EDeepLearningTool::SetGPUIndexes` method.

> **NOTE**
> Using multiple GPUs increases the training and batch classification speed only if these GPUs are Quadro or Tesla models with the TCC driver model (see https://docs.nvidia.com/gameworks/content/developertools/desktop/nsight/tesla_compute_cluster.htm).
> Using multiple GeForce GPUs is slower than using a single one. If there are more than one GPU installed on your computer, set the index of the GPU to use with the `EDeepLearningTool::SetGPUIndexes` method.

- In **Deep Learning Studio**, to choose the processing devices, select an execution profile.



Filename: Unsaved dataset

Number of images: 0

Filename: Classifier

Not trained

Best error: n.a.

Best accuracy: n.a.

Execution profile: CPU

- You can configure these execution profiles to match your needs.
- GPU processing is not possible with 32-bit applications.

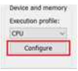
Image cache

The image cache is the part of the memory reserved for storing images during training.

- The default size is 1 GB.
- With large dataset, increasing the image cache size may improve the training speed.

To specify the cache size in bytes:

- In the API, use the `EDeepLearningTool::SetImageCacheSize` method.
- In **Deep Learning Studio**, click on the Configure button below the Execution profile control and select Image cache in the menu.



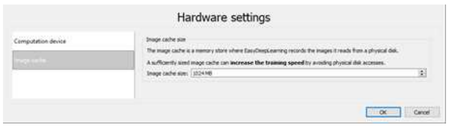


Image cache size

The image cache is a memory store where EasyDeepLearning records the images it reads from a physical disk.

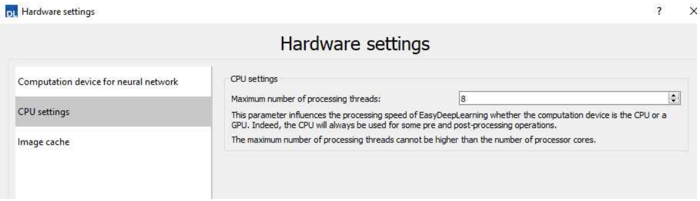
A sufficiently sized image cache can **increase the training speed** by avoiding physical disk accesses.

Image cache size: 1024 MB

Multicore processing

The deep learning tools support multicore processing (see "Multicore Processing" on page 35):

- In the API, use the multicore processing helper function from Open eVision (that is `Easy::SetMaxNumberOfProcessingThreads()` with a value greater than 1).
- In **Deep Learning Studio**, click on the Configure button below the Execution profile control and select CPU Settings in the menu.



CPU settings

Maximum number of processing threads: 8

This parameter influences the processing speed of EasyDeepLearning whether the computation device is the CPU or a GPU. Indeed, the CPU will always be used for some pre and post-processing operations.

The maximum number of processing threads cannot be higher than the number of processor cores.

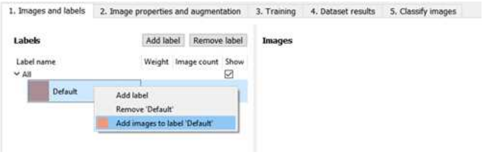
Managing the Dataset

Images and Labels

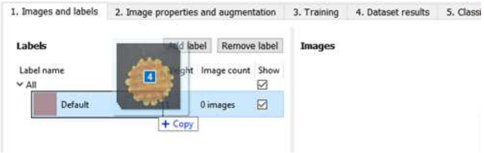
- In the API, a dataset is represented by an object of the `EClassificationDataset` type.
- The supported image file types are:
  - □ PNG
  - □ TIFF
  - □ JPEG
  - □ BMP
  - □ J2K
- The supported Open eVision image object types are:
  - □ `EImageType_BW8`
  - □ `EImageType_BW16`
  - □ `EImageType_C24`

Adding images

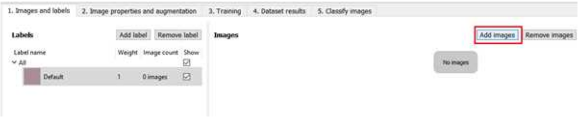
- In **Deep Learning Studio**, add image files (PNG, TIFF, JPEG, BMP and J2K types) to your datasets in one of the following ways:
  - □ Right-click on a label and click Add images to label.



  - □ Drag and drop your files directly on a label.



  - □ Select a label and click on the Add Images button.



- Add a single image and its label to a `EClassificationDataset`, in one of the following ways:
  - □ `EClassificationDataset::AddImage(path, label)` for an image file,
  - □ `EClassificationDataset::AddImage(img, label)` for an **Open eVision** image object.
- Add several image files that share the same label, with the method `EClassificationDataset::AddImages(filter, label)`.

  `filter` is a glob pattern with the wildcard characters:

  - □ `*` means "zero or more characters"
  - □ `?` means "a single character"

  For example, `EClassificationDataset::AddImages("*good*.png", "good")` adds all PNG image files that contain “`good`” in their filename.

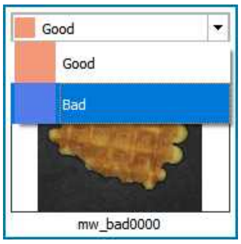
TIP
EasyClassify automatically generates the set of labels from the labels of the images that you add to the dataset.

NOTE
The labels are case sensitive.

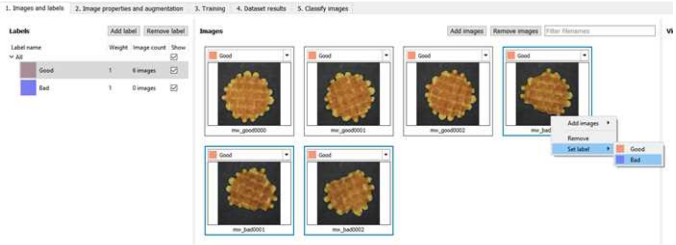
Changing the labels

In **Deep Learning Studio**:

- To change the label of a single image, select or type the label directly on the image thumbnail.



- To change the label of a group of images:
  a. Select the images (keep the CTRL key pressed and click on the images).
  b. Right click on one of the selected images.
  c. Select the new label.



Setting the label weights

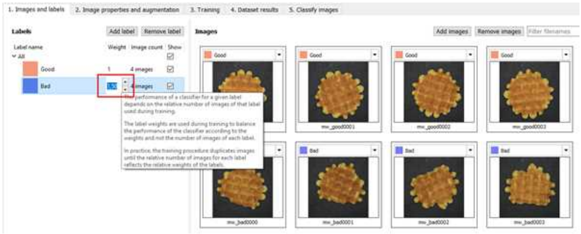
The label weights represent the relative importance that the deep learning tool gives to each class during the training.

TIP
The **EasySegment** tool does not use the label weights as it is trained on the images with one and only one specific label.

TIP
Increase the weight of a label to improve the accuracy of this label. Keep in mind that this also means a lowering of the accuracy of the other labels.

By default, all labels are given the same weight of 1.

- In **Deep Learning Studio**, set the label weights directly in the label list of the first tab:



The performance of a classifier for a given label depends on the relative number of images of that label used during training.

The label weights are used during training to balance the performance of the classifier according to the weights and not the number of images of each label.

In practice, the training procedure duplicates images until the relative number of images for each label reflects the relative weights of the labels.

- In the API, set the weight of a label with `EClassificationDataset::SetLabelWeight (labelId, weight)`.

ROI and Mask

Setting a ROI

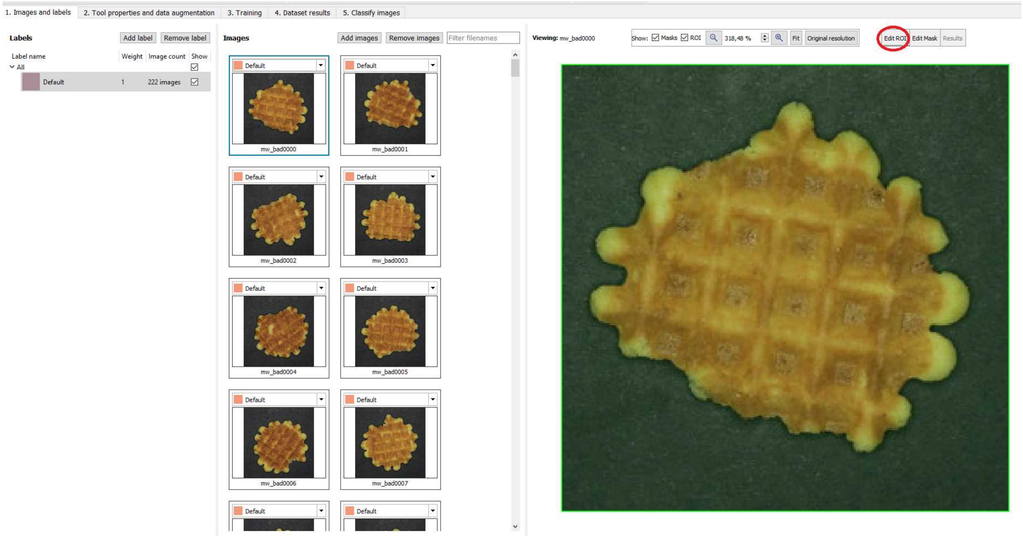
Use an ROI (region of interest) to crop an image or a whole dataset to a rectangular area aligned with the axis.

In the API:

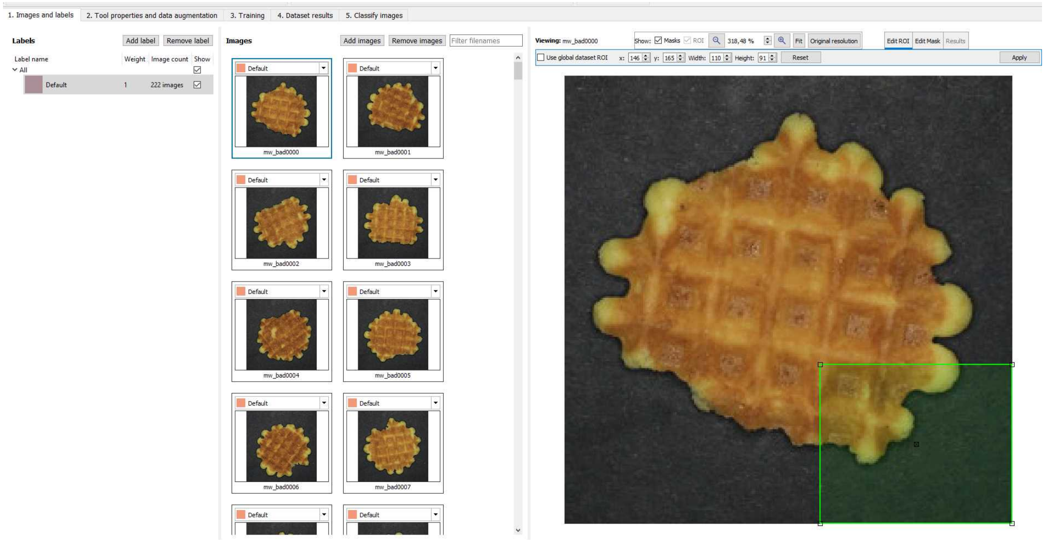
- To define a global ROI, use the `SetGlobalRegionOfInterest` method (set the ROI origin, width and height).
- To define a different ROI for each image, specify the ROI when you add an image to the dataset.

*In **Deep Learning Studio**:*

- To change the ROI:
  a. Select an image from the dataset.
  b. Click on the Edit ROI button.

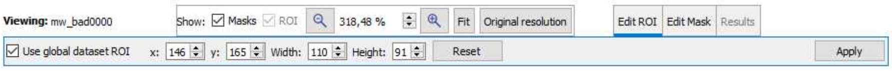


c. Drag the ROI green box, or directly set the ROI origin, width and height.



- To set the same ROI for all the images of the dataset:

a. Check the Use global dataset ROI box.



b. Click on the Apply button.

Setting a mask

Set a mask on an image in a dataset to remove the pixels in the mask area from any computation. The mask works as a “don’t care area”.

In the API:

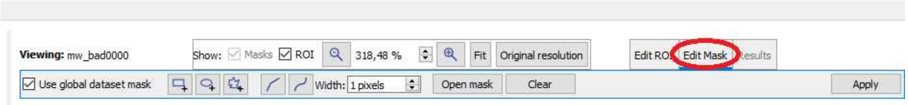
- To define a global mask, use the `SetGlobalMask` method (use a a `ERegion` to define the mask).
- To define a different mask for each image, specify the mask when you add an image to the dataset.

*In **Deep Learning Studio**:*

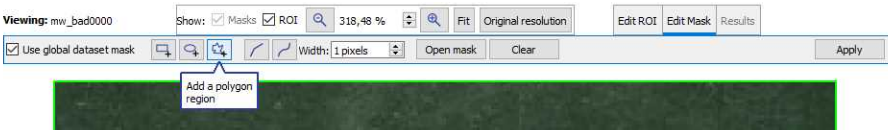
- To change the mask:

a. Select an image from the dataset.

b. Click on the Edit Mask button.



c. Select a kind of `ERegion` to draw the mask.



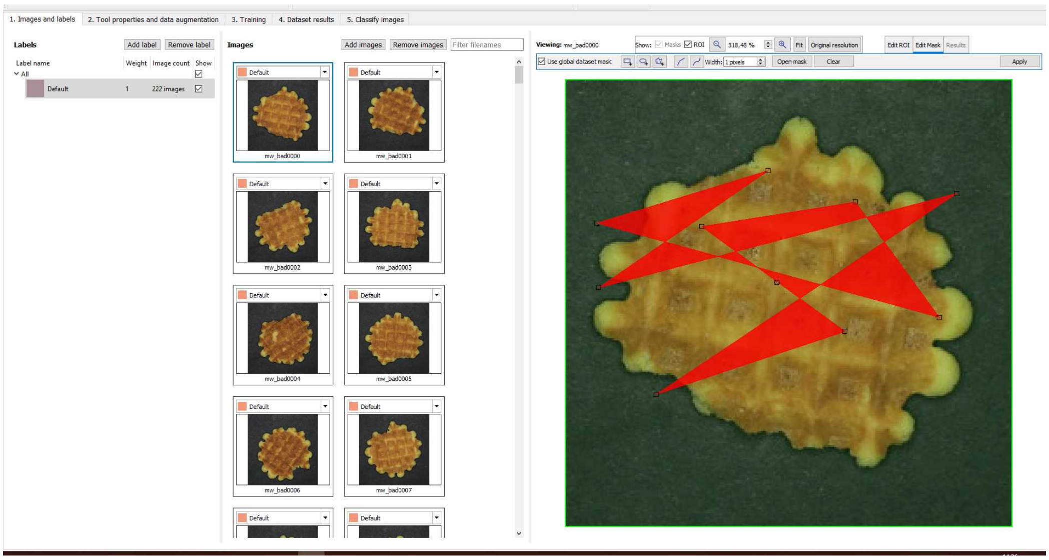
Viewing: mw_bad0000

Show: ☑ Masks ☑ ROI 318,48 %

☑ Use global dataset mask

Width: 1 pixels

d. Draw the mask.

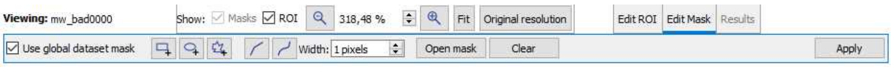


TIP

Click on the Open mask button to use an image to specify a mask. All the pixels of the image (such as an `EROIBW8`) that are over 127 are considered as part of the mask.

- To set the same mask for all the images of the dataset:

a. Check the Use global dataset mask box.



Viewing: mw_bad0000

Show: ☑ Masks ☑ ROI 318,48 %

☑ Use global dataset mask

Width: 1 pixels

b. Click on the Apply button.

Training and Validation Datasets

It is important to use at least 2 separate datasets of images:

- □ A training dataset to train the classifier.
- □ A validation dataset to automatically select the best classifier during the training.
- □ An optional test dataset to evaluate the final performance of your classifier.

> **WARNING**
> These datasets MAY NOT contain:
> - Images of the other datasets.
> - Images of an object of interest extracted from images of the other datasets.

Deep Learning Studio automatically and randomly splits the dataset into a training and a validation dataset. Add images to the test dataset in the tab Classify images.

Why is it important?

Deep learning techniques can suffer from overfitting; this means that the trained classifier is too focused on the specific images present in the training dataset and it is not able to learn a general model of your data. Such tools perform poorly in production.

The validation dataset is used during training to prevent and know when overfitting occurs. This keeps the tool in a state that gives the best performance on the validation dataset. Without the validation dataset, it is impossible to know if a tool that performs well on its training dataset can perform well in production too.

Thus, a tool that gives high performance on the training dataset but much lower performance on the validation dataset has overfitted.

To fix overfitting:

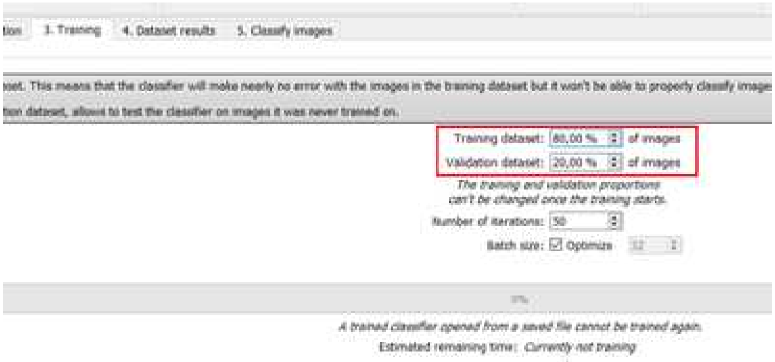
- □ You can add more images in your dataset.
- □ Or, in some cases, you can use data augmentation.

> ✓ **TIP**
> Data augmentation generates random transformations of the images in the training dataset to make the tool robust to geometric, luminosity or noise differences that are not present in the original training dataset.

Splitting the dataset

To create your training and validation datasets:

- In **Deep Learning Studio**:
  - □ Create a single dataset in the Images and labels tab.
  - □ Set the splitting percentages in the Training tab.
  - □ During the training, the dataset automatically splits into a training and a validation dataset according to this splitting percentage.



set. This means that the classifier will make nearly no error with the images in the training dataset but it won't be able to properly classify image

tion dataset, allows to test the classifier on images it was never trained on.

Training dataset: 80,00 % of images

Validation dataset: 20,00 % of images

The training and validation proportions can't be changed once the training starts.

Number of iterations: 50

Batch size: ☑ Optimize 32

0%

A trained classifier opened from a saved file cannot be trained again.

Estimated remaining time: *Currently not training*

- In the API:
  - □ Create directly 2 `EClassificationDataset` objects containing 2 different sets of images.
  - □ Or randomly split an `EClassificationDataset` dataset into 2 parts with the method `EClassificationDataset::SplitDataset(trainingDataset, validationDataset, trainingProportion)`.

Using Data Augmentation

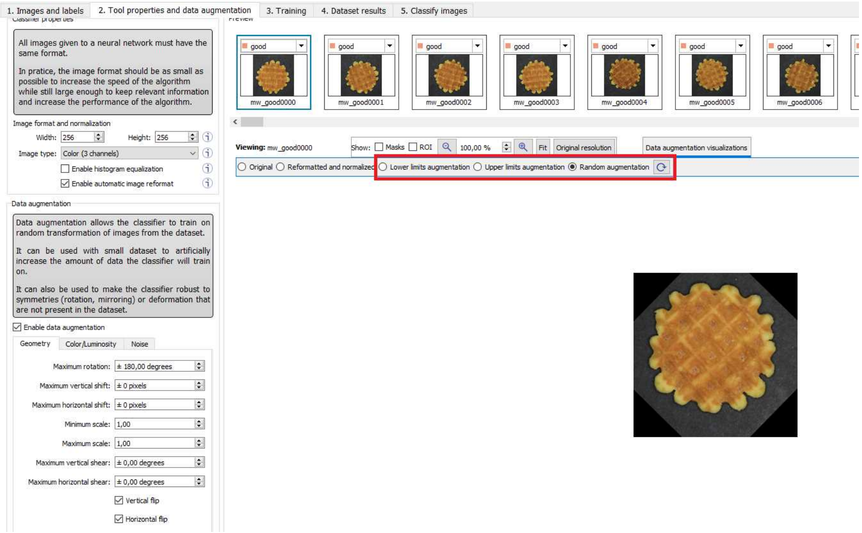
Data augmentation performs random transformations on images given to a deep learning tool (`EClassifier` or `EUnsupervisedSegmenter` object) during the training.

- Experiment different settings to choose the best parameters for your data augmentation.
- Check that the transformations do not change the label of an image (for example a defect that disappears because of a rotation or a contrast change).

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations.

In Deep Learning Studio:

- Configure the data augmentation in the second tab (Image properties and augmentation).
- Display and review the data augmented images with the minimum settings (Lower limits augmentation), the maximum settings (Upper limits augmentation) or the random settings (Random augmentation).



All images given to a neural network must have the same format.

In pratice, the image format should be as small as possible to increase the speed of the algorithm while still large enough to keep relevant information and increase the performance of the algorithm.

Image format and normalization

Width: 256 Height: 256

Image type: Color (3 channels)

□ Enable histogram equalization

☑ Enable automatic image reformat

Data augmentation

Data augmentation allows the classifier to train on random transformation of images from the dataset.

It can be used with small dataset to artificially increase the amount of data the classifier will train on.

It can also be used to make the classifier robust to symmetries (rotation, mirroring) or deformation that are not present in the dataset.

☑ Enable data augmentation

Geometry | Color/Luminosity | Noise

Maximum rotation: ± 180,00 degrees

Maximum vertical shift: ± 0 pixels

Maximum horizontal shift: ± 0 pixels

Minimum scale: 1,00

Maximum scale: 1,00

Maximum vertical shear: ± 0,00 degrees

Maximum horizontal shear: ± 0,00 degrees

☑ Vertical flip

☑ Horizontal flip

Viewing: mw_good0000

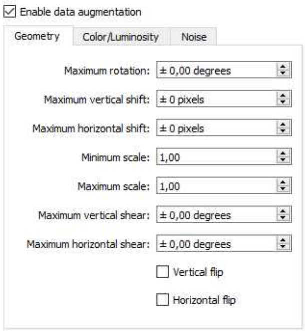
○ Original ○ Reformatted and normalized ○ Lower limits augmentation ○ Upper limits augmentation ● Random augmentation

In the API:

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations.

The possible transformations are:

Geometric transformations



☑ Enable data augmentation

Geometry | Color/Luminosity | Noise

Maximum rotation: ± 0,00 degrees

Maximum vertical shift: ± 0 pixels

Maximum horizontal shift: ± 0 pixels

Minimum scale: 1,00

Maximum scale: 1,00

Maximum vertical shear: ± 0,00 degrees

Maximum horizontal shear: ± 0,00 degrees

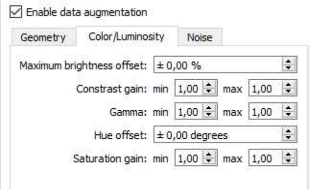
□ Vertical flip

□ Horizontal flip

- Horizontal and vertical flips (enabled with `EClassificationDataset::SetEnableHorizontalFlip` and `EClassificationDataset::SetEnableVerticalFlip`)
- Scaling (between a minimum and maximum value defined with `EClassificationDataset::SetMinScale` and `EClassificationDataset::SetMaxScale`)

- Horizontal and vertical shifts (between `-maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxHorizontalShift(maxValue)` and `EClassificationDataset::SetMaxVerticalShift(maxValue)`)
- Rotations (between 0 and a maximum value defined with `EClassificationDataset::SetMaxRotationAngle)`
- Horizontal and vertical shear (between –`maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxHorizontalShear` and `EClassificationDataset::SetMaxVerticalShear`)

Color and luminosity transformations

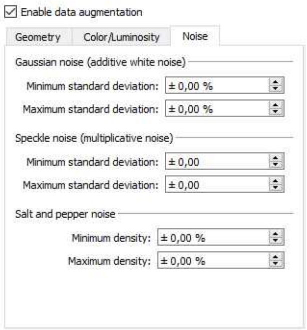


☑ Enable data augmentation

Maximum brightness offset: ± 0,00 %
Constrast gain: min 1,00 max 1,00
Gamma: min 1,00 max 1,00
Hue offset: ± 0,00 degrees
Saturation gain: min 1,00 max 1,00

- Brightness offset (between –`maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxBrightnessOffset`)
- Contrast gain (between a minimum and maximum value defined with `EClassificationDataset::SetMinContrastGain` and `EClassificationDataset::SetMaxContrastGain`)
- Gamma corrections (between a minimum and maximum value defined with `EClassificationDataset::SetMinGamma` and `EClassificationDataset::SetMaxGamma`)
- Hue offset (between –`maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxHueOffset`)
- Saturation gain (between a minimum and maximum value defined with `EClassificationDataset::SetMinSaturationGain` and `EClassificationDataset::SetMaxSaturationGain`)

Noise transformations



☑ Enable data augmentation

Gaussian noise (additive white noise)
Minimum standard deviation: ± 0,00 %
Maximum standard deviation: ± 0,00 %

Speckle noise (multiplicative noise)
Minimum standard deviation: ± 0,00
Maximum standard deviation: ± 0,00

Salt and pepper noise
Minimum density: ± 0,00 %
Maximum density: ± 0,00 %

TIP
The standard deviation is expressed as a percentage of the maximum pixel value.

- Gaussian noise, also called additive white noise, generated with a standard deviation (between a minimum and maximum value defined with `EClassificationDataset::SetGaussianNoiseMinimumStandardDeviation` and `EClassificationDataset::SetGaussianNoiseMaximumStandardDeviation`)
- Speckle noise, a multiplicative noise, generated from a Gamma distribution with a mean of 1 and a standard deviation (between a minimum and a maximum value defined with `EClassificationDataset::SetSpeckleNoiseMinimumStandardDeviation` and `EClassificationDataset::GetSpeckleNoiseMinimumStandardDeviation`).
- Salt and pepper noise generated from a pixel density (between a minimum and a maximum value defined with `EClassificationDataset::SetSaltAndPepperNoiseMinimumDensity` and `EClassificationDataset::SetSaltAndPepperNoiseMaximumDensity`).

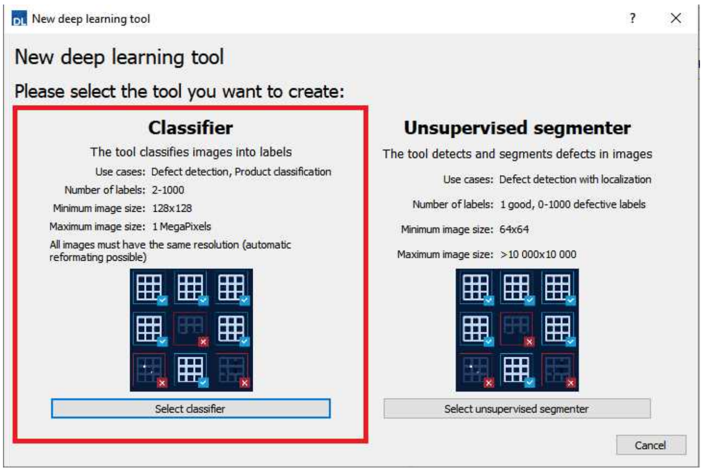
0.2. EasyClassify - Classifying Images

EasyClassify is the deep learning classification library of **Open eVision** (`EClassifier` class).

Deep Learning Studio

To create a classification tool in Deep Learning Studio:

1. Start Deep Learning Studio.
2. Select Classifier in the New deep learning tool dialog.



New deep learning tool

Please select the tool you want to create:

Classifier

The tool classifies images into labels

Use cases: Defect detection, Product classification

Number of labels: 2-1000

Minimum image size: 128x128

Maximum image size: 1 MegaPixels

All images must have the same resolution (automatic reformating possible)

Unsupervised segmenter

The tool detects and segments defects in images

Use cases: Defect detection with localization

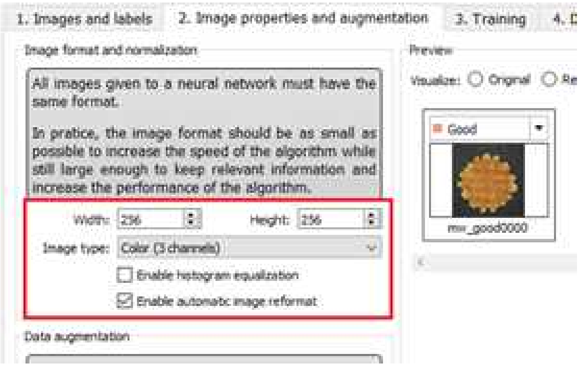
Number of labels: 1 good, 0-1000 defective labels

Minimum image size: 64x64

Maximum image size: >10 000x10 000

Input image format and normalization

- The input image format must have the width, height and number of channels corresponding to the input of the neural network.
- By default, a classifier uses the image format of the first image inserted in the training dataset:
  - □ All other images are automatically reformatted (anisotropic rescaling and conversion between color and grayscale).
  - □ If `EClassifier::SetEnableAutomaticImageReformat(false)` is called, the classifier throws an exception when attempting to train or classify an image that does not have the correct image format.
- In **Deep Learning Studio**, you can set the input image format in the Image properties and augmentation tab.



1. Images and labels
2. Image properties and augmentation
3. Training
4. D
Image format and normalization
Preview
Visualize: ○ Original ○ Re
Good
mx_good0000
Data augmentation

All images given to a neural network must have the same format.

In pratice, the image format should be as small as possible to increase the speed of the algorithm while still large enough to keep relevant information and increase the performance of the algorithm.

Width: 256

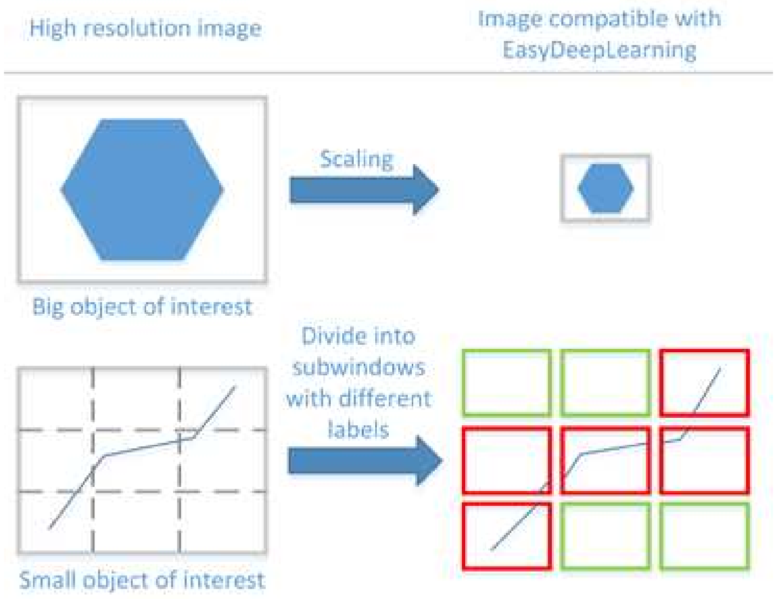
Height: 256

Image type: Color (3 channels)

□ Enable histogram equalization

☑ Enable automatic image reformat

- In the API, you can also set manually the input image format with the methods `SetWidth`, `SetHeight` and `SetChannels` (1 channel for grayscale images and 3 channels for color images).
- The input image format must have a resolution between 128 x 128 and 1024 x 1024.
  For the best processing speed, use the lowest resolution at which your "objects of interest" are still recognizable.
  - □ If your original images are smaller than the minimum resolution, upscale them to a resolution higher or equal to 128 x 128.
  - □ If your original images are larger than the maximum resolution, lower the resolution:
    - If the "objects of interest" are still recognizable, explicitly set the input image format of the classifier to this lower resolution.
    - If the "objects of interest" are not recognizable, divide your original images into sub-windows and use these sub-windows to train the classifier and make prediction. This presents the additional advantage of localizing the "object of interest" inside the original image.


High resolution image
Image compatible with EasyDeepLearning
Scaling
Big object of interest
Divide into subwindows with different labels
Small object of interest

Histogram equalization

The classifier can also apply an histogram equalization to every input image:

- □ In **Deep Learning Studio**, activate it in the image format controls in the Image properties and augmentation tab.
- □ In the API, use `EClassifier::SetEnableHistogramEqualization(true)` to activate it.

Training the classifier

In the API, to train a classifier, call the method `EClassifier::Train(trainingDataset, validationDataset, numberOfIterations)`.

- Iteration:
  - □ An iteration corresponds to going through all the images in the training dataset once.
  - □ The training process requires a large number of iterations to obtain good results.
  - □ The default number of iterations is 50.
  - □ The larger the number of iterations, the longer the training is and the better the results you obtain.

TIP
Calling the `EClassifier::Train()` method several times with the same training and validation dataset is equivalent to calling `EClassifier::Train()` once but with a larger number of iterations. The total number of iterations used to train the classifier is accessible through `EClassifier::GetNumTrainedIterations()`.

- The training process is asynchronous:
  - □ A call to `EDeepLearningTool::Train` launches a new thread that does the training in background.
  - □ The method `EDeepLearningTool::WaitForTrainingCompletion` suspends the program until the whole training is completed.
  - □ The method `EDeepLearningTool::WaitForIterationCompletion` suspends the program until the current iteration is completed.
  - □ During the training, use `EDeepLearningTool::GetCurrentTrainingProgression()` to follow the progression of the training.
- Batch size:

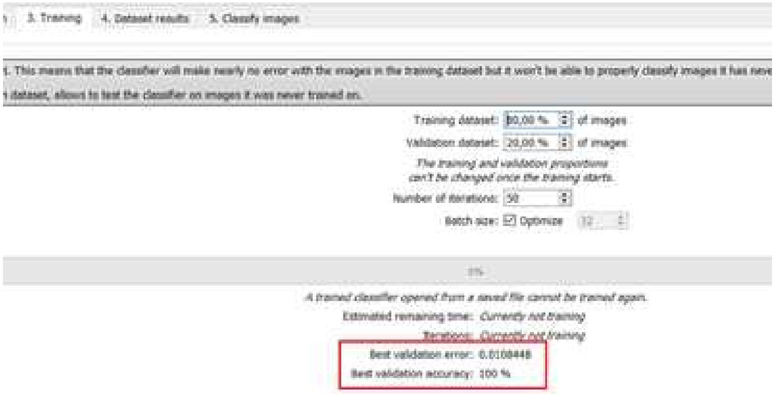
  The batch size corresponds to the number of images that are processed together.

  - □ The training is influenced by the batch size.
  - □ A large batch size increases the processing speed of a single iteration on a GPU but requires more memory.
  - □ The training process is not able to learn a good model with too small batch sizes.
  - □ By default, the batch size is determined automatically during training to optimize the training speed with respect to the available memory.
    - Use `EDeepLearningTool::SetOptimizeBatchSize(false)` to disable this behavior.
    - Use `EDeepLearningTool::SetBatchSize` to change the size of your batch.
  - □ Use `EDeepLearningTool::GetBatchSizeForMaximumInferenceSpeed()` to get the batch size that maximizes the (batch) classification speed on a GPU with respect to the available memory.
  - □ It is common to choose powers of 2 as batch size for performance reasons.

Validating the results

*In **Deep Learning Studio**:*

- The metrics are always computed without applying data augmentation on the images.
- In the Training tab, the metrics Best validation error and Best validation accuracy are computed during the training using the label weights.



. This means that the classifier will make nearly no error with the images in the training dataset but it won't be able to properly classify images it has neve

dataset, allows to test the classifier on images it was never trained on.

Training dataset: 80,00 % of images

Validation dataset: 20,00 % of images

The training and validation proportions can't be changed once the training starts.

Number of iterations: 50

Batch size: ☑ Optimize 32

0%

A trained classifier opened from a saved file cannot be trained again.

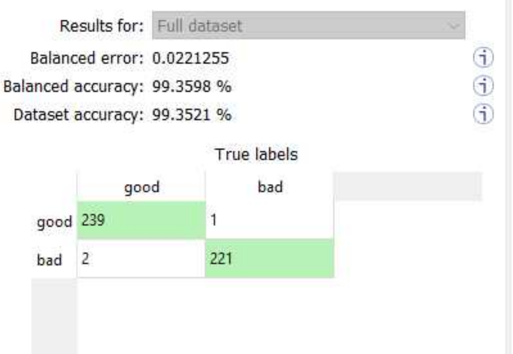
Estimated remaining time: *Currently not training*

Iterations: *Currently not training*

Best validation error: 0.0108448

Best validation accuracy: 100 %

- In the Dataset results tab, there are 3 metrics displayed:
    - □ The weighted error and the weighted accuracy (normalized with respect to the label weights instead of being dependent of the number of images for each label).
    - □ The dataset accuracy (it does not use the label weights).



Results for: Full dataset

Balanced error: 0.0221255

Balanced accuracy: 99.3598 %

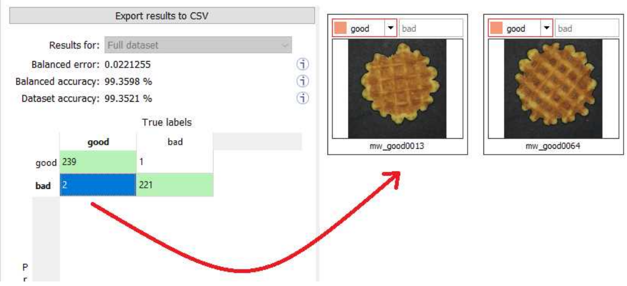
Dataset accuracy: 99.3521 %

	True labels	
	good	bad
good	239	1
bad	2	221

TIP

If your dataset has a very different number of images for each of the labels, it is called *unbalanced*. In this case, the dataset accuracy is biased towards the labels containing the most images (the dataset accuracy mainly reflects the accuracy of these labels).

- In the Dataset results tab, the confusion matrix shows the number of images according to their true labels and their label predicted by the classifier.
    - □ The diagonal elements of the matrix shown in green are the correctly classified images.
    - □ All the other elements of the matrix are badly classified images.
    - □ Select one or more elements of the matrix to show the corresponding images.



Results for: Full dataset

Balanced error: 0.0221255

Balanced accuracy: 99.3598 %

Dataset accuracy: 99.3521 %

	True labels	
	good	bad
good	239	1
bad	2	221

In the API:

- After the completion of each iteration, **EasyClassify** automatically computes several performance metrics about the training and validation dataset:
  - □ Call the methods `EClassifier::GetTrainingMetrics(iteration)` and `EClassifier::GetValidationMetrics(iteration)` to read these metrics.
  - □ The iterations are indexed between 0 and `EClassifier::GetNumTrainedIterations()`-1.
  - □ Call `EClassifier::GetBestIteration()` to retrieve the iteration that produced the best performance.
  - □ After the training, the classifier is back in the state corresponding to this best iteration.
- The metrics are represented by an `EClassificationMetrics` object that contains the following performance metrics:
  - □ The classification error (`EClassificationMetrics::GetError()`), also called the cross-entropy loss: the quantity that is minimized during the training. It is computed from the probabilities computed by the classifier.
    - The error for a single image is the negative of the logarithm of the probability corresponding to the true label of the image. So, if this probability is low, the error for the image is high.
    - The error of the dataset is the average of the errors of each image in the dataset.
  - □ The classification accuracy (`EClassificationMetrics::GetAccuracy()`): the number of images correctly classified divided by the total number of images in the dataset.
  - □ The confusion matrix (`EClassificationMetrics::GetConfusion(groundtruthLabel, predictedLabel)`): the number of images labeled as `groundtruthLabel` that are classified as `predictedLabel`.

> **TIP**
> Call `EClassifier::Evaluate` to evaluate a dataset independently of the training.

Classifying new images

- In **Deep Learning Studio**, open Classify images tab to:
  - □ Classify new images.
  - □ Display detailed results for each image of the main dataset.

- Once the classifier is trained, call `EClassifier::Classify` to classify an Open eVision image.

  This method returns a `EClassificationResult` object:

  - □ `EClassificationResult::GetBestLabel()` returns the most probable label for the image.
  - □ `EClassificationResult::GetBestProbability()` returns the probability associated with the most probable label.
  - □ `EClassificationResult::GetProbability(label)` returns the probability associated with the given label.
  - □ `EClassificationResult::GetRanking(label)` returns the ranking of the given label. The ranking goes from 1 (most probable) to `EClassifier::GetNumLabels()` (least probable).
- You can also do batch classification or directly classify a vector of Open eVision images:
  - □ Images are processed together in groups determined by the batch size.
  - □ On a GPU, it is usually much faster to classify a group of images than a single image.
  - □ On a CPU, implement a multithread approach to accelerate the classification. In that case, each thread must have its own instance of `EClassifier` (see code snippets).

✓ **TIP**

The batch classification has a tradeoff between the throughput (the number of images classified per second) and the latency (the time needed to obtain the result of an image): on a GPU, the higher the batch size, the higher the throughput and the latency. So, use batch classification to improve the classification speed at the cost of a longer time before obtaining the classification result of an image.

- Use `EClassifier::GetHeatmap(img, label)` to obtain an heat map highlighting the pixels that contribute the most to a label. In some cases, this heat map can provide a rough localization of the object corresponding to the label.

Memory requirements

- In addition to the properties of the classifier object and the weights of the neural network, an `EClassifier` object dynamically allocates memory for intermediate results during the training and the classification of new images.
- The size of the intermediate results depends on the width (W), height (H), batch size (B), and whether the operations are performed on a GPU or a CPU.
- For training, these intermediate results need about the following amount of memory:

$$\text{TrainingMemoryCPU} = 0.000453 \times W \times H \times B - 292 \text{ (MB)}$$
$$\text{TrainingMemoryGPU} = 0.000440 \times W \times H \times B + 25 \text{ (MB)}$$

- For classification, these intermediate results need up to the following memory:

$$\text{ClassificationMemoryCPU} = 0.000232 \times W \times H \times B - 97\ (MB)$$
$$\text{ClassificationMemoryGPU} = 0.000226 \times W \times H \times B + 13\ (MB)$$

- For example, training a classifier or making classifications with 256 x 256 images and a batch size of 32 on a GPU will take around respectively 950 MB or 500 MB.

> **TIP**
> Since large memory allocations take a lot of time, a classification does not released this memory and the next classifications can reuse it as long as the width, height, batch size and computation device remain the same. As such, the first classification is always be slower due to the memory allocations.

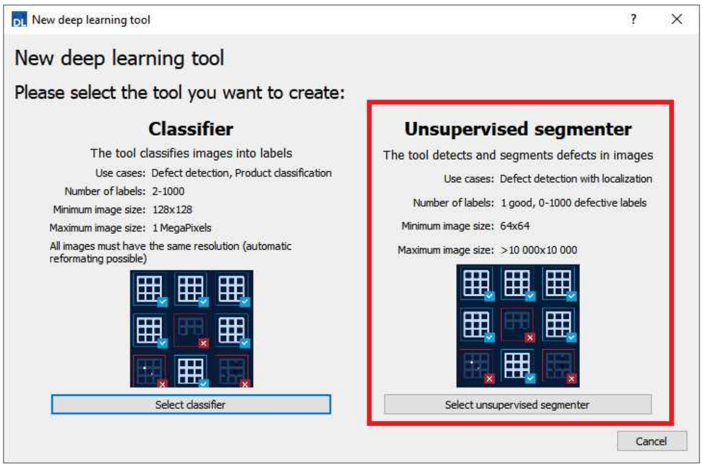
0.3. EasySegment - Detecting and Segmenting Defects

EasySegment is the deep learning segmentation library of Open eVision. It contains the unsupervised segmentation tool (EUnsupervisedSegmenter class).

Deep Learning Studio

To create create an unsupervised segmentation tool in Deep Learning Studio:

1. Start Deep Learning Studio.
2. Select Unsupervised segmenter in the New deep learning tool dialog.



New deep learning tool

Please select the tool you want to create:

Classifier

The tool classifies images into labels

Use cases: Defect detection, Product classification

Number of labels: 2-1000

Minimum image size: 128x128

Maximum image size: 1 MegaPixels

All images must have the same resolution (automatic reformating possible)

Unsupervised segmenter

The tool detects and segments defects in images

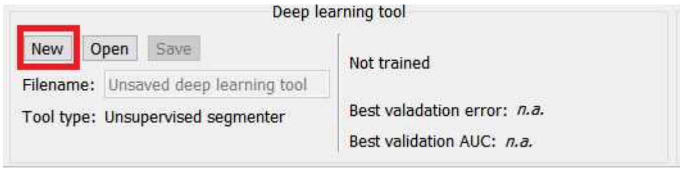
Use cases: Defect detection with localization

Number of labels: 1 good, 0-1000 defective labels

Minimum image size: 64x64

Maximum image size: >10 000x10 000

The following dialog is displayed at the start of Deep Learning Studio or when you create a new deep learning tool from the toolbar.



Filename: Unsaved deep learning tool

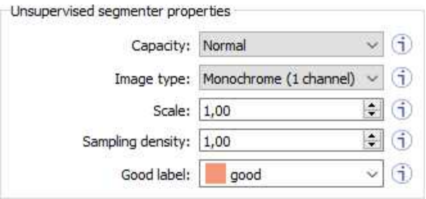
Tool type: Unsupervised segmenter

Not trained

Best valadation error: *n.a.*

Best validation AUC: *n.a.*

Configuration


Unsupervised segmenter properties

Capacity: Normal

Image type: Monochrome (1 channel)

Scale: 1,00

Sampling density: 1,00

Good label: 🟧 good

The unsupervised segmenter tool has 5 parameters:

1. The Capacity of the neural network (default: Normal) represents the quantity of information it is capable of learning. A larger capacity makes the tool slower.

   In the API:

   - The capacity is represented by the enumerate type `EUnsupervisedSegmenterCapacity`.
   - `EUnsupevisedSegmenter::Capacity` sets the capacity of the tool.

2. The Image type (default: Monochrome (1 channel)):

   In the API:

   - To use monochrome (grayscale, 1 channel) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `true`.
   - To use color (3 channels) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `false`.

3. Use the Scale (`EUnsupervisedSegmenter::Scale`) to automatically resize your images to a lower resolution and accelerate the processing.

4. The Sampling density (`EUnsupervisedSegmenter::SamplingDensity`) is the parameter of the sliding window algorithm used to process whole images using patches of size (`EUnsupervisedSegmenter::PatchSize`).

   - It indicates how much overlap there is between the image patches (100-100/`SamplingDensity` %).
   - In practice, the stride between 2 consecutive patches is `PatchSize`/`SampleDensity` pixels.

5. The Good label is the name of the class that contains the good images.

Training the tool

In the API, to train an unsupervised segmenter, call the `EDeepLearningTool::Train (trainingDataset, validationDataset, numberOfIterations)` method.

- An *iteration* corresponds to training on 10 000 image patches and computing the results for each training and validation image.
  - □ The training process requires a large number of iterations to obtain good results.
  - □ The default number of iterations is 50.
  - □ The larger the number of iterations, the longer the training is and the better the results you may obtain.

TIP
Calling the `EDeepLearningTool::Train` method several times with the same training and validation dataset is equivalent to calling it once but with a larger number of iterations.
Call `EDeepLearningTool::GetNumTrainedIterations` to get the total number of iterations used to train the classifier.

- The training process is *asynchronous*:
  - □ `EDeepLearningTool::Train` launches a new thread that does the training in background.
  - □ `EDeepLearningTool::WaitForTrainingCompletion` suspends the program until the whole training is completed.
  - □ `EDeepLearningTool::WaitForIterationCompletion` suspends the program until the current iteration is completed.
  - □ During the training, `EDeepLearningTool::GetCurrentTrainingProgression` shows the progression of the training.
- The *batch size* corresponds to the number of image patches that are processed together.
  - □ The training is influenced by the batch size.
  - □ A large batch size increases the processing speed of a single iteration on a GPU but requires more memory.
  - □ The training process is not able to learn a good model with too small batch sizes.
  - □ By default, the batch size is determined automatically during the training to optimize the training speed with respect to the available memory.
    - Use `EDeepLearningTool::SetOptimizeBatchSize(false)` to disable this behavior.
    - Use `EDeepLearningTool::SetBatchSize` to change the size of your batch.
  - □ `EDeepLearningTool::GetBatchSizeForMaximumInferenceSpeed` gets the batch size that maximizes the batch classification speed on a GPU according to the available memory.
  - □ It is common to choose powers of 2 as the batch size for performance reasons.

Validating the results

There are 2 types of metric for the unsupervised segmentation tool:

- □ *Unsupervised* metric only uses the results of the tool on good images. There is only one unsupervised metric: the error.
- □ *Supervised* metrics requires both good and defective images. The supervised metrics are the AUC (Area Under ROC Curve), the ROC curve, the accuracy, the good detection rate (also called the true negative rate), the defect detection rate (also called the true positive rate).

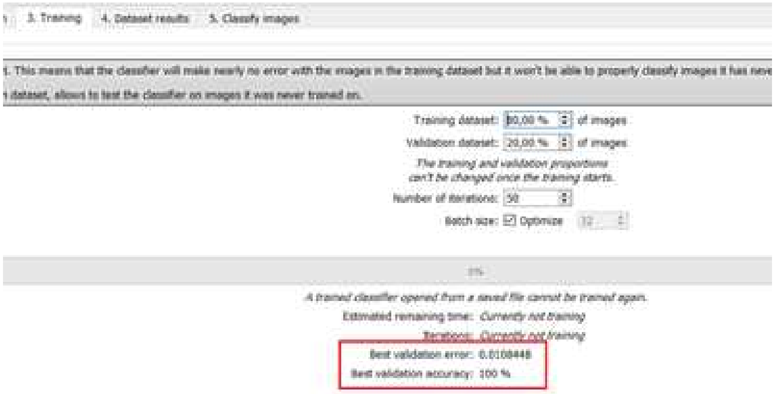
The unsupervised segmentation tool computes a score for each image (see `EUnsupervisedSegmenterResult::ClassificationScore`). The label of a result is obtained by thresholding this score with the segmenter classification threshold (`EUnsupervisedSegmenter::ClassificationThreshold`). So, the supervised metrics also depends on the value of this classification threshold.

The ROC curve (Receiver Operating Characteristic) is the plot of the defect detection rate (the true positve rate) against the rate of good images classified as defective (also called the false positive rate). It is obtained by varying the classification threshold. The ROC curve shows the possible tradeoffs between the good detection rate and the defect detection rate.

The area under the ROC curve (AUC) is independent of the chosen classification threshold and represents the overall performance of the tool. Its value is between 0 (bad performance) and 1 (perfect performance).

In ***Deep Learning Studio****:*

- In the Training tab, the metrics Best validation error and Best validation AUC are computed during the training on the validation dataset without using data augmentation. The validation error, the training error and the validation AUC are plotted for each iteration.



. This means that the classifier will make nearly no error with the images in the training dataset but it won't be able to properly classify images it has neve

dataset, allows to test the classifier on images it was never trained on.

Training dataset: 80,00 % of images

Validation dataset: 20,00 % of images

The training and validation proportions can't be changed once the training starts.

Number of iterations: 50

Batch size: ☑ Optimize 32

0%

A trained classifier opened from a saved file cannot be trained again.

Estimated remaining time: *Currently not training*

Iterations: *Currently not training*

Best validation error: 0.0108448

Best validation accuracy: 100 %

- In the Dataset results tab, various metrics, the confusion matrix, a cumulative score histogram, and the ROC curve are displayed. You can also change the classification threshold directly in this tab.
  - □ The cumulative score histogram shows the cumulative proportion of good (in green) and defective (in red) images with respect to the scores of the image.
  - □ You can change the classification threshold in 3 ways : direct input, dragging the threshold line in the score histogram and selecting a point on the ROC curve.

In the API:

- The metrics are represented by an `EUnsupervisedSegmenterMetrics` object that contains the following performance metrics:
  - □ The error on good image (`EUnsupervisedSegmenterMetrics::GetError`)
  - □ The confusion matrix (`EUnsupervisedSegmenterMetrics::GetConfusion`)
  - □ If the results for bad images are included in the metrics, `EUnsupervisedSegmenterMetrics::IsTotallyUnsupervised` is `false` and the following metrics are also be accessible:
    - The accuracy (`EUnsupervisedSegmenterMetrics::GetAccuracy`)
    - The Area under ROC curve (`EUnsupervisedSegmenterMetrics::GetAreaUnderROCCurve`)
    - The ROC point corresponding to the classification threshold (`EUnsupervisedSegmenterMetrics::GetROCPoint`)

Applying the tool to new images

*In **Deep Learning Studio**:*

- Open the Classify images tab to:
  - □ Apply the segmenter to new images.
  - □ Display detailed results for each image of the main dataset.
- Once the unsupervised segmenter is trained, call `EUnsupervisedSegmenter::Apply` to detect and segment defects in an **Open eVision** image.

  This method returns a `EUnsupervisedSegmenterResult` object:

  - □ `EUnsupervisedSegmenterResult::IsGood` and `EUnsupervisedSegmenterResult::IsDefective` returns whether the tool has decided that the image is good or defective according to the `EUnsupervisedSegmenterResult::ClassificationScore` and the `EUnsupervisedSegmenter::ClassificationThreshold`.

- `EUnsupervisedSegmenterResult::GetSegmentationMap` returns an `EImageBW8` image where all pixels with a value different than 0 are *defective* pixels.
  The value of a defective pixel is proportional to the importance of the defect at that position.
- `EUnsupervisedSegmenterResult::GetRegion` returns an `ERegion` object corresponding to the segmented region of the image (all the pixels of `EUnsupervisedSegmenterResult::GetSegmentationMap` that have a value strictly higher than 0).

PART VII

3D PROCESSING TOOLS

0.1. Easy3D - Using 3D Toolset

Basic Concepts

Easy3D

Easy3D is a set of tools for solving computer vision problem using 3D acquisition and processing. Easy3D supports laser line triangulation for fast and precise acquisition of depth maps.

> **TIP**
> Depth maps are gray scale images where each pixel represents a displacement in the third dimension. Because of the acquisition procedure, they are usually not dimensionally correct. So, while Open eVision 2D image operators are compatible with depth maps, you should not use them for processes requiring precise measurements.

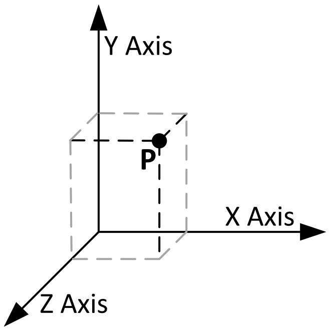
Easy3D provides a calibration tool to generate corrected, metric point clouds and meshes from depth maps. Most 3D operators work on point clouds or meshes. The included export functions to the standard PCD file format allows integration with other 3D tools.

Easy3D also allows the computation of ZMaps. A ZMap is the projection of a point cloud on a given reference plane. Like depth maps, ZMaps are gray scale images, but are also dimensionally correct. As such, they can be used with all Open eVision 2D functions.

All the Easy3D tools are placed in the Easy3D namespace.

3D representation

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



Open eVision provides different containers to store 3D objects :

- Depth maps
- Point clouds
- Meshes
- ZMaps

Depth map

A depth map is a way to represent a 3D object using a 2D grayscale image where each pixel (u, v) in the image contains a third coordinate as its gray value.

The grayscale values of a depth map do not necessarily represent a Z metric coordinate. In the context of a laser triangulation setup, these values represent the displacement of the laser line profile, which is not the physical height of the 3D surface.

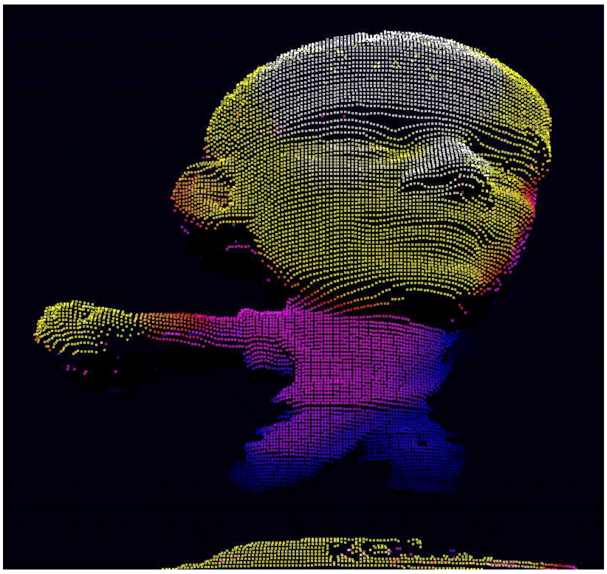
A depth map contains a gray scale image coded on 8, 16 or 32 bits per pixel.

- One specific gray value, called the undefined value, is reserved for the representation of invalid pixels.
- By default, this value is 0 for integer depth map types (`EDepthMap8` and `EDepthMap16`).
- By default, this value is the lowest float value (-3.402823 e+38) for the 32 bits floating point depth map types (`EDepthMap32f`).

The calibration process aims to convert the depth map representation to real, metric 3D representations such as point clouds or meshes.

Point cloud

A point cloud is a set of 3D points (x, y and z coordinates) representing the scanned object in the world metric space.



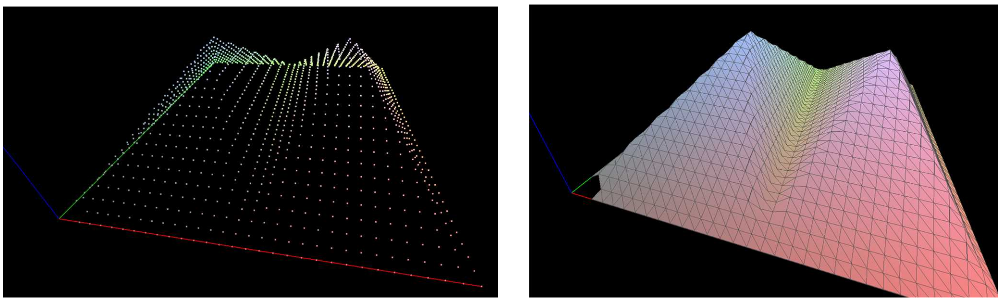
In addition to the calibration process included in Easy3D, point clouds can be produced using various 3D acquisition techniques, like stereo reconstruction or time of flight cameras.

Mesh

A Mesh is a geometric representation of a 3D surface, a set of connected 3D points.

In an `EMesh` object, 3 points are connected to define a triangle.

TIP
This kind of 3D representation is also called a "triangle mesh".



A point cloud and the corresponding mesh (displayed with Open eVision `E3DViewer`)

An `EMesh` object contains a point cloud and the indexes of the vertices of all mesh triangles.

`EMesh` uses a metric space representation that can be generated from a depth map and that can be used to produce a ZMap.

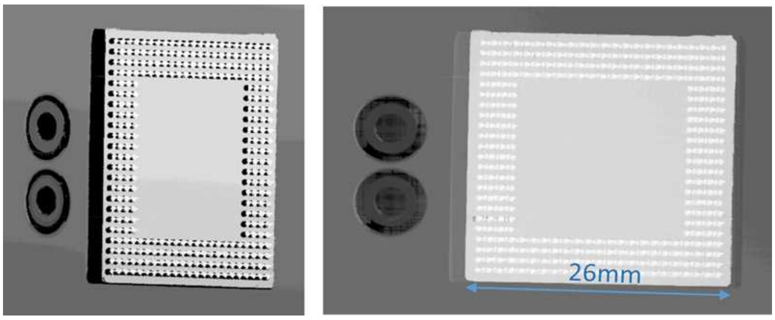
ZMap

ZMaps are another representation for 3D data.

- They are grayscale images like depth maps but represent metric and corrected 3D points.
- They are convenient representations for measurement and matching.

- □ They are compatible with most of the 2D processing functions.

ZMaps are generated by the projection of a point cloud or a mesh onto an arbitrary 3D plane.



A depth map and the corresponding ZMap

A ZMap contains an image in which each pixel value represents a positive distance from the reference plane.

TIP
Use the method `AsEImage()` to obtain a reference to the contained image.

A ZMap also contains the following information:

- □ The transformation from the World coordinates to the ZMap coordinates.
- □ The size of a pixel, called the "resolution".

TIP
Like in a depth map, a specific pixel value is reserved to represent undefined pixels. To get this pixel value, use the method `GetUndefinedValue()`.

Static Methods

EFilters class

The `EFilters` class contains static methods used to apply filters to ZMaps or depth maps.

RemoveNoise

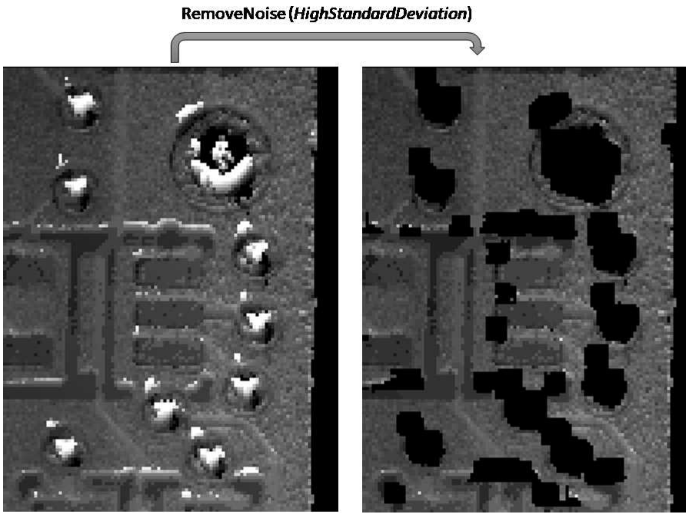
The `RemoveNoise()` method removes outliers from a depth map or a ZMap.

- It takes a depth map or a ZMap as input and generates a depth map or a ZMap respectively.

  The undefined points are not taken into account.
- It is based on a square moving kernel. The size of the kernel is (2 x `halfKernelSize` + 1) where `halfKernelSize` is a parameter of the method.

- The threshold parameter is scaled with regard to the Z resolution of the filtered depth map or ZMap.
- There are 3 variations of this filter, depending on the method parameter:
  - □ ENoiseRemovalMethod_AbsoluteDifferenceFromMean removes a point when it deviates from the average in the neighborhood, including itself. The threshold is an absolute difference.
  - □ ENoiseRemovalMethod_RelativeDifferenceFromMean removes a point when it deviates from the average in the neighborhood, including itself. The threshold is a multiple of the standard deviation.
  - □ ENoiseRemovalMethod_HighStandardDeviation removes a point when the standard deviation in the neighborhood, including itself, is higher than a defined threshold.

Example: Removing points showing a high standard deviation



The code below removes pixels with a standard deviation higher than a defined threshold.

```
// Load the ZMap data
EZMap16 zmap;
zmap.Load(...);
// Compute the filtered ZMap. The new ZMap is called filteredZmap
// The size of the kernel is 7x7, the threshold is 30.0
EZMap16 filteredZmap;
filteredZmap.SetSize(zmap);
EFilters::RemoveNoise(zmap, filteredZmap, ENoiseRemovalMethod_HighStandardDeviation, 3, 30.0, 0.0);
```

EStatistics class

The EStatistics class contains static methods used to compute statistics on ZMaps or depth maps.

ComputeAverageMap

The ComputeAverageMap() method computes the local average map.

- You can use this method as a low-pass filter.
- Undefined points are not taken into account.

- This method is based on a square moving kernel. The size of the kernel is (2 x `halfKernelSize` + 1) where `halfKernelSize` is a parameter of the method.

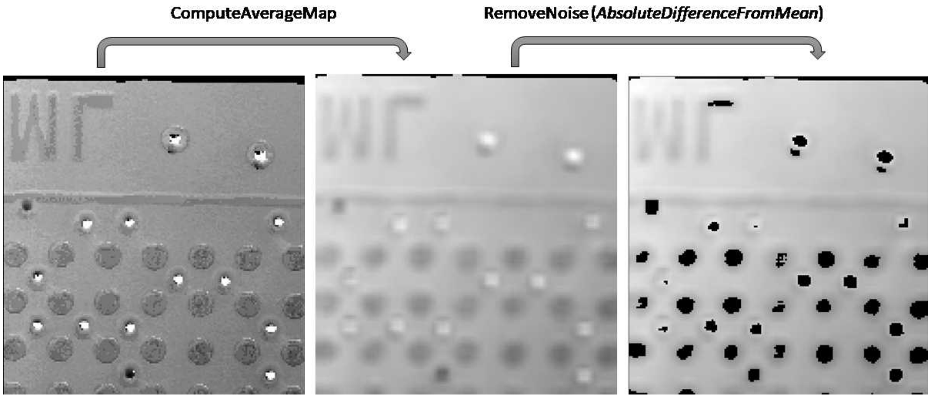
ComputeStandardDeviationMap

The `ComputeStandardDeviationMap()` method computes a map of the local standard deviation.

- You can use this method to determine visually the threshold value to use with the `RemoveNoise()` method when using the `ENoiseRemovalMethod_HighStandardDeviation` setting.

NOTE
Be aware, however, that in the generated map, a pixel with the value 0 can either be undefined or have a standard deviation equal to zero.

Example: Using a low pass filter on a ZMap, then removing points showing a deviation larger than a defined threshold



The code below first applies an low pass filter, then removes from the result the pixels showing a deviation from the neighborhood larger than the defined threshold.

```
// Load the ZMap data
EZMap16 zmap;
zmap.Load(...);
// Compute the filtered ZMap. The new ZMap is called averagedZMap
// The size of the kernel is 7x7, the threshold is 30.0
EZMap16 averagedZMap;
averagedZMap.SetSize(zmap);
EStatistics::ComputeAverageMap(zmap, averagedZMap, 3, 0.2);
// Compute the filtered ZMap. From averagedZMap, compute filteredZMap
// The size of the kernel is 31x31, the threshold is 20.0
EZMap16 filteredZMap;
filteredZMap.SetSize(zmap);
EFilters::RemoveNoise(averagedZMap, filteredZMap, ENoiseRemovalMethod_AbsoluteDifferenceFromMean,
15, 20.0, 0.2);
```

ComputePixelStatistics

The `ComputePixelStatistics()` method returns basic statistical information about pixel values:

- □ Minimum
- □ Maximum
- □ Average
- □ Standard deviation
- □ Number of valid (not undefined) pixels).

Use an `ERegion` object to specify the region of the ZMap or depth map used to compute the statistics.

ComputeStatistics

The `ComputeStatistics()` method returns the same information as the `ComputePixelStatistics()` method, but scaled with respect of the Z resolution.

Use an `ERegion` object to specify the region of the ZMap or depth map used to compute the statistics.

Point Cloud

Coordinates Transformations

Affine Transforms

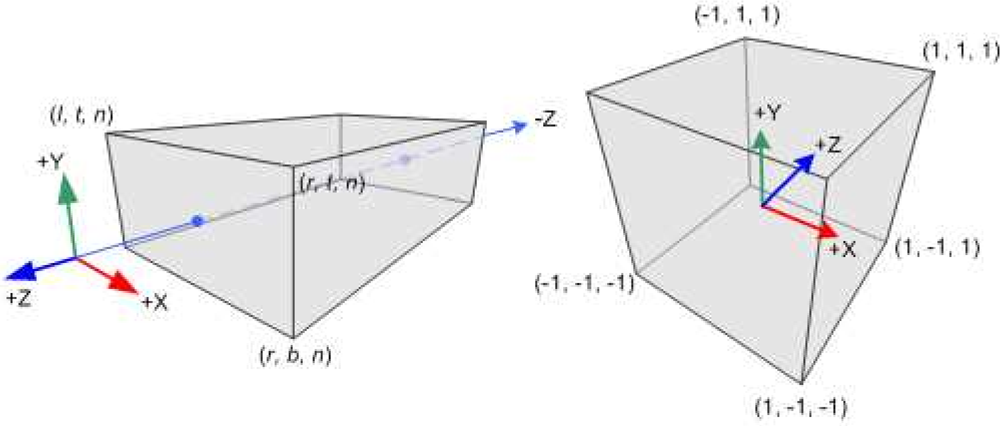
Affine transforms allow you to reposition the point cloud inside the 3D space.

Open eVision provides you with the following basic transformations:

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

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

- □ An orthographic projection transforms a volume of space in the shape of a rectangular parallelepiped (and the points it contains) into the canonical view (a cubic space of size 2 and centered on the origin).



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

Reducing a Point Cloud

Cropping

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

Open eVision provides the following cropping functions:

- `ESimpleCropper`: simple cropping on the X, Y and/or Z coordinates (aligned rectangle 3D region)
- `ERectangularCropper`: cropping the points outside (or inside) an oriented rectangular parallelepiped
- `ESphericalCropper`: cropping the points outside (or inside) a sphere.
- `EPlaneCropper`: cropping the points depending on their position with respect to a plane

These classes produces a new point cloud with the selected points.

Decimation

The random decimator, `ERandomDecimator`, decimates a point cloud by copying a specified number of points, randomly selected, to a new point cloud.

Specify the number of points to keep as parameter of the constructor.

```
EPointCloud pc;
pc.LoadPCD("c:\\images\\data.pcd");
// Explicitely decimate the point cloud
ERandomDecimator decimator(5000);
EPointCloud pcDecimated;
decimator.Decimate(pc, pcDecimated);
pcDecimated.SavePCD("c:\\images\\decimatedData.pcd");
```

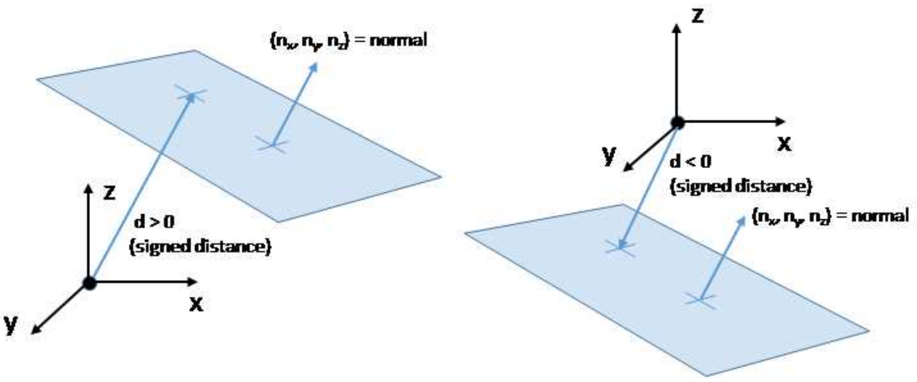
Managing Planes

E3DPlane

A plane can be represented as an `E3DPlane` object.

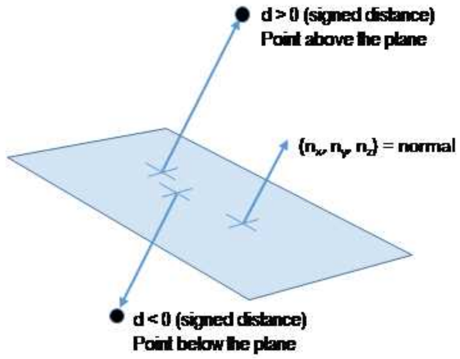
This plane is characterized by:

- Its normal which is a vector of norm 1, perpendicular to the plane.
- Its signed distance from the origin, which is the smallest distance from the origin to the plane. The signed distance is positive when the vector binding the origin to the closest point on the plane has the same direction as the normal and is negative when it has the opposite direction.



Once a plane is defined, you can measure the signed distance between this plane and any point in the space (using the method `DistanceTo()`):

- A positive distance means that the vector connecting the plane to the point has the same direction as the normal.
- A negative distance means that the vector has the opposite direction.



EPlaneFinder

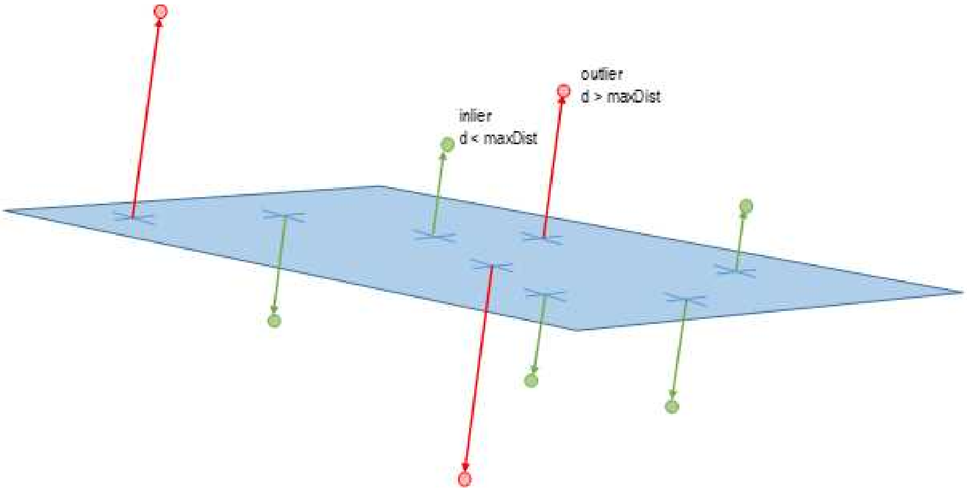
You can search for a plane in a point cloud using the object `EPlaneFinder` object.

The main parameters of this object are:

□ The maximum distance between the searched plane and a point that belongs to this plane.

□ The expected ratio between the numbers of inliers and the total number of points in the point cloud.
- An **inlier** is point that belongs to a plane (closer than this maximum distance).
- An **outlier** is a point that is not an inlier.

The picture below illustrates how points of the space are classified as inliers (in green) and outliers (in red) according to their distance to the searched plane.

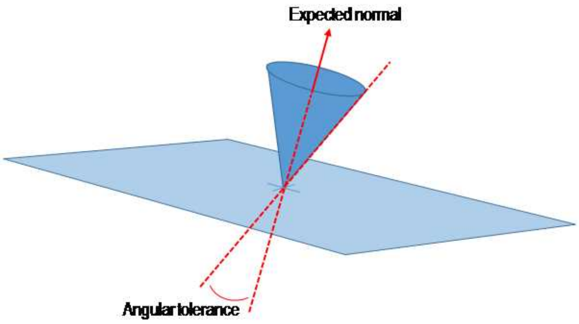


A EPlaneFinder object produces a E3DPlane object. The algorithm searches for a plane containing as many inliers as possible. This plane is the biggest plane if the samples are evenly distributed.

The maximum distance between the plane and the inliers is a mandatory parameter: it should include the deviation due to the noise but also take warpage into account.

The parameter that specifies the ratio of inliers with respect to the total number of points has a default value of 0.3, meaning that we estimate that about 30% of the points belong to the plane. This parameter is not as critical as the maximum distance but it affects the maximum time the algorithm will spend in searching a plane as well as its robustness.

Optionally, you can specify the expected normal vector to the plane to search. In that case, you should also specify an angular tolerance with respect to this expected direction.



When an expected normal is specified, the algorithm only searches for a plane that satisfies the condition. Setting this condition might speed up the plane search.

TIP

Finally, it is important to note that, by default, the `EPlaneFinder` decimates the input point cloud to accelerate the search. The default decimator reduces the input point cloud to 10000 points. Alternatively, you can disable this decimation, or you can decimate a point cloud explicitly, by using an `ERandomDecimator` object and use the decimated point cloud as input for the `EPlaneFinder`. In this case you should disable the default decimator.

Once the main plane is found, a fit is done on all the inliers points and the result is returned (see `EPlaneFitter` below).

EPlaneFitter

The `EPlaneFitter` operator computes a fit on all the points of a point cloud and returns a `E3DPlane` object.

Aligning

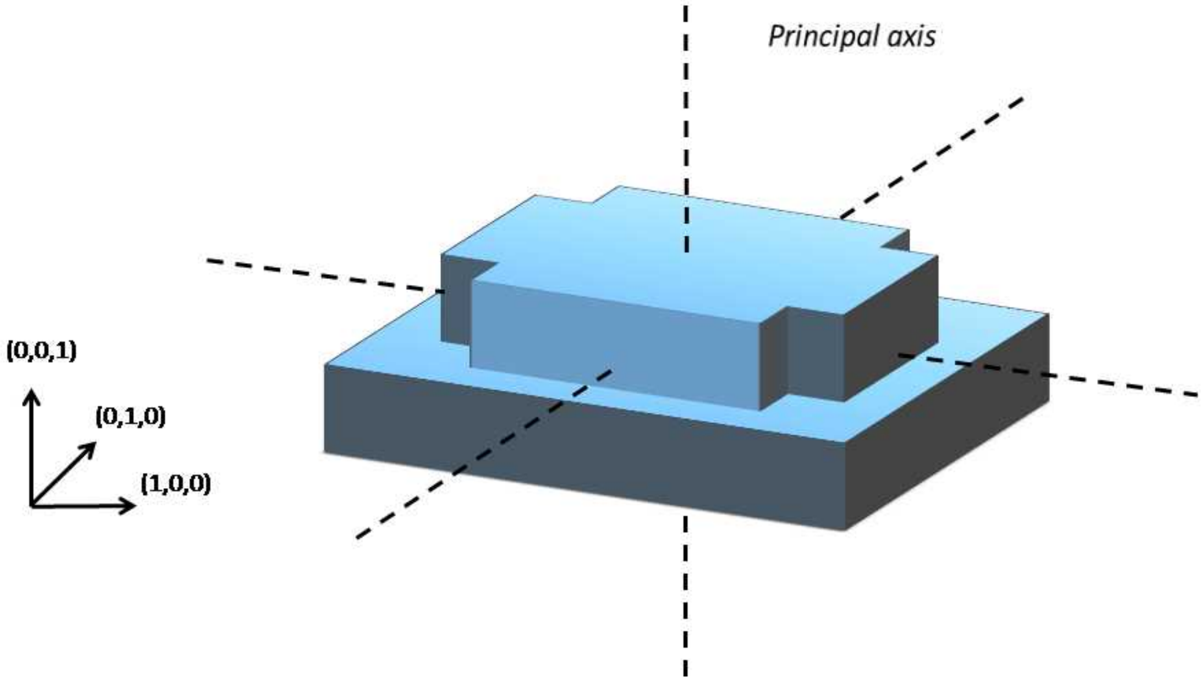
EPrincipalAxisExtractor

The `EPrincipalAxisExtractor` computes the “principal axis” of an object from a point cloud (`EPointCloud`) and returns a `E3DTranformMatrix` containing a solid transformation that defines a new orthogonal basis.

This new orthogonal basis has the following characteristics:

- The center is the center of gravity of the point cloud.
- The axis are oriented along the “principal axis” of the object. This is the result of the “PCA” calculation (principal axis analysis).
- The directions of the axis are selected so that the new basis is as close as possible of the basis defined by the reference transformation.

The next figure illustrates the orientation of the principal axis of an object.



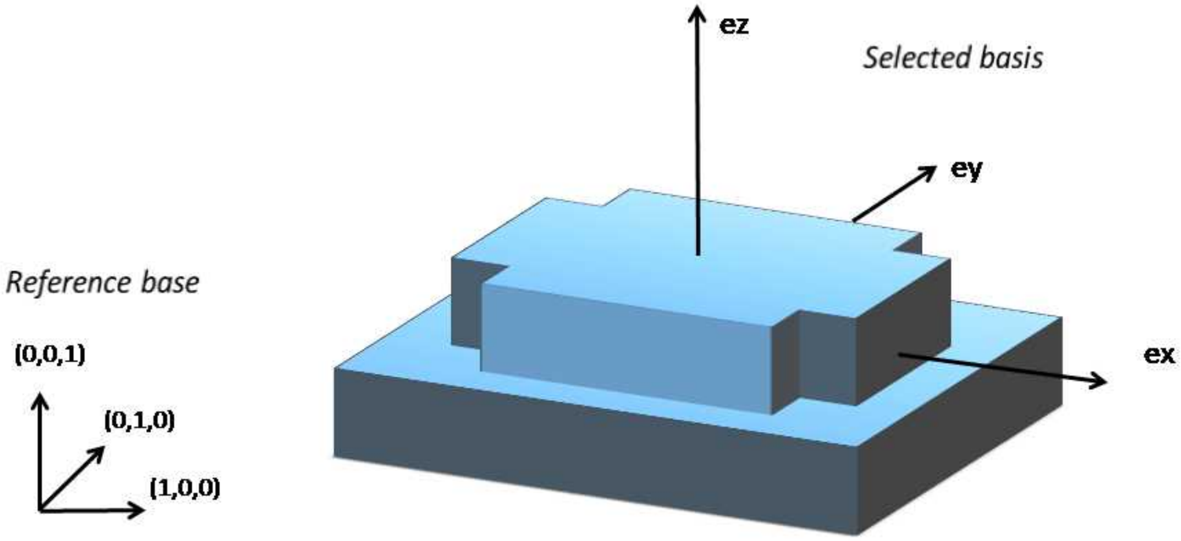
The principal axis extraction is done using the `Extract()` method that takes a `EPointCloud` as input and returns an `E3DTransformMatrix`. Optionally, you can pass 3 other output parameters by reference to retrieve the value of the standard deviation along the 3 principal axis.

You can use the returned `E3DTranformMatrix` object to transform the 3D coordinates of a point. For example, apply the transformation matrix to the origin (0, 0, 0) to return the center of gravity of the object.

Specification of a reference transformation

The reference transformation is an optional parameter of the `EPrincipalAxisExtractor` object. It defines a reference basis used to select an orthogonal basis out of the principal axis. The selected basis will be the closest to the reference basis.

TIP
If no reference transformation was supplied, the default reference basis is ((0, 0, 1), (0, 1, 0), (0, 0, 1)), that corresponds to the identity transformation. On the figure below, the default reference basis determines the direction of the axis **ex**, **ey** and **ez**.



EFeaturesAligner

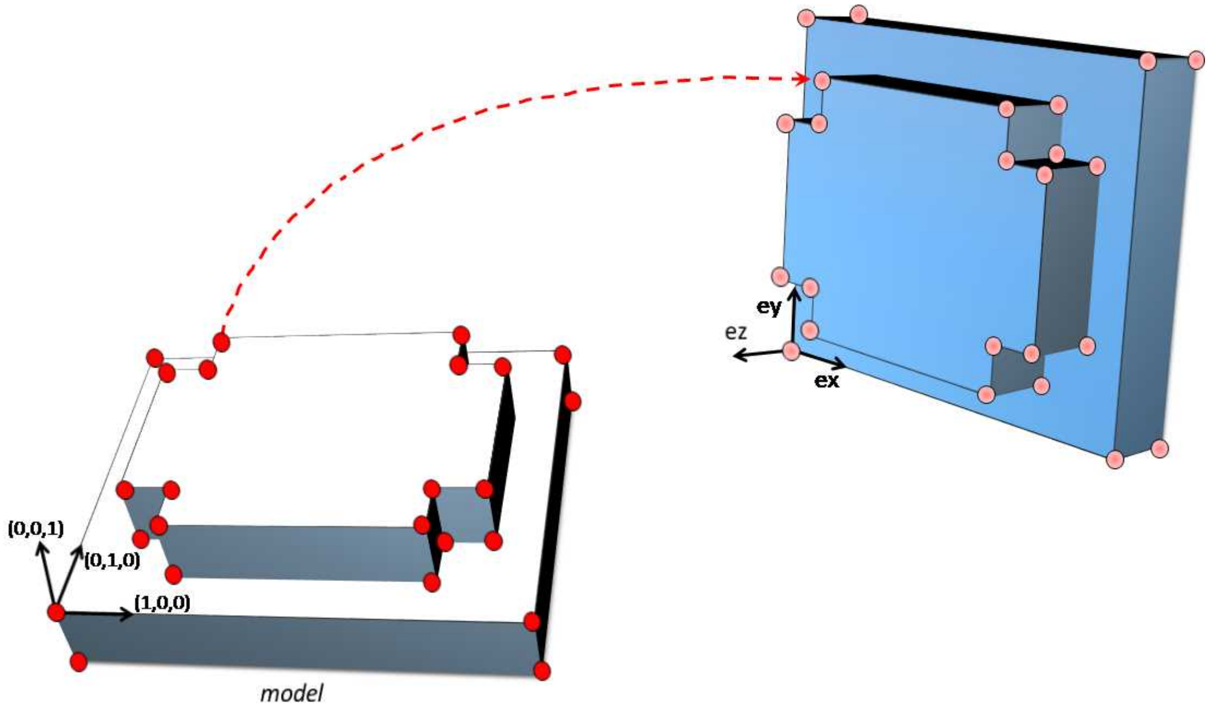
A EFeaturesAligner object finds the best transformation that maps a list of points to another list of points.

- The first list of points is called the "model". It is stored in the EFeaturesAligner object.
- The second list of points is called "measured points". It is passed as a parameter to the Compute() method. If successful, the result of this method is a E3DTransformMatrix object.
- The 2 lists should form matching pairs. In other words, the first point of the first list matches the first point of the second list, the second point of the first list matches the second point of the second list, and so on…

With the Polarity parameter, you can define which transformation is returned. It can be either:

- □ The one that moves one point from the first list (the model) to the second list of points (the measured points) if the polarity parameter is set to EAlignmentPolarity_ModelToMeasured (default).
- □ The one that moves a point from the second list (the measured points) to the first list (the model) if the polarity parameter is set to EAlignmentPolarity_MeasuredToModel.

The figure below illustrates the computation of the alignment transformation. In this example a model is aligned to an object using the coordinates of their corners.



Once the transformation is computed, use the method GetOrthoBasis of the E3DTransformMatrix object to get the basis (**ex**, **ey**, **ez**) and the center point **t** that defines the new basis.

You can also apply the computed transformation on any 3D point as illustrated in the code below.

```
EFeaturesAligner alignTool;
E3DTransformMatrix alignBase;
E3DPoint ex, ey, ez, t;
std::vector<E3DPoint> model3d;
std::vector<E3DPoint> points3d;
// add points to model3d and points3d
// ...
alignTool.SetModelPoints(model3d);
alignBase = alignTool.Compute(points3d);
// Get the orthogonal basis and store it in ex, ey, ez and t
alignBase.GetOrthoBasis(ex, ey, ez, t);
// Applying the transformation on point P1, results in point P1b
E3DPoint P1 = E3DPoint(...) ;
E3DPoint P1b = alignBase*P1;
```

As you can see, the application of the transformation on a point is simply done by multiplying the transformation matrix by the point (as done in the example above).

On the other hand, if you need to transform a point cloud or a list of points, it is more efficient to use the ApplyTransform() method of an EAffineTransformer object.

Mesh

A mesh is a geometric representation of a 3D surface. The surface is defined by a triangle mesh connecting the 3D points. Like a point cloud, a mesh is expressed in the metric space.

Like a point cloud, you can generate a mesh from a depth map and use it to produce a ZMap.

Generation

An `EMesh` object is generated from a depth map using the `EDepthMapToMeshConverter` class.

Like `EDepthMapToPointCloudConverter`, this class uses a calibration model to transform the depth map pixels to 3D world positions. In addition, the depth map pixel connectivity is used to build the triangle mesh. Adjacent pixels produce surface triangles.

Use `SetCalibrationModel()` to select a calibration model and the method `Convert()` to generate an `EMesh` from an 8 bits or 16 bits depth map.

Access and usage

In an `EMesh` object the 3D world positions are stored as an `EPointCloud` (accessible through the method `GetPointCloud()`). The triangle mesh is stored as an array of point indexes, where 3 consecutive indexes define a triangle. The method `GetTriangleIndexes()` provides a read-only access to the triangle mesh.

You can use either the Open eVision proprietary format to save and load `EMesh` objects using the `Save()` and `Load()` methods, or use the STL standard file format (https://en.wikipedia.org/wiki/STL_(file_format)) using the `SaveSTL()` and `LoadSTL()` methods which respectively write to and read from ASCII STL files.

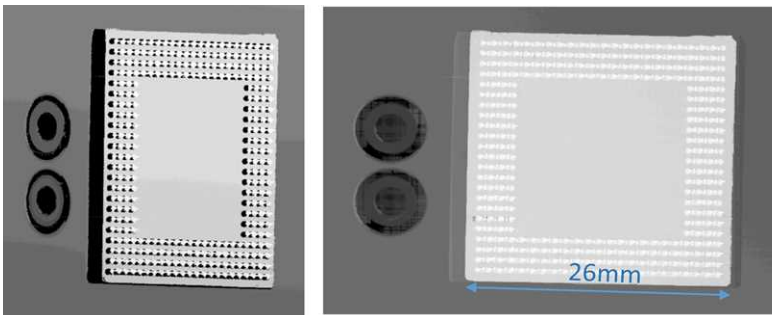
You can use an `EMesh` to produce a ZMap (see "Generating a ZMap" on the next page). Because an `EMesh` represents a surface, the so generated ZMap can show better continuity and less undefined pixels.

ZMap

Generating a ZMap

A ZMap is the projection of a point cloud or a mesh on a reference plane, with the distance coded as gray scale values:

- □ They are grayscale images, compatible with all Open eVision 2D libraries.
- □ They are distortion free, with affine transformation from/to metric coordinate system.



A depth map (left) and the corresponding ZMap (right), with default generation parameters and undefined pixel filling enabled

All Open eVision 2D processing are available on ZMaps: filtering, thresholding, blob extraction, measuring with EasyGauge, model matching with EasyFind or EasyMatch…

The `EPointCloudToZMapConverter` class implements the conversion from a point cloud to a ZMap (`EMeshToZMapConverter` converts a mesh to a ZMap). With all parameters at default value, the `Convert()` method automatically chooses the projection plane, the orientation, the map size and the resolution.

Several methods are available to further control the conversion:

- `SetReferencePlane()` defines a world space projection plane. The values of the ZMap pixels are the distance of the point cloud to that reference plane.

  By default, the reference plane crosses the origin and is perpendicular to the world Z axis. The plane is defined as a `E3DPlane` object.

- `SetOrientationVector()` sets a world space vector representing the expected direction of the X (width) axis of the ZMap.

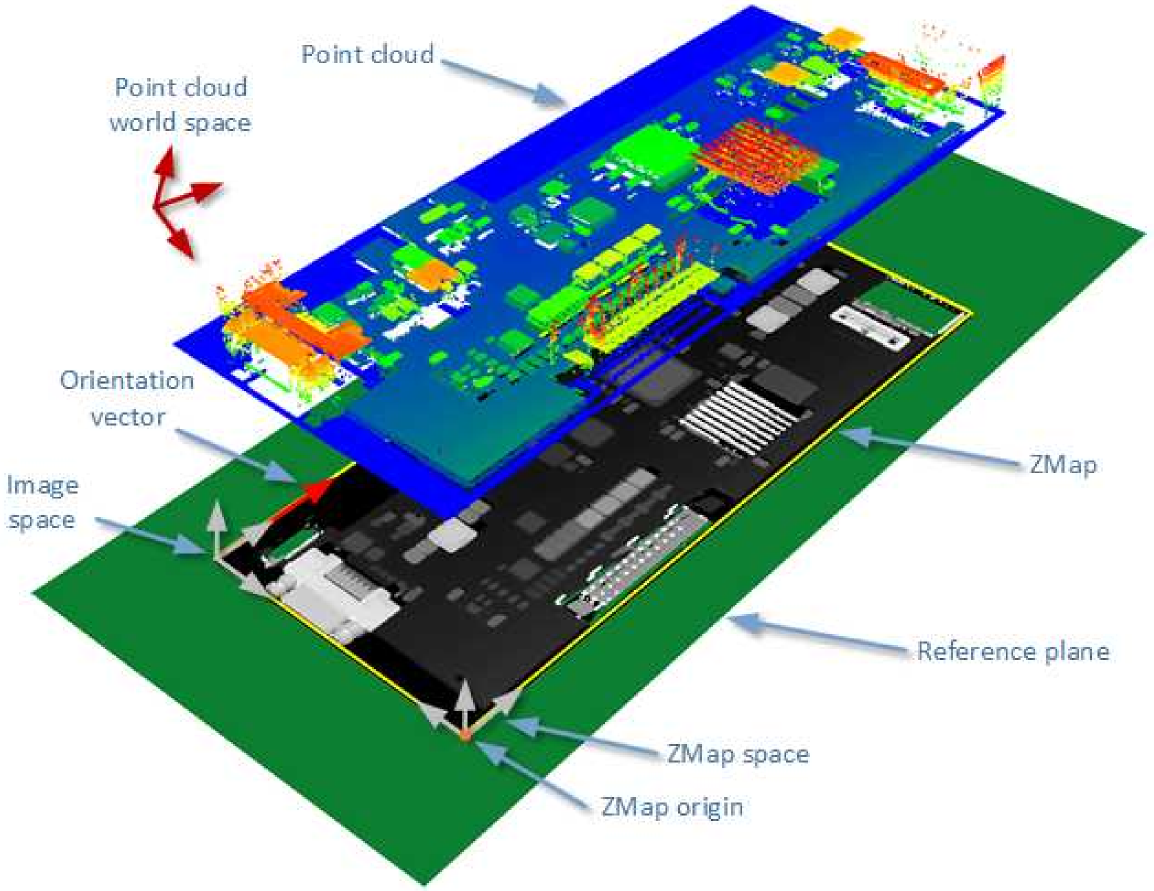
  The orientation vector allows to “rotate” the object around the normal of the reference plane.

- `SetOrigin()` specifies the world position that is on the ZMap lower left pixel (0, 0).
- `SetMapSize()` defines the resolution (number of pixels in X and Y axis) of the generated ZMap.
- `SetMapXYResolution()` adjusts the X and Y resolution of the ZMap pixels, in world space unit per pixel (for example mm/pixel). This value is used to compute the ZMap size (width and height), depending on the projected size of the point cloud on the reference plane.

- SetMapZResolution() sets the Z resolution, in world space unit per pixel unit (gray value). The Z resolution is used to compute the transformation of the distance to the reference plan to the integer 8, 16 or 32 bits pixel value.
- EnableFillMode() and SetFillMode() control the options used to fill the "hole" in the ZMap. A hole exists when no 3D point is projected in the ZMap at a pixel position.

The methods SetReferencePlane(), SetOrientationVector() and SetOrigin() are used to setup the transformation between the world space and the ZMap space. This transformation is rigid (distances are kept).

Alternatively, it is possible to directly set that transformation with the method SetWorldToZMapTransform() using a rigid matrix as parameter. In that case, the reference plane, the orientation vector and the origin parameters are ignored.

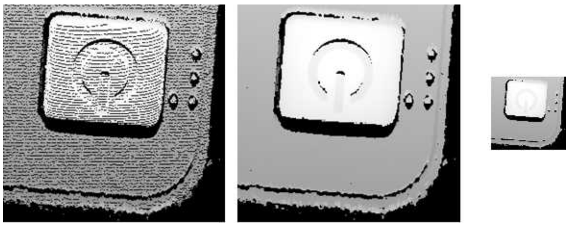


The projection of a point cloud on a ZMap,
showing 3 coordinate systems: the world space, the ZMap space and the pixel space.

the `Convert()` method performs the effective projection of a point cloud (`EPointCloud`) or a 3D object (`EMesh`) to the 8, 16 or 32 bits ZMap.

When generating a ZMap from a point cloud, only individual points are projected on the ZMap. Depending on the point cloud density and the ZMap resolution, some regions of the ZMap may remain “undefined”. To get around this problem, adjust the resolution of the ZMap (`SetMapXYResolution` method) to remove “holes” on the ZMap.

By default, the point cloud to ZMap converter performs a filling algorithm. This process tries to replace undefined pixels with locally interpolated values.



Left: high resolution ZMap, the pixel scale exceeds the point cloud density
Center: the same generator parameters with the filling enabled
Right: a reduced ZMap scale/resolution, without filling

As a mesh defines a surface, its triangles are projected onto the ZMap plane. Thus, the generated image shows better continuity and less undefined pixels. However, the generation of a ZMap from an `EMesh` is slower than from an `EPointCloud`.

Creating a Point Cloud from a ZMap

To generate a point cloud from a ZMap, use the `EZMapToPointCloudConverter` class.

The `Convert()` method takes:

- A ZMap source
- A `EPointCloud` destination.
- 2 optional parameters:
  - □ An `ERegion` that defines the domain of the ZMap to convert.
    By default, Open eVision uses all the defined pixels of the ZMap generate the point cloud.
  - □ A parameter to select the world space (by default) or the ZMap space to store the resulting positions in the point cloud.

Managing the Coordinates

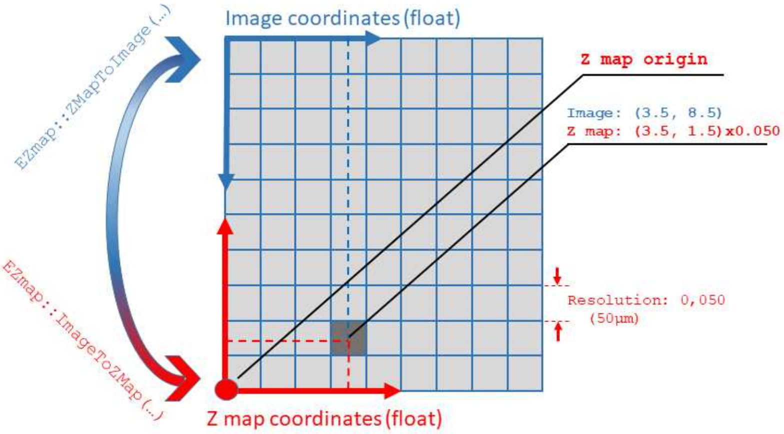
Coordinate systems on a ZMap

A ZMap has multiple coordinate systems:

- □ The **world space** system is the original, metric space from which the ZMap has been generated. Point clouds and meshes are expressed in the world coordinate system.
- □ The **ZMap space** is defined by a rigid transformation of the **world space**. The basis linked to this transformation is attached to the lower left corner of the ZMap.
- □ The **image space** is the system attached to the image representation of the ZMap. Its origin is the upper left corner of the ZMap and its unit length is one pixel along the X and Y axis.

The transformations between:

- □ The **image space** and the **ZMap space** include a scale factor.
- □ The **ZMap space** and the **world space** are solid transformations.



EZMap

The EZMap object exposes a set a methods to convert coordinates between world, ZMap and image spaces:

- □ ImageToZMap converts a 2D position in the image to ZMap coordinates.
- □ ZMapToImage is the reciprocal operation and converts a ZMap position to an image position.
- □ ZMapToWorld is a method to transform positions from the 3D ZMap space to the 3D world space. The world space is the original point cloud or mesh space.
- □ WorldToZMap is the reciprocal operation, converting from world space to ZMap.
- □ ImageToWorld and WorldToImage combine the functions above to transform directly from image space to world space (or the other way).

These methods only perform geometric transformations between the various coordinate systems and do not access the actual ZMap gray scale values.

The functions that accesses the pixel values are:

- □ GetWorldPositionFromPixelPosition() is a method transforming the actual pixel value at integer position (u, v) to the original world space. This method queries the ZMap internal representation to get the pixel value w and transform the pixel space (u, v, w) coordinates to a world space position.
- □ GetPixelPositionFromWorldPosition() is a method to get a pixel value from a world position. The world position is projected on the ZMap and the pixel value is returned. If the world position is outside the ZMap domain, the method returns FALSE.

3D Viewer

Use the E3DViewer class to easily create an interactive 3D display. The viewer displays point clouds, meshes and ZMaps.

You can create E3DViewer as a child of an existing window or without a parent. In that last case, a new window is created.

> **NOTE**
> As E3DViewer uses OpenGL interface, it requires a compatible display device.

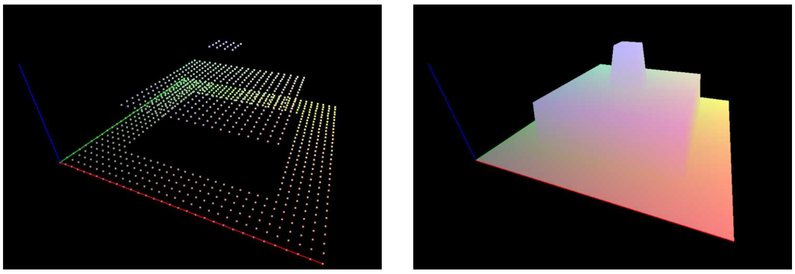
Call the ConfigureRenderSource() method with a valid 3D geometry to display it. At each call, ConfigureRenderSource() replaces the current displayed object.

The supported classes are:

- EPointCloud
- EMesh
- EZMap8, EZMap16 and EZMap32f.

TIP

When you configure a new render source with ConfigureRenderSource(), the view point is automatically adapted to display the whole object.

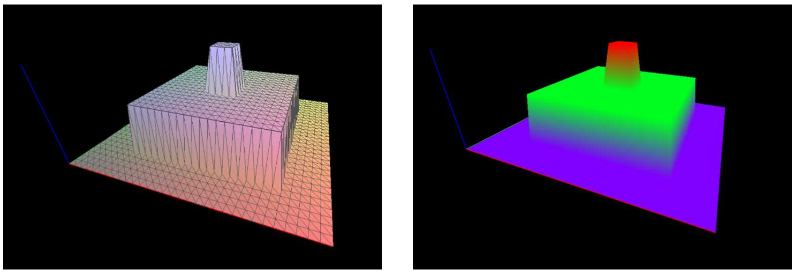


E3DViewer in action: point cloud display (left) and 3D object display (right)

To display the geometry in false colors:

- Use the GenerateColors() method that computes RGB colors from the position of the vertices.
- It supports various predefined color ramps.
- Use the SetColors() method to use custom colors (one EC24 entry is requested for each render source vertex).

Use the methods SetPointSize(), SetWireframeMode() and SetRenderDecimationLevel() to adjust the rendering attributes.



E3DViewer in action: wire frame enable (left) and HueFromZ color ramp (right)

In the 3D navigation window, use the mouse as follows:

- Press the left button to rotate the image horizontally and vertically.
- Press the right button to translate the image horizontally and vertically.
- Use the wheel to zoom in and out.

In addition, use the following keys:

- Press **R** to reset the viewer.
- Press **W** to show or hide the triangle edges (in wire frame mode).
- Press **+** and **–** to increase or decrease the point size.

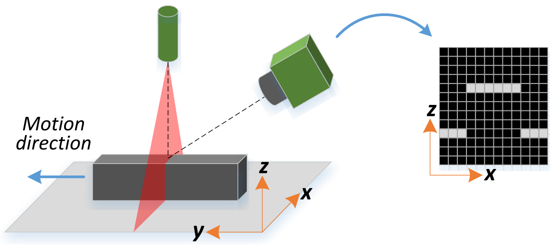
Use methods `SetViewTarget()`, `SetViewingAngle()` and `SetViewDistance()` to change the view point programmatically.

Use the methods `SetAutoRotate()` and `StopAutoRotate()` to manage the automatic rotation of the 3D view.

0.2. Easy3DLaserLine - Laser Line Extraction and Calibration

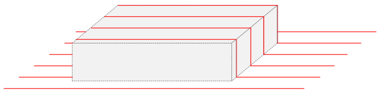
Laser Triangulation

In a laser-line triangulation system, a laser line is projected on the object to measure. A camera is looking at the laser line from a different point of view. The line deformation observed by the camera contains the shape information of the measured object.



The scanning of the object consists in moving it under the laser line and recording multiple images.

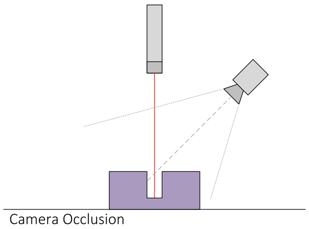
From the scanning you can reconstruct its 3D shape.



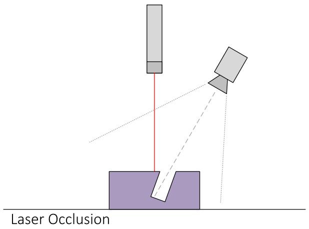
Occlusions

Using the laser triangulation method, the laser may be unable to reach some parts of the object or the camera may be unable to view them. This is called occlusion.

- On the left illustration, the camera does not see the bottom of the hole, inducing camera occlusion.
- On the right illustration, the laser does not reach the bottom of the hole, inducing laser occlusion.



Camera Occlusion



Laser Occlusion

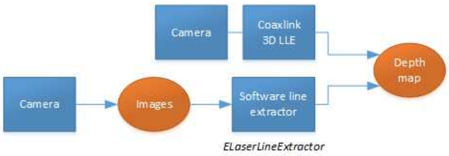
TIP
You can limit or avoid occlusions by using advanced scanning methods, for example by using two cameras or two lasers.

The Laser Line 3D Acquisition Pipeline

The 3D acquisition pipeline starts with the acquisition of a laser line profile and ends up with the point cloud, mesh or ZMap.

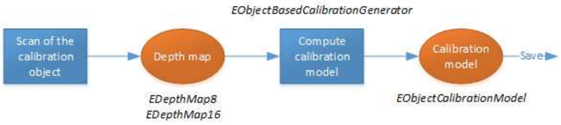
The source material for 3D processing is the depth map, coming from a Coaxlink Quad 3D-LLE or generated from a list of images.

3 types of depth map are available, one for each different pixel coding scheme (8, 16 or 32 bits).



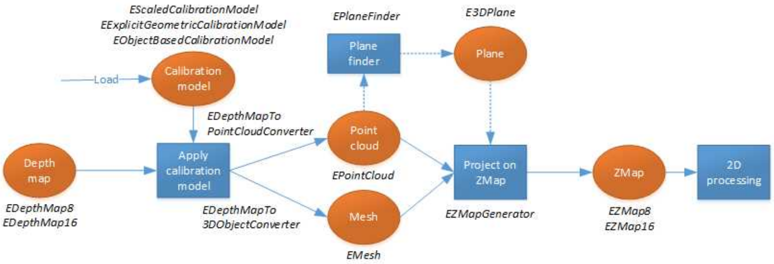
The generation of a depth map, from a hardware or a software source

Some processing methods can use the depth map directly, but most measurement and matching processes need metric, distortion-free representations. Calibration of the laser triangulation setup is therefore required. Calibration is used to turn the depth map into a point cloud or mesh expressed in a metric space that we call “world space”.


EObjectBasedCalibrationGenerator
Scan of the calibration object
Depth map
Compute calibration model
Calibration model
Save
EDepthMap8
EDepthMap16
EObjectCalibrationModel

The generation of an object based calibration model, from a scan of the reference object

A point cloud is a list of 3D points, expressed in a world space coordinate system. The point cloud can be projected on a plane, producing a ZMap, which is a convenient and effective representation for 2D processing with a metric scale.



The workflow from the depth map to the ZMap

The following sections describe the classes and methods useful for a 3D workflow. The Use Case - Measuring a Remote Controller goes through this processing pipeline.

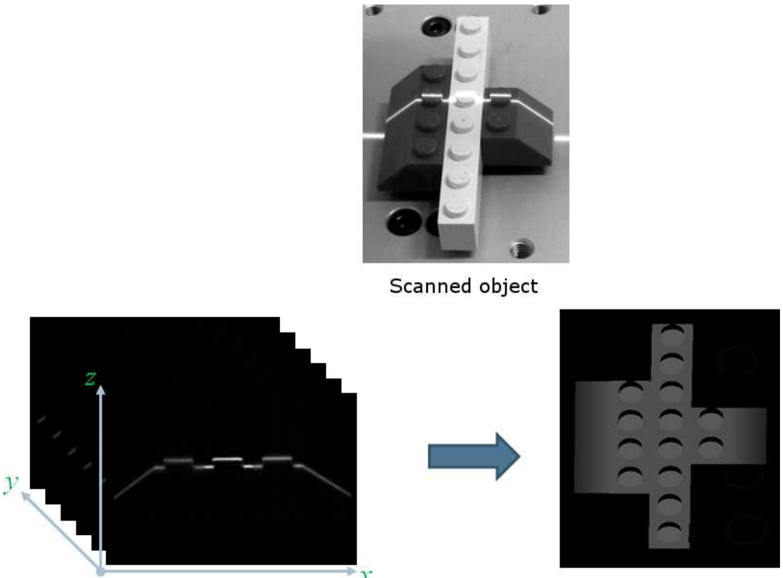
Laser Line Extraction

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

The objective of an LLE algorithm is to measure the line position along a vertical profile in every column of a sensor frame, within a user-defined region of interest (ROI).

For every step of the object position, the detection analyzes each column of a frame individually and produces a row of output positions, stored as gray values.

The figure below illustrates a depth map generation.



Scanned object

Profiles

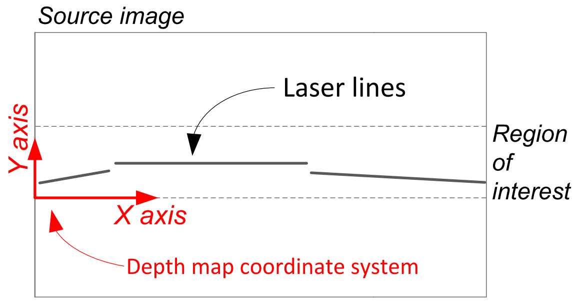
Depth map

The `ELaserLineExtractor` class provides the laser line extraction functionality in Open eVision. It implements several algorithms to extract the laser line (see below for more details):

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

TIP
You can also set a threshold to exclude pixels with low intensity.

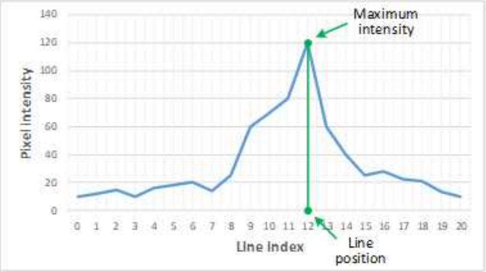
The line position returned by the laser line extraction algorithms is relative to the bottom of the region of interest. So, values in the depth map range from 0 (bottom of the ROI) to the height of the ROI.



Laser line extraction methods

Maximum detection

The maximum detection algorithm analyzes all the pixels in a ROI column to determine the one with the maximum intensity. The figure below shows the laser line position on a given ROI column.



Maximum detection on a ROI profile

We also recommend to include in the processing chain:

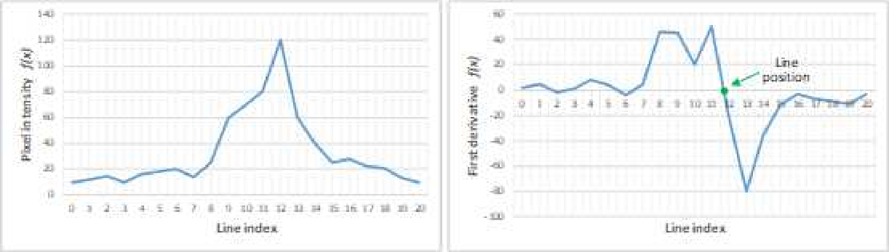
- □ A low-pass filter to reduce the high frequency variations in the image.
- □ A threshold to eliminate the background noise from the sensor.

Peak detection

The peak detection algorithm relies on a discrete simplification of the first derivative function.

$$\frac{df}{dx} = f(x+1) - f(x-1) = f'(x)$$

The f '(x) outputs the slope of a given f (x) along the x.



f (x) and f '(x) plots

We compute the line position by detecting where f ’(x) changes its signal based on the two-point form line equation:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

where (x_1, y_1) and (x_2, y_2) are two points on the line with $x_2 \neq x_1$, we obtain the following equation for y = 0:

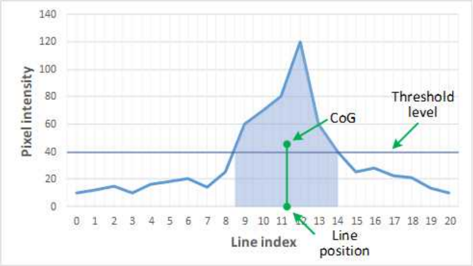
$$x = \frac{x_1 y_2 - x_2 y_1}{y_2 - y_1}$$

Center of gravity

The center of gravity (CoG) method uses an algorithm that calculates the center of mass of an image object. Also know as "centroid of plane figures", the CoG is obtained by the following equations:

$$\overline{X} = \frac{\sum ax}{\sum a} \qquad \overline{Y} = \frac{\sum ay}{\sum a}$$

where $\overline{X}$ and $\overline{Y}$ are the coordinates of the CoG and a is the pixel intensity along the x and y axes.

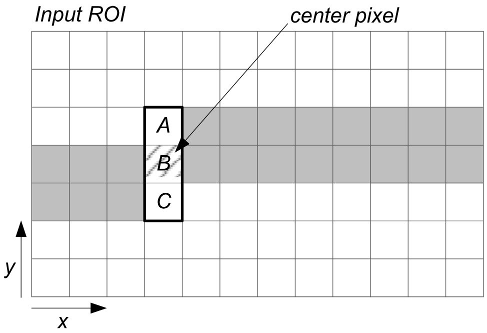


Center of gravity on a ROI profile

Low-pass linear filter

Optionally, you can apply a low-pass linear filter in front of the line extraction in order to reduce noise and high frequencies in the image.

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



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

Calibration

The calibration is used to apply the transformation between a depth map and a point cloud or a mesh.

There are 3 ways to setup this conversion:

- Apply a simple scale on the pixel coordinates of the depth map (`EScaleCalibrationModel` class)
- Use the explicit geometric model (`EExplicitGeometricCalibrationModel` class)
- Use the object-based calibration approach (`EObjectBasedCalibrationModel` class)

These models share the same base class `ECalibrationModel` and exposes the method `Apply()`, which is used to apply the conversion between a depth map pixel and a 3D point. It takes as input the coordinates of one point in a depth map and it returns the coordinates of the corresponding point in the 3D space.

The method `Apply` is not aware of the possible mirroring of the corresponding depth map and cannot make use of `EDepthMap::AxisSystemType` (see below). If necessary (when the corresponding depth map is vertically mirrored) the y coordinates should be flipped before calling the `Apply` method.

- The class `EDepthMapToPointCloudConverter` generates a point cloud from a depth map, using one of the calibration models.
- The class `EDepthMapToMeshConverter` generates a mesh from a depth map, using one of the calibration models.

By convention:

- The origin of the referential is the lower-left corner of the depth map.
- The center of the first pixel at the lower-left corner is at x = 0.5 and y = 0.5.
- The center of the pixel at the upper-right corner is at x = width - 0.5 and y = height - 0.5 where width is the width of the depth map and height is its height.

Mirrored depth maps

By default, Easy3D considers that the origin of the 3D axis of the depth map is the bottom left of the internal image buffer, and the Y axis is pointing up. This means that the depth map image is not seen as vertically mirrored compared to the real world image of the scanned object.

Nevertheless, depending on your acquisition setup this mirroring can happen (for example if the direction of the scan is inverted).

If this is your case, you can set the `EDepthMap::SetAxisSystemType` to `EAxisSystem_UpperLeftCorner`, meaning that the origin of the 3D axis is on the upper left corner and the Y axis is pointing down.

This value changes the behavior of the methods :

- `EObjectBasedCalibrationGenerator.Compute`
- `EDepthMapToPointCloudConverter.Convert`

□ EDepthMapToMeshConverter.Convert

Scale calibration

The scale model (EScaleCalibrationModel) only applies a simple factor on the X, Y and Z axis. These factors are the only parameters of EScaleCalibrationModel.

For depth maps coming from laser triangulation setup, this transformation does not produce corrected, metric points. It's main use is to display depth maps as 3D data with the E3DViewer class.

Explicit geometric calibration

The explicit geometric model (EExplicitGeometricCalibrationModel) defines a simple and ideal laser triangulation setup. The explicit calibration makes some strong assumptions on the setup geometry and can only be used when a minimum set of parameters are known:

□ The angles of the camera and the laser plane, in the counter clockwise direction. The camera angle must be positive.

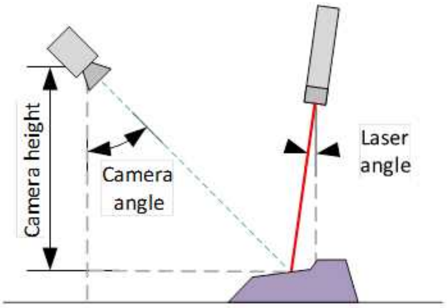
□ The height of the camera above the scanned object.

□ The field of view of the camera defined by the sensor size (mm) and the optical focal length (mm).

□ The physical distance between two line scans of the depth map (depends on acquisition rate and motion speed).

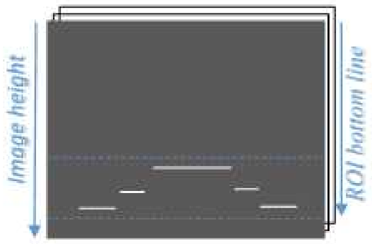
□ The size of the image and the ROI origin used in laser line extraction (between the top (0) and the bottom (height) of the image).

TIP
Use the "Easy3D_Setup_Configuration.xlsx" spreadsheet to compute and check your setup configuration and parameters.

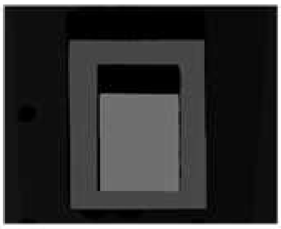


Explicit calibration setup with camera angle, laser angle and camera height

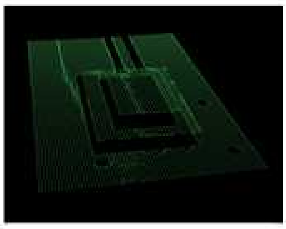
The setup of an explicit geometric calibration uses the constructor of the `EExplicitGeometricCalibrationModel` class.



Camera source images



2.5D Depth map



3D Point cloud

Object-based calibration

Object-based calibration gives real world, metric, coordinates from an arbitrary laser triangulation setup. From the scan of a reference object, the calibration process tries to calculate all the parameters required for the transformation to the world space (position and attributes of the camera, position of the laser plane, relative motion of the object, optical distortion...).

For more details, please refer to the "Object-Based Calibration Guidelines" below section.

Object-Based Calibration Guidelines

Easy3D calibration is a powerful process that uses a single scan of a calibration object to calibrate a laser triangulation setup.

1. The calibration process generates a calibration model.
2. **Easy3D** uses this calibration model to transform the laser profile scans (or depth maps) into metric, distortion free point clouds.

- The calibration model includes all the geometric parameters required for this transformation:
  - □ The relative position of the laser and the camera.
  - □ The projection and the distortion model of the camera.
  - □ The relative motion of the object.

This document explains all the steps involved in the calibration process, from the design of the calibration object to the **Open eVision** API.

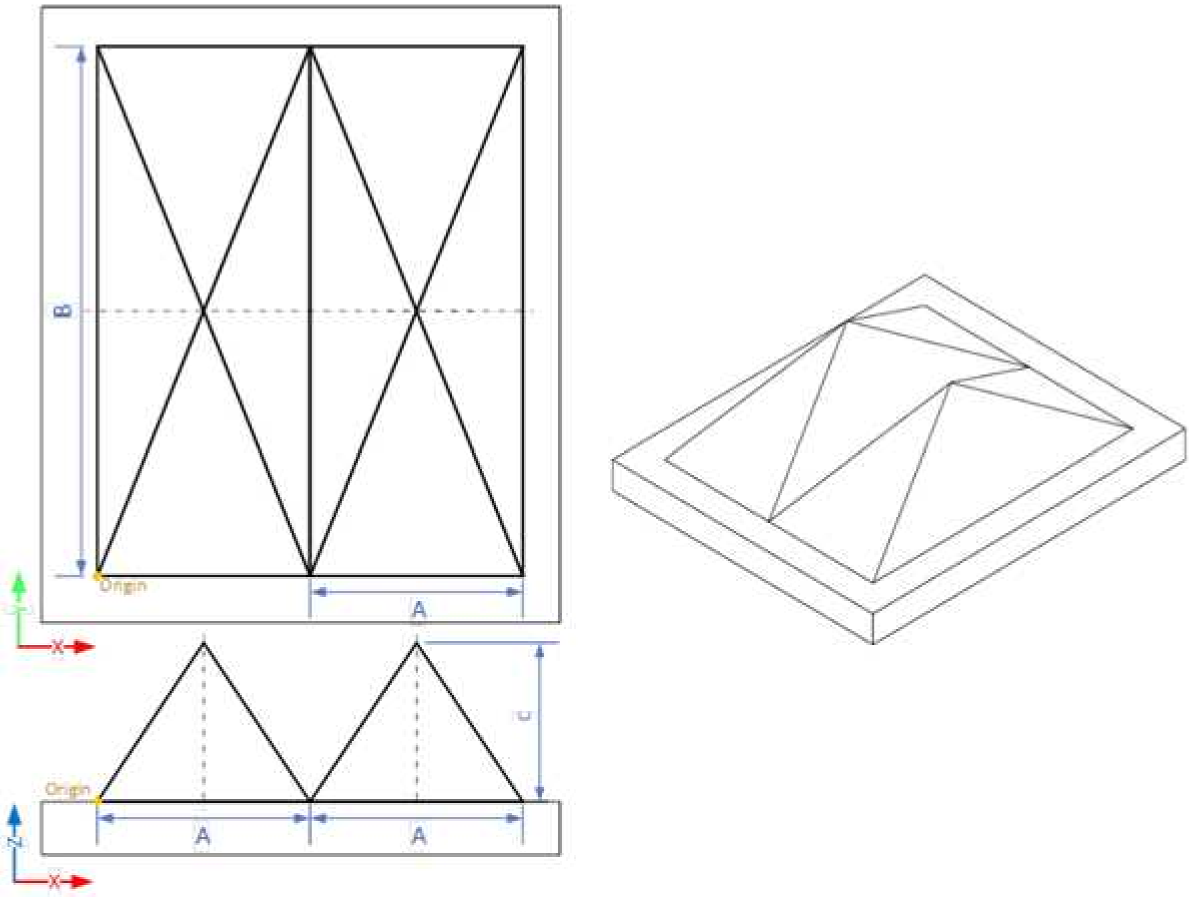
The calibration object

The general principle of **Easy3D** calibration is to match a scan of a known calibration object to its true geometric dimensions.

The double pyramid

TIP
In **Open eVision** 2.7 the “double truncated pyramid” calibration object is recommended over the "double pyramid" model.

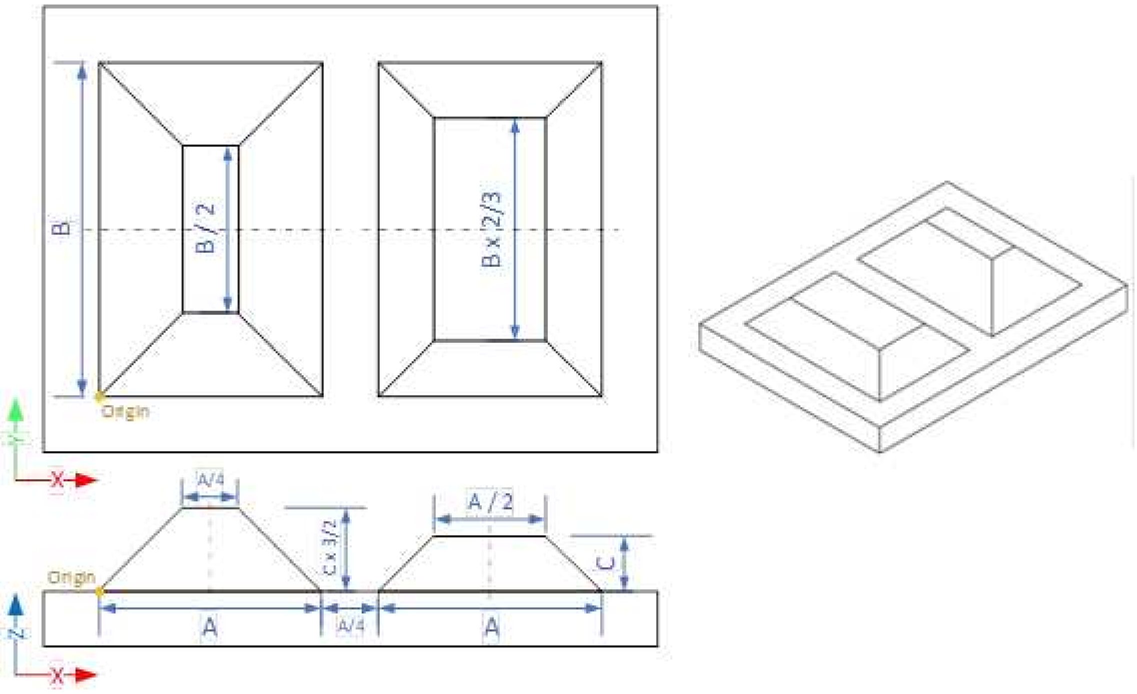
The dimensions of the “double pyramid” calibration object along the X-, Y- and Z-axes are named A, B and C respectively.



The "double pyramid" calibration model

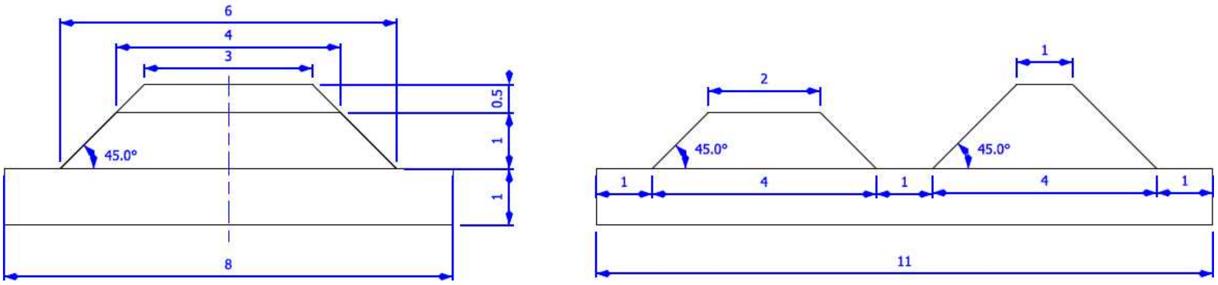
The truncated double pyramid

- The dimensions of the “double truncated pyramid” calibration object the X-, Y- and Z-axes are named A, B and C respectively.
- The design of the double truncated pyramid must follow the ratios given in the illustration below.



The "double truncated pyramid" calibration model (recommended)

- For example, the provided CAD files of the calibration object use A = 4 cm, B = 6 cm and C = 1 cm. The Calibration Object Size, required for the calibration process, are the values A, B and C.



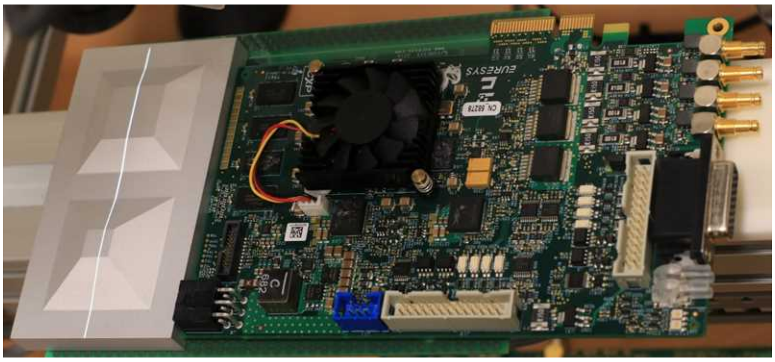
The "double truncated pyramid" calibration model with A = 4, B = 6 and C = 1

Building a calibration object

Overall dimensions

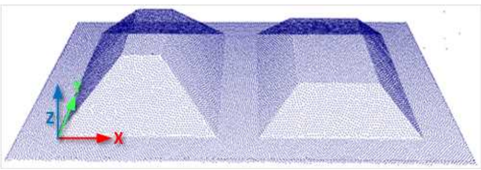
- Manufacture a calibration object that fits the working area of the project.
- For example, if the project targets the inspection of a PCB (a printed circuit board as illustrated), design your calibration object with:
  a. The dimension A or B (it does not matter) similar to the width of the PCB.
  b. The height (C) of only several millimeters.

> **TIP**
> This is not a strict requirement, if the scanned object is slightly larger or smaller than the calibration object, the calibration process is still valid.



A PCB scanning setup with the associated calibration object
The calibration object dimensions (A, B and C) match the width and the height of the PCB

> **TIP**
> There is no constraint on the orientation of the calibration object during the scan:
> - The X-axis can be aligned with the motion direction or with the laser line.
> - After the calibration process, the origin and axes of the 3D calibrated point cloud follow the conventions of the reference design.

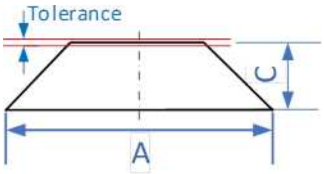


A calibrated point cloud with the origin and the axis of the coordinates system
The 3D origin is located at the external corner of the higher pyramid

Precision and tolerance

The relevant dimensions of the calibration object are the width, the length and the height of the pyramids (called A, B and C in the illustrations).

- The relative dimensions to A, B and C (B/2, A/4...) are important and you must execute them with the same precision.
- The dimensional tolerances are related to the overall expected precision.
If you want to achieve measurements on the point cloud with a precision of 0.01 mm, the manufacturing of the calibration object must have the same precision.
- These tolerances only apply to the pyramids geometry, the calibration process does not use the dimensions of the support.
- The planar surfaces must be flat between 2 parallel planes separated by the target tolerance, as illustrated.



The tolerance of the pyramids sides is defined as the smallest distance between two parallel planes that contain the entire surface

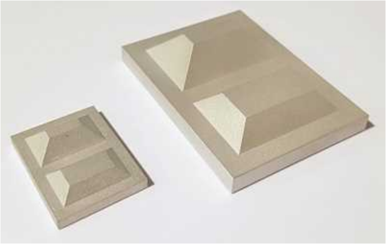
Material and surface finishing

TIP
The goal is to obtain the laser profile as thinnest as possible over the whole object surface with the largest reflected energy.

The build material and the surface finishing are also important and must have:

- □ A good reflectance, with diffuse reflection (no specular reflections).
- □ No transmission and limited diffusion inside the material.

TIP
You can obtain a good surface finishing using aluminum material and blasting. Blasting gives the surfaces a satin gray finish.

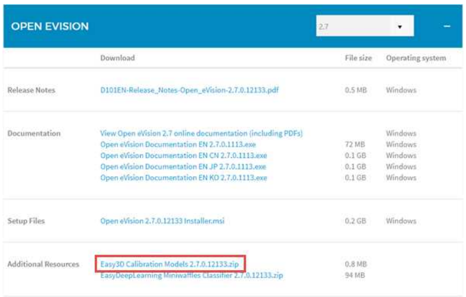


2 aluminum machined calibration objects with a micro-abrasive blasting surface treatment

3D CAD models

The calibration object models are available in various 3D CAD format like STEP, OBJ and STL.

Download these files from the Open eVision download area in the Additional Resources section (www.euresys.com/Support).



OPEN EVISION

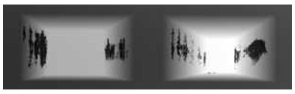
2.7

	Download	File size	Operating system
Release Notes	D101EN-Release_Notes-Open_eVision-2.7.0.12133.pdf	0.5 MB	Windows
Documentation	View Open eVision 2.7 online documentation (including PDFs) Open eVision Documentation EN 2.7.0.1113.exe Open eVision Documentation EN CN 2.7.0.1113.exe Open eVision Documentation EN JP 2.7.0.1113.exe Open eVision Documentation EN KO 2.7.0.1113.exe	 72 MB 0.1 GB 0.1 GB 0.1 GB	Windows Windows Windows Windows Windows
Setup Files	Open eVision 2.7.0.12133 Installer.msi	0.2 GB	Windows
Additional Resources	Easy3D Calibration Models 2.7.0.12133.zip EasyDeepLearning Miniwaffles Classifier 2.7.0.12133.zip	0.8 MB 94 MB	

Download the calibration object models

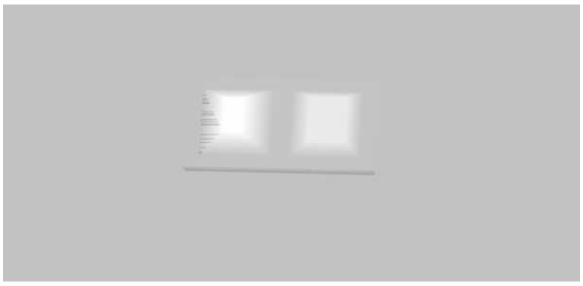
Scanning the calibration object

- The scan of the calibration object produces a depth map.
- To ensure a correct detection of the calibration object and a precise calibration model, you must fulfill the following criteria:
  - □ All faces of the calibration object must be visible on the depth map (this affects the orientation of both the camera and the laser).
  - □ No other object can be higher than the calibration object in the depth map.
  - □ The depth map must have at least 200 x 200 pixels.
  - □ The calibration object must cover at least 50% of the defined pixels of the depth map.
- Examples of bad scans:



Missing pixels on the side faces

Not enough lines



The calibration object is too small on the depth map

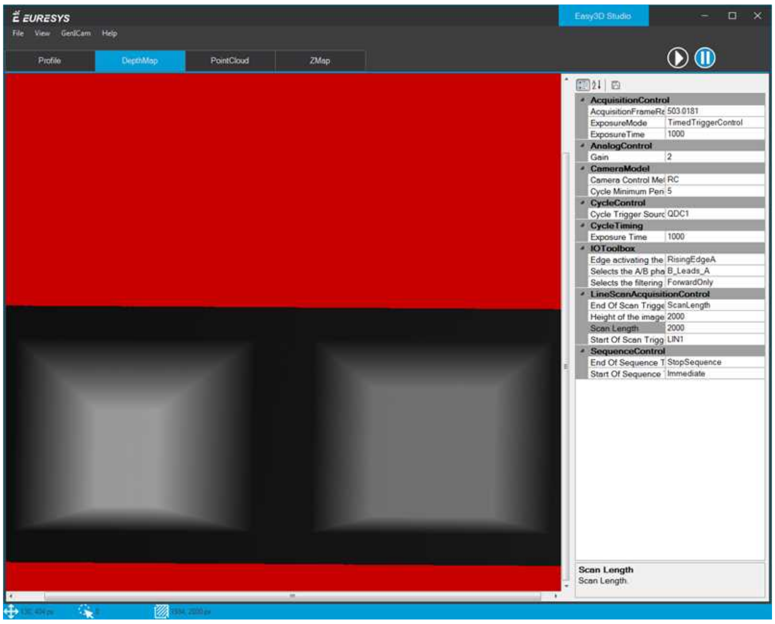
Calibration with Easy3D Studio

Easy3D Studio is a free application that helps you to set up a laser triangulation scanner. You can easily set the acquisition parameters of the **Coaxlink Quad 3D LLE** frame grabber and perform the calibration.

The DepthMap *panel*

This panel displays:

- The scanned image.
- The acquisition parameters on the right side.



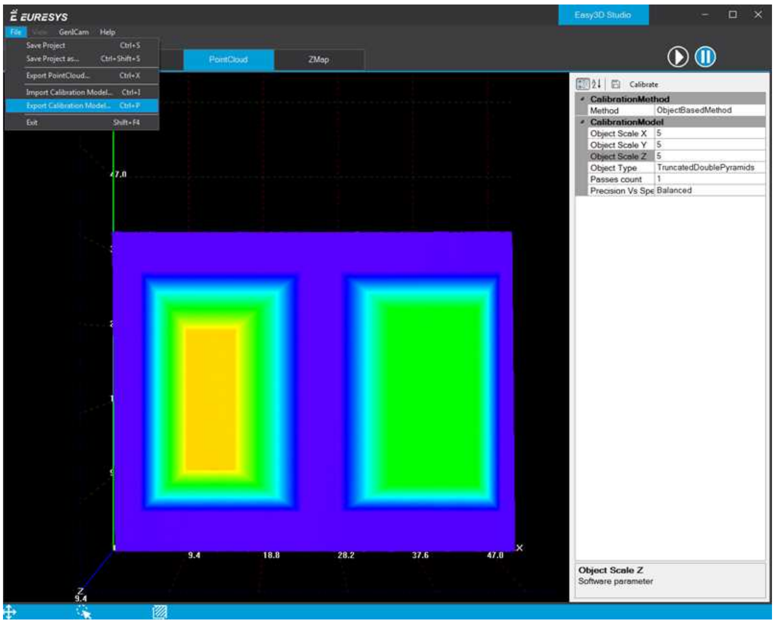
AcquisitionControl	
AcquisitionFrameRa	503.0181
ExposureMode	TimedTriggerControl
ExposureTime	1000
AnalogControl	
Gain	2
CameraModel	
Camera Control Met	RC
Cycle Minimum Peri	5
CycleControl	
Cycle Trigger Sourc	QDC1
CycleTiming	
Exposure Time	1000
IOToolbox	
Edge activating the	RisingEdgeA
Selects the A/B pha	B_Leads_A
Selects the filtering	ForwardOnly
LineScanAcquisitionControl	
End Of Scan Trigge	ScanLength
Height of the image	2000
Scan Length	2000
Start Of Scan Trigg	LIN1
SequenceControl	
End Of Sequence T	StopSequence
Start Of Sequence	Immediate

Scan Length
Scan Length.

The PointCloud panel

This panel displays:

- □ The depth map of the scanned image.
- □ The object-based calibration parameters on the right side.
- □ The Calibrate button computes the calibration model using the last scanned depth map.
- □ When the calibration model is ready, the depth map is transformed into a point cloud.
- □ You can export the calibration model for later use.


EURESYS
Easy3D Studio
File
View
GenICam
Help
Save Project Ctrl+S
Save Project as... Ctrl+Shift+S
Export PointCloud... Ctrl+X
Import Calibration Model... Ctrl+I
Export Calibration Model... Ctrl+P
Exit Shift+F4
PointCloud
ZMap
Calibrate
47.0
9.4
18.8
28.2
37.6
47.0
X
Z
9.4

CalibrationMethod	
Method	ObjectBasedMethod
CalibrationModel	
Object Scale X	5
Object Scale Y	5
Object Scale Z	5
Object Type	TruncatedDoublePyramids
Passes count	1
Precision Vs Spe	Balanced

Object Scale Z
Software parameter

Required parameters

The calibration based on a calibration object requires several parameters:

- Set the Object Type as DoublePyramid or TruncatedDoublePyramid.
  - □ The DoublePyramid object type is deprecated and not recommended.
- Set the Object Size to represent the real size of the calibration object.
  - □ If your calibration object has a base of 20 mm by 30 mm and a height of 5 mm, set these values in the Object Size A/B/C parameters.
  - □ The point cloud after the calibration uses coordinates in millimeters.
- Set the parameter Precision Vs Speed Trade Off to define the time spent on the calibration process.
  - □ The 3 possible values are Fast, Balanced and Precise.

- Set the parameter Passes count to define the number of iterations used to refine the calibration model.
  - Use 1 for the fastest processing.
  - Use up to 3 for slower but potentially better calibration model.

Using the calibration with Open eVision

- The class `EObjectBasedCalibrationModel` is the container for the object based calibration model.
- The class `EObjectBasedCalibrationGenerator` performs the computation of such a model using an EDepthMap8/16/32f as input.

The following code snippet illustrates the calculation of a calibration model:

```
// Initialize a depth map from an image of a double truncated pyramid
EDepthMap16 depth_map;
depth_map.LoadImage("ctx1 calibration object.png");    // from Easy3D sample images
depth_map.SetZResolution(1.f / (1 << 5));               // 11.5 fixed point pixel format

// Initialize the calibration generator
EObjectBasedCalibrationGenerator calib_generator;
calib_generator.SetCalibrationObjectType(EObjectBasedCalibrationType_
TruncatedDoublePyramid);
calib_generator.SetCalibrationObjectSize(40.f, 60.f, 10.f); // Size of the calibration
object

// Compute the calibration model
EObjectBasedCalibrationModel calib_model;
calib_model = calib_generator.Compute(depth_map);
float error = calib_model.GetCalibrationError();

// Save the calibration model
calib_model.Save("calib.model");
```

The following code snippet illustrates the use of a saved calibration model:

```
// Load the calibration model
EObjectBasedCalibrationModel calib_model;
calib_model.Load("calib.model");

// Load a depth map (captured in the same context)
EDepthMap16 depth_map;
depth_map.LoadImage("ctx1 shapes.png");
depth_map.SetZResolution(1.f / (1 << 5));

// Initialize a converter, use the loaded model
EDepthMapToPointCloudConverter converter;
converter.SetCalibrationModel(calib_model);

// Convert the depth map to a metric point cloud and save it
EPointCloud point_cloud;
converter.Convert(depth_map, point_cloud);
point_cloud.SavePCD("point_cloud.pcd");
```

To experiment and learn about the **Easy3D** calibration, a C++ sample called `3DCalibration` is provided with the source code in the **Open eVision** distribution.

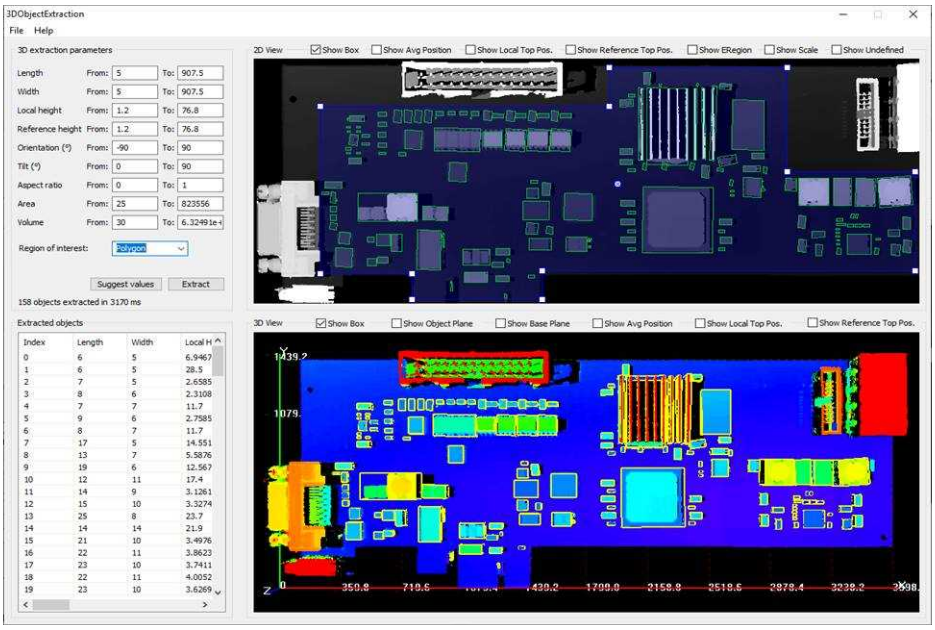
0.3. Easy3DObject - Extracting 3D Objects

Purpose and Workflow

Introduction

- The Easy3DObject tool extracts objects and their features from a ZMap.
  - □ The E3DObjectExtractor class uses a set of criteria to select the objects to extract.
  - □ The extracted objects are instances of the ED3Object class.
- Open eVision provides a demo application with C++ source code and 2 C++ / C# samples:

  This demo application exposes most of the features of the **Easy3DObject** tool.



3DObjectExtraction

File Help

3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

Suggest values Extract

158 objects extracted in 3170 ms

Extracted objects

Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	7	5	2.6585
3	8	6	2.3108
4	7	7	11.7
5	9	6	2.7585
6	8	7	11.7
7	17	5	14.551
8	13	7	5.5876
9	19	6	12.567
10	12	11	17.4
11	14	9	3.1261
12	15	10	3.3274
13	25	8	23.7
14	14	14	21.9
15	21	10	3.4976
16	22	11	3.8623
17	23	10	3.7411
18	22	11	4.0052
19	23	10	3.6269

2D View ☑ Show Box □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos. □ Show ERegion □ Show Scale □ Show Undefined

3D View ☑ Show Box □ Show Object Plane □ Show Base Plane □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos.

Library workflow

1. Load or build a ZMap (from an image or a point cloud).
2. Construct an E3DObjectExtractor instance.
3. Set the selection criteria of the E3DObjectExtractor instance.
4. Extract the 3D objects, with or without an ERegion.
5. Get and process the extracted objects list.

Load or build a ZMap

A ZMap is a grayscale image with a metric coordinate system. It is sometimes referred to as a “height map”.

You can create a ZMap from an 8- or a 16-bit image or generate it from a point cloud.

□ Before using an image as a ZMap, set the resolution.

✓ **TIP**
The resolution is the metric size of a pixel (for example in mm / pixel) and the height difference between 2 consecutive grayscale levels.

□ From a point cloud, use the `EPointCloudToZMapConverter` class to generate a ZMap. Choose the target ZMap resolution according to the point cloud sampling.

□ Depending on the 3D scan precision, you can use a ZMap with 8- or 16-bit per pixel.

✓ **TIP**
A 16-bit processing is more accurate but slower than an 8-bit processing.

Object Features

Units

Both the `E3DObjectExtractor` parameters and the `E3DObject` features are expressed in metric units.

□ For example: if the resolution of the input `EZMap` is expressed in mm / pixel, the length parameter is expressed in mm.

□ Use the `Resolution` accessors of the `EZMap` to query and change its resolution.

Angles are expressed in the unit defined by `Easy.AngleUnit`.

✓ **TIP**
In this documentation, we use the default setting and all angles are expressed in degrees.

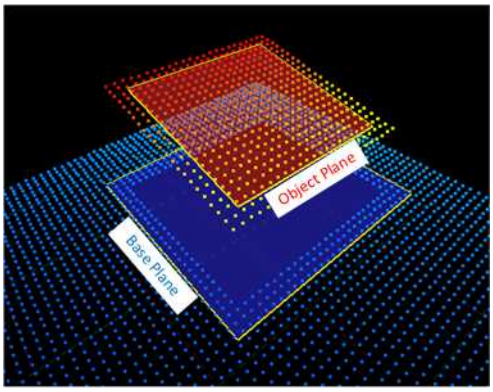
Object plane and base plane

The `E3DObjectExtractor` fits a plane to the pixels of each `E3DObject` output:

- Use `E3DObject.Plane` to access this plane.

The `E3DObjectExtractor` also tries to fit a plane to the pixels surrounding an `E3DObject`

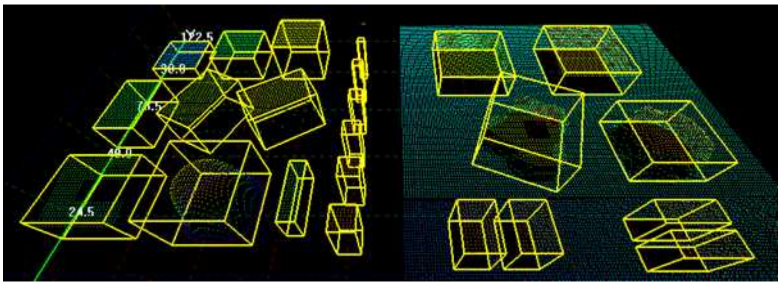
- This plane is called the *base plane*.
- It is an estimation of the local background around the object.
- If there are too many undefined pixels in this area, the base plane is equal to the reference plane of the input `EZMap`.



Bounding box

The *bounding box* is the minimal enclosing rectangle for all the object positions.

- It is oriented in the XY plane of the ZMap space (rotation around the Z axis of the ZMap).
- Its rotation is used as the orientation of the object (see `E3DObject.GetOrientation`).
- Its X and Y sizes are the object length and width (see `E3DObject.GetLength` and `E3DObject.GetWidth`).
- Its Z size is always in the Z axis of the ZMap direction.

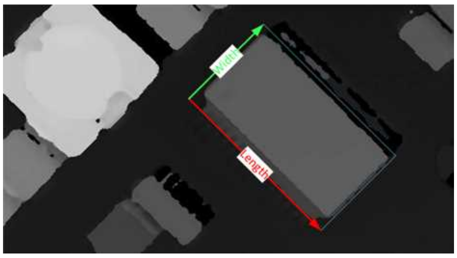


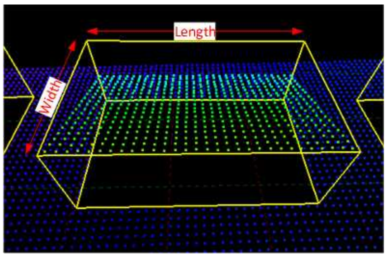
Length and width

The *length* of an object is the largest dimension on the XY plane in the ZMap space. It is the same as the size of the major axis of the bounding box.

The *width* of an object is the smallest dimension on the XY plane in the ZMap space. It is the same as the size of the minor axis of the bounding box.

Use the `E3DObjectExtractor.LengthRange` and the `E3DObjectExtractor.WidthRange` accessors to set the ranges of allowed values for the length and the width.





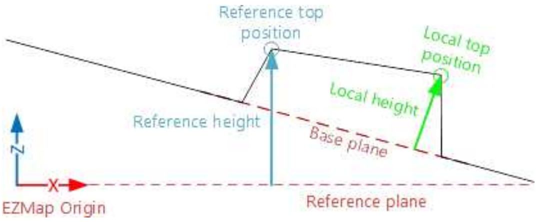
Local and reference top positions and heights

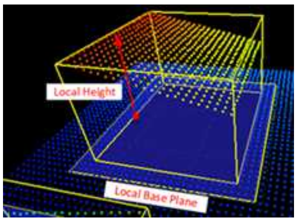
The *local top position* of an object is the position (3D coordinates) of the point in the `E3DObject` that is the furthest from the base plane.

The *local height* of an object is the distance between the local top position and the base plane.

The *reference top position* of an object is the position (3D coordinates) of the point in the `E3DObject` that is the furthest from the reference plane.

The *reference height* of an object is the distance between the reference top position and the reference plane.





If there are too many undefined pixels in the object surroundings:

- The base plane is equal to the reference plane of the input `EZMap`.
- The local top position is equal to the reference top position.
- The local height is equal to the reference height.

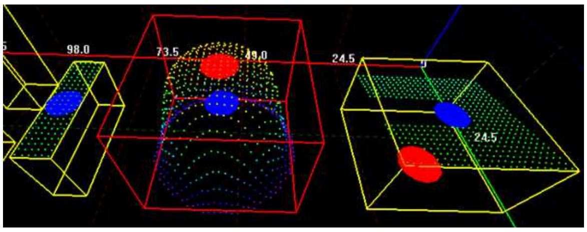
Use the `E3DObjectExtractor.LocalHeightRange` and the `E3DObjectExtractor.ReferenceHeightRange` accessors to set the ranges of allowed values for the local and the reference height.

Average position

The *average position* is the arithmetic mean of the 3D positions of the object, also known as the barycenter.

In the illustration below:

- The average position is displayed in blue.
- The top position is displayed in red.
- On the left object, the average and the top positions are at the same place.
- On the center object the average position is “inside” the object.

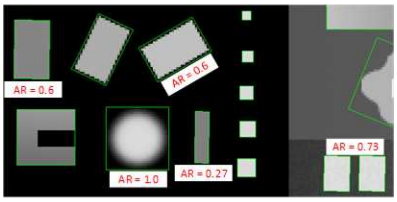


Aspect ratio

The *aspect ratio* is the width (the smallest dimension on the XY plane) divided by the length (the largest dimension).

- It lies between 0 and 1.
- The smaller the ratio, the more elongated the object is.
- A square has an aspect ratio of 1.

Use the `E3DObjectExtractor.AspectRatioRange` accessor to set the range of allowed values for the aspect ratio.

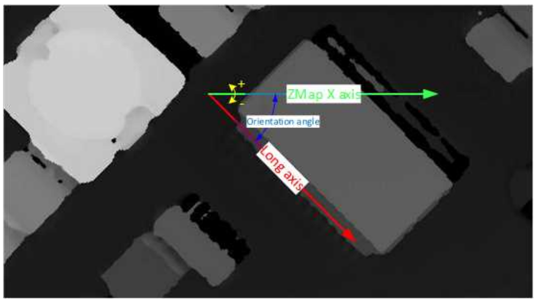


Orientation angle

The *orientation angle* is the angle between the X axis of the EZMap and the longest axis (the length) of the object.

- The angle is measured in the clockwise direction.
- The value must lie between -90° and +90°.

Use the E3DObjectExtractor.OrientationRange accessor to set the range of allowed values for the orientation angle.



Local and reference tilt angles

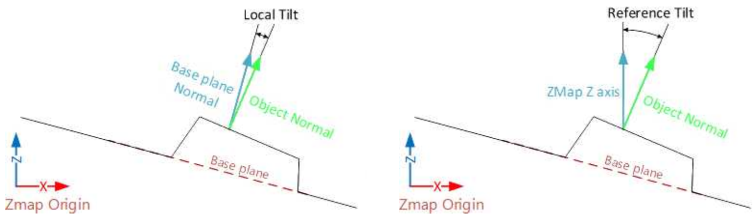
The *local tilt angle* is the angle between the base plane and the object plane.

- A value of 0 means that the object top surface is parallel to its base.
- The value must lie between 0° and +90°.

The *reference tilt angle* is the angle between the object plane and ZMap XY plane.

- A value of 0 means that the object top surface is parallel to its base.
- The value must lie between 0° and +90°.

Use the E3DObjectExtractor.LocalTiltRange and the E3DObjectExtractor.ReferenceTiltRange accessors to set the range of allowed values for the tilt angles.

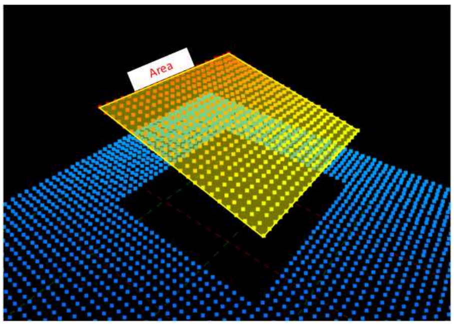


Area

The object *area* is the area of the top surface of the object projected on the reference plane of the EZMap.

- □ It is equal to [the number of pixels in the object] × [the x-resolution of the EZMap] × [the y-resolution of the EZMap].

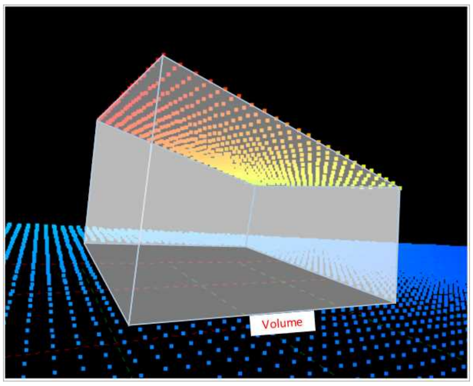
Use the E3DObjectExtractor.AreaRange accessor to set the range of allowed values for the area.



Volume

The object *volume* is the volume that lies between the top surface and the base plane of the object.

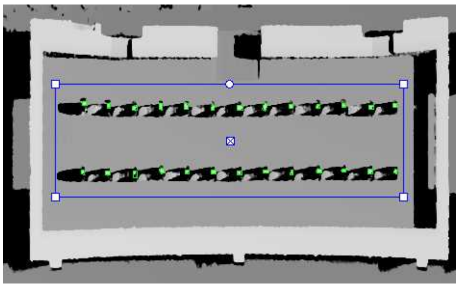
Use the E3DObjectExtractor.VolumeRange accessor to set the range of allowed values for the volume.



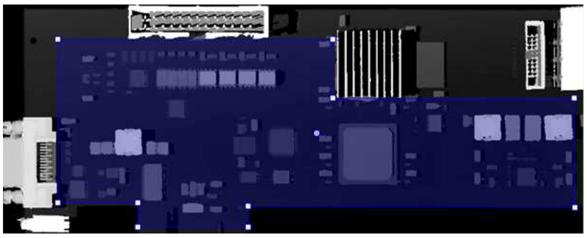
Extracting and Using Objects

Extracting the objects

Use the E3DObjectExtractor.Extract method to perform the objects extraction.



You can limit the extraction to an ERegion, for example to ignore parts of the ZMap that are not interesting and/or to speed up the extraction process.



The processing speed of the extraction depends directly on:

- The number of pixels in the ZMap or in the ERegion.
- The number of segmented objects.

TIP
Adjust the extraction ranges to reduce the number of objects and speed up the extraction process.

Using the objects

The E3DObjectExtractor.Extract method populates a list of E3DObject fulfilling your set criteria.

- Each E3DObject is a collection of descriptive features of the associated 3D points in the EZMap, such as its oriented bounding box, its local height and its volume.
- Call the associated E3DObject method to access a feature.
- The E3DObject list is sorted from the smallest area to the largest area.

The code snippet below provides an example for extracting features from the E3DObject list.

```
////////////////////////////////////////////////////
// This code snippet shows how to read a QR code  //
// and retrieve the decoded data.                 //
////////////////////////////////////////////////////

// get the extracted objects and loop over them
std::vector<Easy3D::E3DObject> objects = extractor.GetObjects();
int nObjects = objects.size();
for (int index = 0; index < nObjects; ++index)
{
  // inspect bounding box dimensions
  E3DPoint bbCenter = objects[index].GetBoundingBox().GetCenter();
  float bbHeight = objects[index].GetBoundingBox().GetXSize();
  float bbLength = objects[index].GetBoundingBox().GetYSize();

  // inspect object plane and base plane
  Easy3D::E3DPlane opjPlane = objects[index].GetPlane();
  Easy3D::E3DPlane basePlane = objects[index].GetBasePlane();

  // inspect the ERegion that exactly contains the object
  ERegion objRegion = objects[index].GetRegion();
}
```

Visualizing the objects

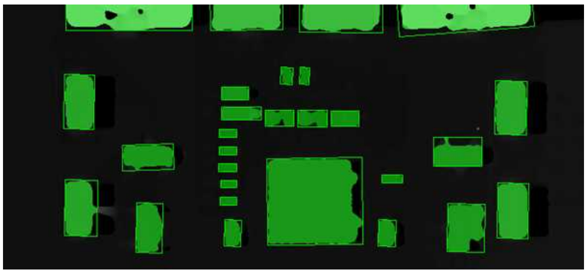
To visualize some of these features in 2D or 3D:

- Use the E3DObject.Draw method.
- Or submit a list of E3DObject to an E3DViewer.

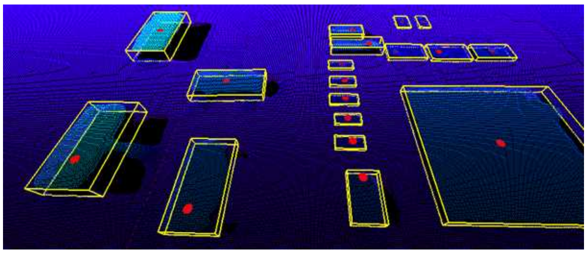
TIP
In an E3DViewer, use the ERenderStyle structure to choose your rendering style.

The following code snippets illustrate how to draw some object features:

- In a 2D graphic context: Drawing a 2D Feature from the List of E3DObjects



- In a 3D viewer: Drawing 3D Features from a List of E3DObjects

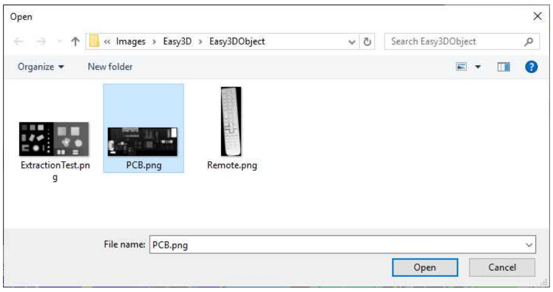


Use Case - Inspecting a PCB

The purpose of this use case is to test if all the components are present and correctly placed on the PCB.

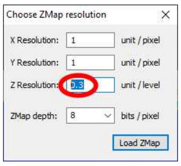
TIP
This example uses the sample image `Sample Images\Easy3D\Easy3DObject\PCB.png` and the illustrations are based on the **Easy3DObject** demo application.

1. Load the PCB image.



File name: PCB.png

2. Set the resolution.
   - The provided PCB sample is an 8-bit gray scale image.
   - Use a Z resolution of 0.3 metric unit per gray scale level for a realistic proportion.



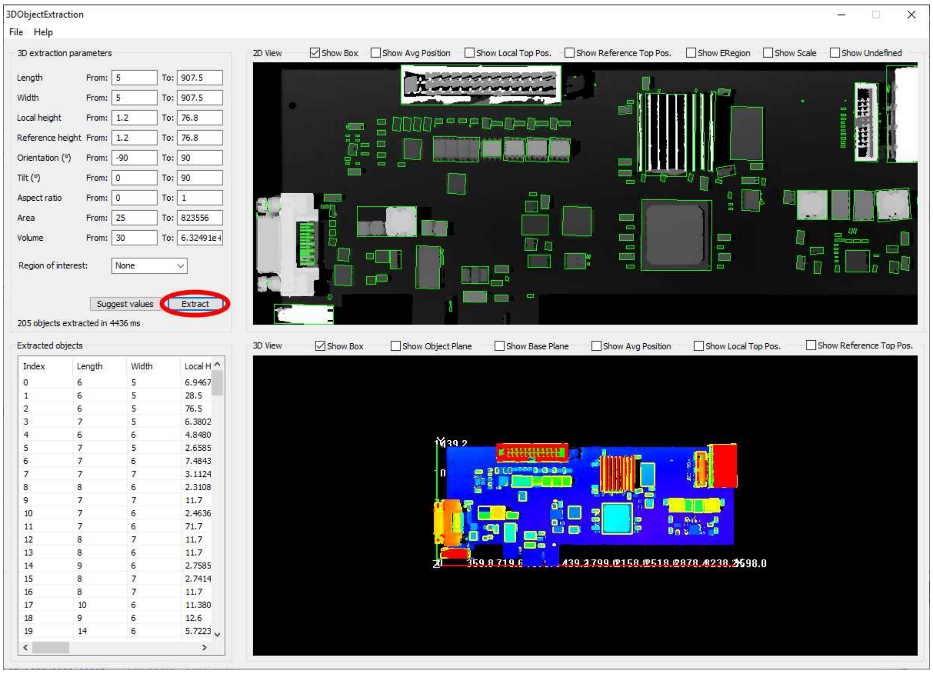
X Resolution: 1 unit / pixel
Y Resolution: 1 unit / pixel
Z Resolution: 0.3 unit / level
ZMap depth: 8 bits / pixel

3. Keep the suggested parameters for a first extraction.
   - The suggested parameters are set from the ZMap width, height and resolution.

4. Click on the Extract button to perform the extraction.

   When the extraction is done:

   - The object list is filled.
   - Click on a column title to sort the object list.
   - The various measures are displayed.
   - The 2D View and the 3D View show the extracted object bounding boxes.



3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: None

205 objects extracted in 4436 ms

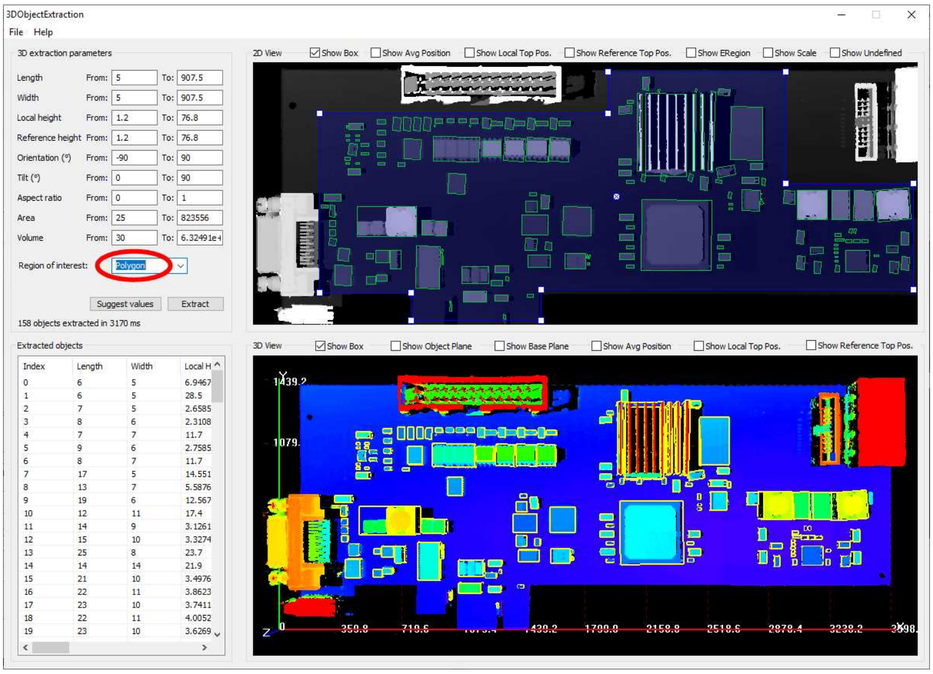
2D View ☑ Show Box □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos. □ Show ERegion □ Show Scale □ Show Undefined

Extracted objects

Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	6	5	76.5
3	7	5	6.3802
4	6	6	4.8480
5	7	5	2.6585
6	7	6	7.4843
7	7	7	3.1124
8	8	6	2.3108
9	7	7	11.7
10	7	6	2.4636
11	7	6	71.7
12	8	7	11.7
13	8	6	11.7
14	9	6	2.7585
15	8	7	2.7414
16	8	7	11.7
17	10	6	11.380
18	9	6	12.6
19	14	6	5.7223

3D View ☑ Show Box □ Show Object Plane □ Show Base Plane □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos.

5. Use a polygon region of interest to restrict the searched area.
   - You can limit the extraction to a region defined as a rectangle, a polygon or an ellipse in the demo application.
   - Use the Open eVision API, to define and use any `ERegion`.



3DObjectExtraction

File Help

3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

158 objects extracted in 3170 ms

Extracted objects

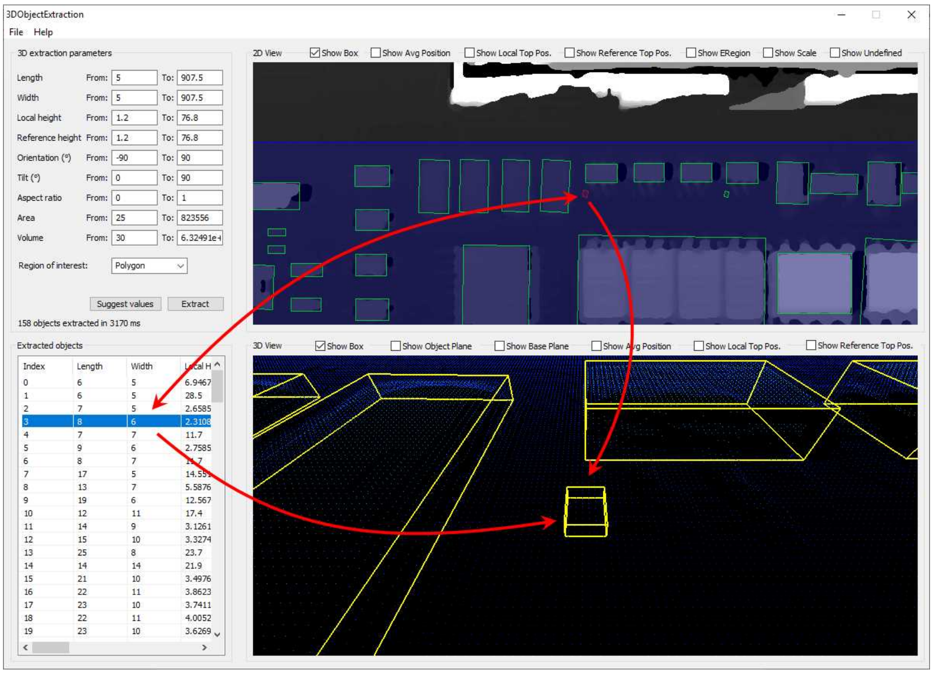
Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	7	5	2.6585
3	8	6	2.3108
4	7	7	11.7
5	9	6	2.7585
6	8	7	11.7
7	17	5	14.551
8	13	7	5.5876
9	19	6	12.567
10	12	11	17.4
11	14	9	3.1261
12	15	10	3.3274
13	25	8	23.7
14	14	14	21.9
15	21	10	3.4976
16	22	11	3.8623
17	23	10	3.7411
18	22	11	4.0052
19	23	10	3.6269

2D View ☑ Show Box □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos. □ Show ERegion □ Show Scale □ Show Undefined

3D View ☑ Show Box □ Show Object Plane □ Show Base Plane □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos.

6. Press again the Extract button to generate a new list of objects. Now, only the objects located inside the region are extracted.

7. The 2D View and 3D View automatically focus on the object selected in the list. You can also select an object by clicking on a bounding box in the 2D View.



3DObjectExtraction

File Help

3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

158 objects extracted in 3170 ms

Extracted objects

Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	7	5	2.6585
3	8	6	2.3108
4	7	7	11.7
5	9	6	2.7585
6	8	7	11.7
7	17	5	14.551
8	13	7	5.5876
9	19	6	12.567
10	12	11	17.4
11	14	9	3.1261
12	15	10	3.3274
13	25	8	23.7
14	14	14	21.9
15	21	10	3.4976
16	22	11	3.8623
17	23	10	3.7411
18	22	11	4.0052
19	23	10	3.6269

2D View ☑ Show Box □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos. □ Show ERegion □ Show Scale □ Show Undefined

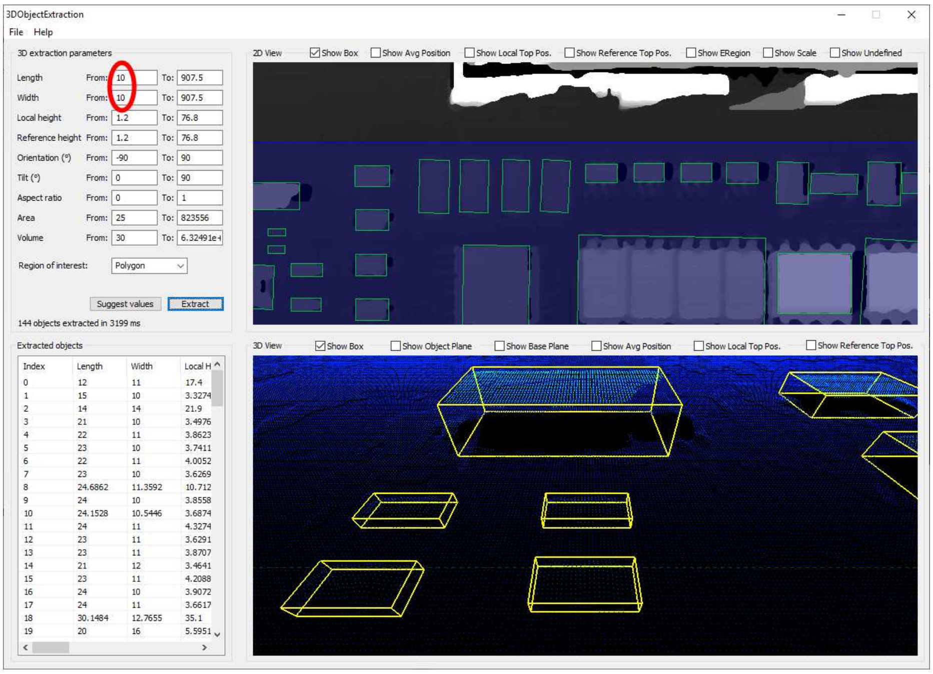
3D View ☑ Show Box □ Show Object Plane □ Show Base Plane □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos.

8. Use the size ranges to discard the smaller components.

To add or remove objects:

- Change the extraction parameters, like the length and width ranges.
- In the illustration below, objects smaller than 10x10 metric unit are not extracted.

NOTE
After changing a parameter, press the Extract button to perform a new extraction.



3D extraction parameters

Length	From:	10	To:	907.5
Width	From:	10	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

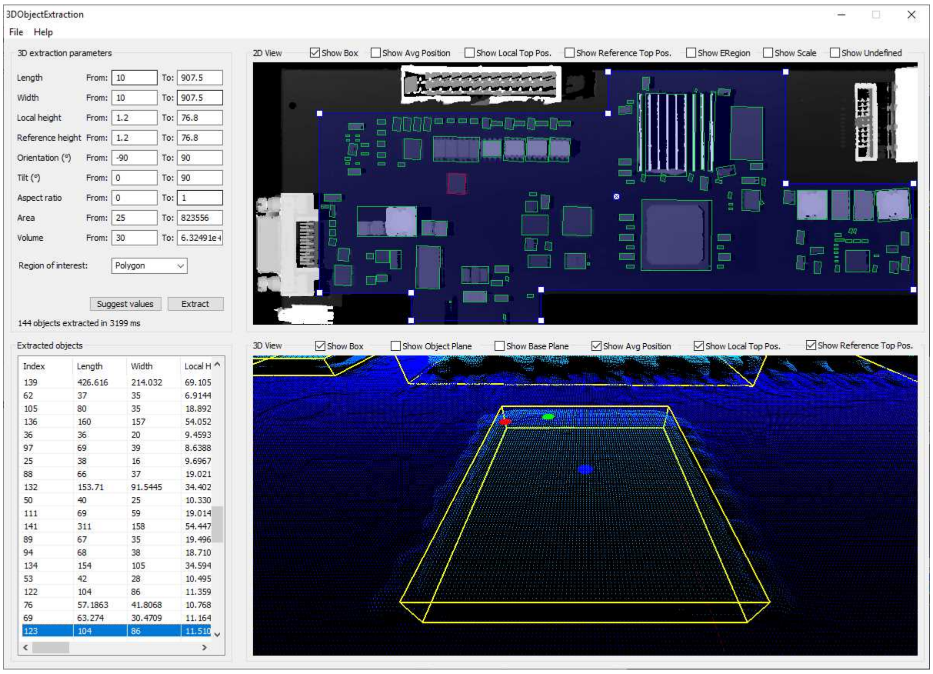
Region of interest: Polygon

144 objects extracted in 3199 ms

Extracted objects

Index	Length	Width	Local H
0	12	11	17.4
1	15	10	3.3274
2	14	14	21.9
3	21	10	3.4976
4	22	11	3.8623
5	23	10	3.7411
6	22	11	4.0052
7	23	10	3.6269
8	24.6862	11.3592	10.712
9	24	10	3.8558
10	24.1528	10.5446	3.6874
11	24	11	4.3274
12	23	11	3.6291
13	23	11	3.8707
14	21	12	3.4641
15	23	11	4.2088
16	24	10	3.9072
17	24	11	3.6617
18	30.1484	12.7655	35.1
19	20	16	5.5951

9. Check or uncheck the boxes at the top of the views to toggle the display of most of the object features, either in the 2D View or the 3D View.
   - In the illustration below, the object list is sorted by local height.
   - The first object is selected and displayed in both views.



3DObjectExtraction

File Help

3D extraction parameters

Length	From:	10	To:	907.5
Width	From:	10	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

144 objects extracted in 3199 ms

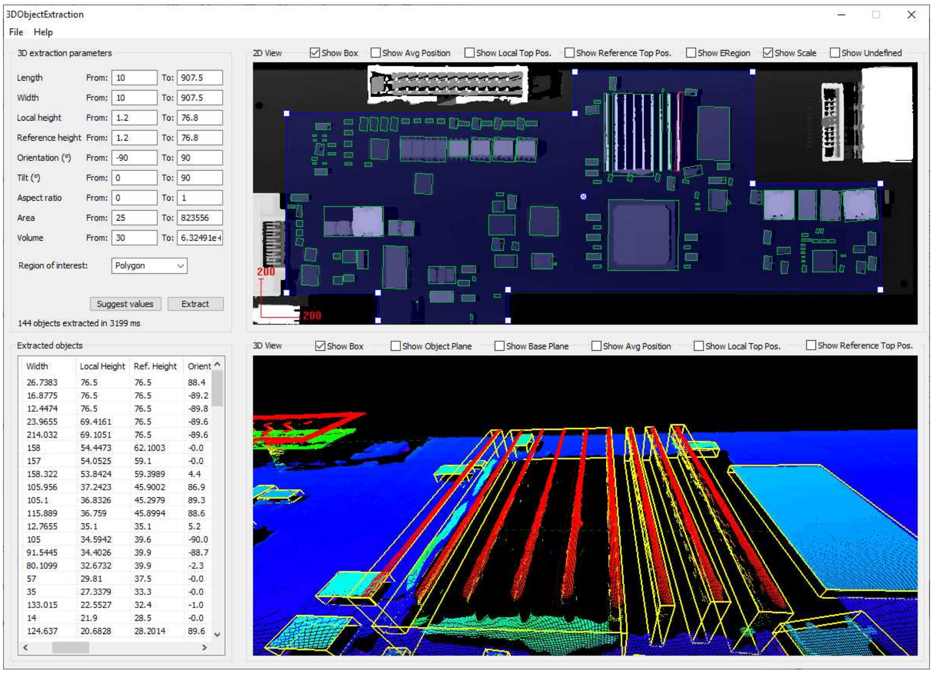
2D View ☑ Show Box □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos. □ Show ERegion □ Show Scale □ Show Undefined

3D View ☑ Show Box □ Show Object Plane □ Show Base Plane ☑ Show Avg Position ☑ Show Local Top Pos. ☑ Show Reference Top Pos.

Extracted objects

Index	Length	Width	Local H
139	426.616	214.032	69.105
62	37	35	6.9144
105	80	35	18.892
136	160	157	54.052
36	36	20	9.4593
97	69	39	8.6388
25	38	16	9.6967
88	66	37	19.021
132	153.71	91.5445	34.402
50	40	25	10.330
111	69	59	19.014
141	311	158	54.447
89	67	35	19.496
94	68	38	18.710
134	154	105	34.594
53	42	28	10.495
122	104	86	11.359
76	57.1863	41.8068	10.768
69	63.274	30.4709	11.164
123	104	86	11.510

10. Adjust the extraction parameters to accept or reject objects based on the results.



3DObjectExtraction

File Help

3D extraction parameters

Length	From:	10	To:	907.5
Width	From:	10	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

144 objects extracted in 3199 ms

2D View ☑ Show Box □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos. □ Show ERegion ☑ Show Scale □ Show Undefined

3D View ☑ Show Box □ Show Object Plane □ Show Base Plane □ Show Avg Position □ Show Local Top Pos. □ Show Reference Top Pos.

Extracted objects

Width	Local Height	Ref. Height	Orient
26.7383	76.5	76.5	88.4
16.8775	76.5	76.5	-89.2
12.4474	76.5	76.5	-89.8
23.9655	69.4161	76.5	-89.6
214.032	69.1051	76.5	-89.6
158	54.4473	62.1003	-0.0
157	54.0525	59.1	-0.0
158.322	53.8424	59.3989	4.4
105.956	37.2423	45.9002	86.9
105.1	36.8326	45.2979	89.3
115.889	36.759	45.8994	88.6
12.7655	35.1	35.1	5.2
105	34.5942	39.6	-90.0
91.5445	34.4026	39.9	-88.7
80.1099	32.6732	39.9	-2.3
57	29.81	37.5	-0.0
35	27.3379	33.3	-0.0
133.015	22.5527	32.4	-1.0
14	21.9	28.5	-0.0
124.637	20.6828	28.2014	89.6

11. Open the Help menu and click on Generate code snippet to generate the C++ code corresponding to the current configuration.

    The generated code illustrates how you can:

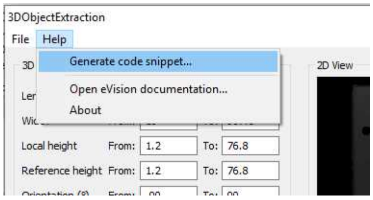
    □ Load a ZMap.

    □ Define a region.

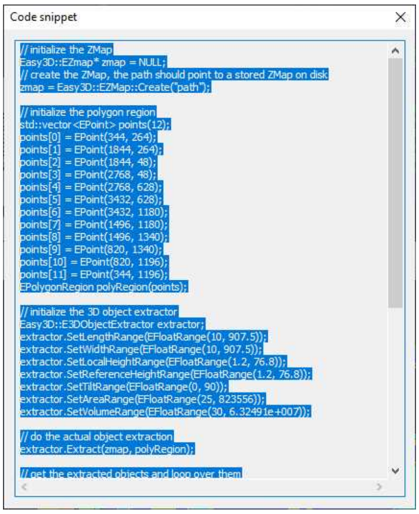
    □ Set the configuration parameters of the extraction.

    □ Start the extraction process.

    □ Iterate through the resulting objects list.



Local height From: 1.2 To: 76.8
Reference height From: 1.2 To: 76.8



```
// initialize the ZMap
Easy3D::EZmap* zmap = NULL;
// create the ZMap, the path should point to a stored ZMap on disk
zmap = Easy3D::EZMap::Create("path");

// initialize the polygon region
std::vector<EPoint> points(12);
points[0] = EPoint(344, 264);
points[1] = EPoint(1844, 264);
points[2] = EPoint(1844, 48);
points[3] = EPoint(2768, 48);
points[4] = EPoint(2768, 628);
points[5] = EPoint(3432, 628);
points[6] = EPoint(3432, 1180);
points[7] = EPoint(1496, 1180);
points[8] = EPoint(1496, 1340);
points[9] = EPoint(820, 1340);
points[10] = EPoint(820, 1196);
points[11] = EPoint(344, 1196);
EPolygonRegion polyRegion(points);

// initialize the 3D object extractor
Easy3D::E3DObjectExtractor extractor;
extractor.SetLengthRange(EFloatRange(10, 907.5));
extractor.SetWidthRange(EFloatRange(10, 907.5));
extractor.SetLocalHeightRange(EFloatRange(1.2, 76.8));
extractor.SetReferenceHeightRange(EFloatRange(1.2, 76.8));
extractor.SetTiltRange(EFloatRange(0, 90));
extractor.SetAreaRange(EFloatRange(25, 823556));
extractor.SetVolumeRange(EFloatRange(30, 6.32491e+007));

// do the actual object extraction
extractor.Extract(zmap, polyRegion);

// get the extracted objects and loop over them
```