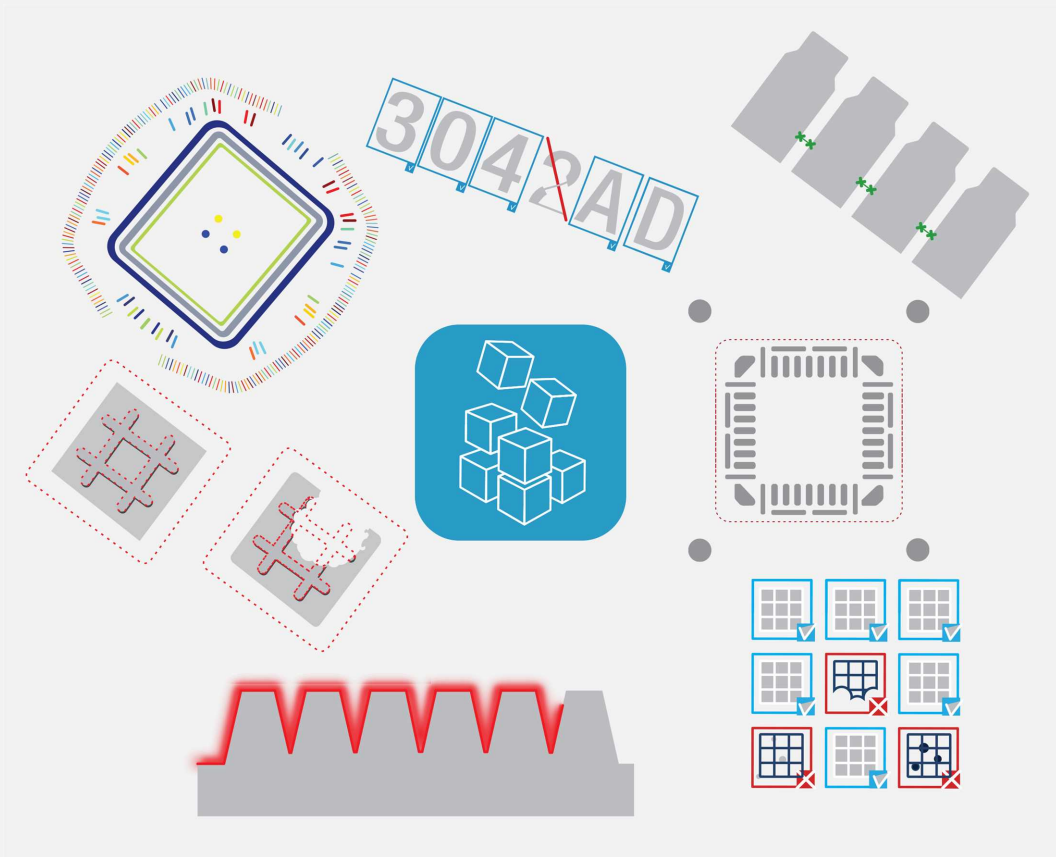


Open eVision

Examples in C++



This documentation is provided with Open eVision 2.16.1 (doc build 1156).
www.euresys.com

Contents

PART I : STARTING UP	7
1. Installing Open eVision	8
2. Selecting the Licensing Model	11
3. Manipulating Pixels Containers and Files	12
3.1. Pixel Container File Save	12
3.2. Pixel Container File Load	14
3.3. Memory Allocation	15
3.4. Image and Depth Map Buffer	16
3.5. Image Coordinate Systems	19
3.6. Image Drawing and Overlay	20
3.7. 3D Rendering of 2D Images	20
3.8. Vector Types and Main Properties	22
3.9. ROI Main Properties	26
3.10. Arbitrarily Shaped ROI (ERegion)	28
3.11. Flexible Masks	35
3.12. Profile	39
PART II : GENERAL PURPOSE LIBRARIES	41
1. EasyImage - Pre-Processing Images	42
1.1. Intensity Transformation	42
1.2. Thresholding	45
1.3. Arithmetic and Logic	46
1.4. Non-Linear Filtering	49
1.5. Geometric Transforms	54
1.6. Noise Reduction and Estimation	56
1.7. Scalar Gradient	59
1.8. Vector Operations	59
1.9. Canny Edge Detector	61
1.10. Harris Corner Detector	62
1.11. Overlay	63
1.12. Operations on Interlaced Video Frames	64
1.13. Flexible Masks in EasyImage	64
1.14. Computing Image Statistics	65
2. EasyColor - Pre-Processing Color Images	70
2.1. Bayer Conversion	73
2.2. LUT for Gain/Offset (Color)	76
2.3. LUT for Color Calibration	77
2.4. LUT for Color Balance	77
PART III : MATCHING AND MEASUREMENT TOOLS	80
1. EasyObject - Analyzing Blobs	81
1.1. Image Segmenters	84
1.2. Image Encoder	88
1.3. Holes Construction	90
1.4. Normal vs. Continuous Mode	92
1.5. Selecting and Sorting Blobs	94
1.6. Object Template Matcher	96
1.7. Advanced Features	98

Computable Features	98
Draw Coded Elements	103
Flexible Masks in EasyObject	104
2. EasyGauge - Measuring down to Sub-Pixel	107
2.1. Workflow	107
2.2. Gauge Definitions	108
2.3. Find Transition Points Using Peak Analysis	112
2.4. Find Shapes Using Geometric Models	117
2.5. Gauge Manipulation: Draw, Drag, Plot, Group	119
2.6. Calibration and Transformation	120
2.7. Calibration Using EWorldShape	121
2.8. Advanced Features	124
2.9. Unwarp an Image	126
3. EasyFind - Matching Geometric Patterns	129
3.1. Workflow	129
3.2. Learning Process	132
3.3. Finding Process	133
3.4. Advanced Features	134
4. EasyMatch - Matching Area Patterns	136
4.1. Workflow	136
4.2. Learning Process	137
4.3. Matching Process	139
4.4. Advanced Features	139
5. EChecker2 - Validating Golden Templates	141
5.1. EChecker2	141
5.2. Creating a Model	141
5.3. Inspecting an Image	144
PART IV : TEXT AND CODE READING TOOLS	146
1. EasyBarCode - Reading Bar Codes	147
1.1. Reading Bar Codes	147
1.2. Reading Mail Bar Codes	151
2. EasyMatrixCode - Reading Matrix Codes	155
2.1. EasyMatrixCode vs EasyMatrixCode2	155
2.2. EasyMatrixCode	155
Specifications	155
Workflow	156
Reading a Matrix Code	156
Learning a Matrix Code	157
Computing the Print Quality	158
Using GS1 Data Matrix Codes	158
2.3. EasyMatrixCode2	159
Specifications	159
Workflow	161
Reading a Matrix Code	161
Learning a Matrix Code	162
Computing the Print Quality	162
Using GS1 Data Matrix Codes	163
Asynchronous Processing	163
Advanced Parameters	164
3. EasyQRCode - Reading QR Codes	165
3.1. Workflow	165
3.2. QR Codes Specifications	166

3.3. Reading QR Codes	170
4. EasyOCR - Reading Texts	174
4.1. Workflow	174
4.2. Learning Process	175
4.3. Segmenting	176
4.4. Recognition	177
5. EasyOCR2 - Reading Texts (Improved)	180
5.1. Workflow	180
5.2. Detection	182
5.3. Learning	186
5.4. Recognition	186
PART V : DEEP LEARNING INSPECTION TOOLS	188
1. Deep Learning Tools - Inspecting Images with Deep Learning	189
1.1. Purpose and Workflow	189
1.2. Tools and Resources	192
1.3. Hardware Support (CPU/GPU)	195
1.4. Managing the Images	197
Images and Labels	197
Adding Images	199
Editing the Segmentation of an Image	200
Editing the Objects of an Image	202
ROI and Mask	204
1.5. Managing the Dataset	208
Training and Validation Datasets	208
Using Data Augmentation	210
1.6. Training a Deep Learning Tool	213
2. EasyClassify - Classifying Images	214
2.1. Tool and Images	214
2.2. Validating the Results	216
2.3. Classifying New Images	218
2.4. Benchmarks for EasyClassify	220
3. EasySegment - Detecting and Segmenting Defects	222
3.1. Unsupervised vs Supervised Modes	222
3.2. EasySegment Unsupervised	223
Tool and Configuration	223
Validating the Results	225
Applying the Tool to New Images	226
Benchmarks for EasySegment Unsupervised	227
3.3. EasySegment Supervised	229
Tool and Configuration	229
Using the Supervised Segmenter	231
Evaluating the Results	233
Benchmarks for EasySegment Supervised	238
4. EasyLocate - Locating Objects and Defects	240
4.1. Tool and Configuration	240
4.2. Locating Objects	244
4.3. Validating the Results	246
4.4. Benchmarks for EasyLocate	249
PART VI : 3D PROCESSING TOOLS	251
1. Easy3D - Using 3D Toolset	252

1.1. Basic Concepts	252
1.2. Static Methods	256
1.3. Point Cloud	259
Mapping Attributes	259
Coordinates Transformations	259
Reducing a Point Cloud	260
Managing Planes	261
Aligning	264
1.4. Mesh	267
1.5. ZMap	268
Generating a ZMap	268
Creating a Point Cloud from a ZMap	270
Managing the Coordinates	271
1.6. 3D Viewer	272
1.7. Photometric Stereo	279
Photometric Stereo and Process	279
Calibration	280
Computation and Results	282
Processing the Results with Open eVision Tools	285
Improving the Results	287
2. Easy3DLaserLine - Laser Line Extraction and Calibration	290
2.1. Laser Triangulation	290
2.2. The Laser Line 3D Acquisition Pipeline	292
2.3. Laser Line Extraction	293
2.4. Calibration	297
2.5. Object-Based Calibration Guidelines	299
3. Easy3DObject - Extracting 3D Objects	308
3.1. Purpose and Workflow	308
3.2. Object Features	309
3.3. Extracting and Using Objects	315
3.4. Use Case - Inspecting a PCB	319
4. Easy3DMatch - 3D Alignment and Comparison	325
4.1. Purpose and Workflow	325
4.2. Alignment (E3DAligner)	327
4.3. Comparison (E3DComparer)	331
4.4. Alignment and Comparison (E3DMatcher)	335
PART VII : ADVANCED PROGRAMMING	342
1. Multicore Processing	343
2. EGrabberBridge - Using Images from Coaxlink	347
3. Handling the Memory in .NET	349
4. Using Open eVision in a DLL	354

PART I
STARTING UP

1. Installing Open eVision

Installer package on Windows

Open eVision comes as a single installer package `open_evision-win-X.Y.Z.B.exe`. 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 are installed on the system.

Older versions

Open eVision does not replace other Open eVision versions but installs alongside. Only maintenance releases (with the same major and minor version but a different revision) automatically update an existing installation.

Command-line interface

To install Open eVision with the command line, use:

- `open_evision-win-X.Y.Z.B.exe -silent 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 "open_evision-win-X.Y.Z.B.exe -silent INSTALLTYPE=[install_type]"`

Installation logs

- By default, the Open eVision installer generates installation log files in the `%LOCALAPPDATA%/Temp` directory.
- If you want the installer to generate the installation logs somewhere else, use the following command:
 - `open_evision-win-X.Y.Z.B.exe -log [logFilename]`
 - Where `[logFilename]` indicates where the main installation log is saved.
 - Additional log files may be created along the main one, their name is derived from `[logFilename]`.

Installer package on Linux

Open eVision comes as a single tarball `open_evision-linux-x86_64-X.Y.Z.B.deb.tar.gz`. It contains a Debian package that contains everything needed to run or develop your applications using Open eVision.

Packages

- The Open eVision tarball provides the following packages:
 - `open_evision-linux-x86_64.X.Y.Z.B.deb`: the library, the sample programs and sample images.
 - `neo-linux-license-manager-x86_64.X.Y.Z.B`: the Neo License Manager.
 - `codemeter-lite_X.Y.Z.B_amd64.deb`: a license system dependency.
- To install Open eVision:
 1. Open a terminal in the tarball directory.
 2. Decompress it:

```
$ sudo tar -xvf open_evision-linux-x86_64-X.Y.Z.B.deb.tar.gz
```
 3. Install the packages:

```
$ sudo apt install ./open_evision-linux-x86_64.X.Y.Z.B.deb ./ neo-linux-license-manager-x86_64.X.Y.Z.B ./codemeter-lite_X.Y.Z.B_amd64.deb
```
 4. We recommend you to copy/paste the `Sample Programs` folder to your home directory from:

```
/opt/euresys/Open_eVision_X_Y/
```

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.

Open eVision licenses are activated using the included license managers.

- Since release 2.13:
 - A new licensing system, named Neo Licensing System, is available with Open eVision.
 - The Neo License Manager manages these new licenses.
 - You can launch it at the end of the installation, or from the Windows start menu.
- Previous licensing systems:
 - The licensing systems used before 2.13 are still available.
 - Open eVision License Manager, the corresponding license manager, is still available too.
 - You can launch it at the end of the installation, or from the Windows start menu.



NOTE

On Windows XP, the Open eVision 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.

The header of Open eVision has a lot of content, and it is highly recommended:

- To place the content into a precompiled header to avoid unnecessarily recompilations.
- When using Visual Studio, to use the following settings to avoid compilation issues:
 - `/bigobj`: allows the compiler to handle more code in the headers.
 - `/zm256` (or bigger): allows bigger precompiled headers.
- To disable **Browse Information**, as it is superseded by Intellisense and it is known to crash when handling big projects.

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.

2. Selecting the Licensing Model

Starting with Open eVision 2.13, you can select the Open eVision licensing model(s) that you use with your application.



TIP

This avoids the delays that are sometimes added by the initialization of licensing models you do not use.

The `SelectLicensingModel` function

- To select the licensing models to enable with Open eVision, call the following function:
`Preconfiguration::SelectLicensingModels(ELicensingModel model)`
- Always call it before any other Open eVision function as calling it after the first call to Open eVision has no effect.
- The `SelectLicensingModels` function takes a single argument:
 - The licensing models to enable, as described by the `ELicensingModel` enumeration.

The `ELicensingModel` enumeration

- The `ELicensingModel` enumeration contains the different licensing models you can enable within Open eVision:
 - `Neo`: the new licensing model introduced with Open eVision 2.13.
 - `LegacyDongle`: the licensing model associated to the dongles used before 2.13.
 - `LegacySoftware`: the licensing model associated to the software-based licenses used before 2.13
 - `Legacy`: this licensing model mirrors the licensing as it operated in Open eVision before 2.13 (`LegacyDongle` + `LegacySoftware`).
 - `All` (default): this licensing model enables all the others (`Neo` + `LegacyDongle` + `LegacySoftware`).
- To use another combination of licensing models, use the `|` (`OR`) operator as follows:
`Preconfiguration::SelectLicensingModels(ELicensingModel_LegacyDongle | ELicensingModel_Neo)`

3. Manipulating Pixels Containers and Files

3.1. Pixel Container File Save

Images and depth maps

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

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

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

`Save` throws an exception when:

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



TIP

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

Image file type arguments

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

(*) Default value.

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

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

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

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

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

JPEG compression values

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

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

Representative JPEG 2000 compression quality values

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

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

Saving point clouds

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

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



TIP

The PCD format is supported in ASCII and binary modes.

3.2. Pixel Container File Load

Loading images and depth maps

- Use the `Load` method to load image data into an image object:
 - It has one argument: the **path**: path, filename, and file name extension.
 - File type is determined by the file format.
 - The destination image is automatically resized according to the size of the image on disk.
- The `Load` method throws an exception when:
 - File type identification fails
 - File type is incompatible with pixel type of the image object



TIP

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



TIP

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

Loading point clouds

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

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



TIP

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

3.3. Memory Allocation

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

Internal memory allocation

The image object dynamically allocates and deallocates a buffer.

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

To declare an image with internal memory allocation:

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

External memory allocation

The user controls `buffer` allocation or `links a third-party image` in the memory buffer to an Open eVision image.

- Image size and buffer address must be specified.
- When an image object is destroyed, the buffer is unaffected.

To declare an image with external memory allocation:

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



NOTE

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

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

3.4. Image and Depth Map Buffer

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

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

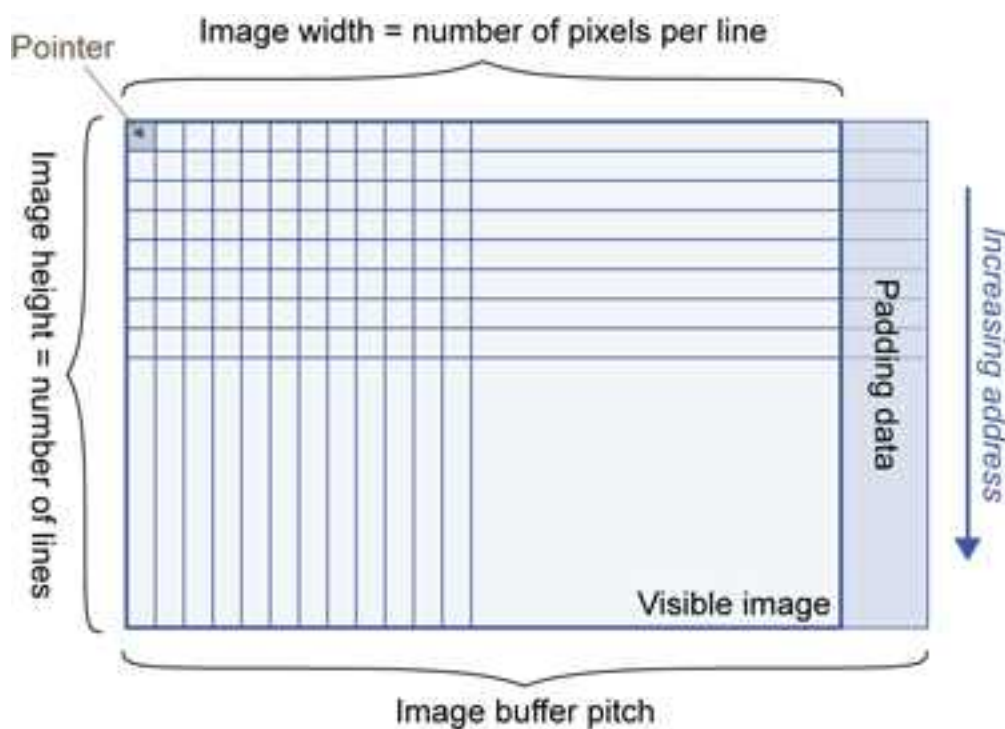


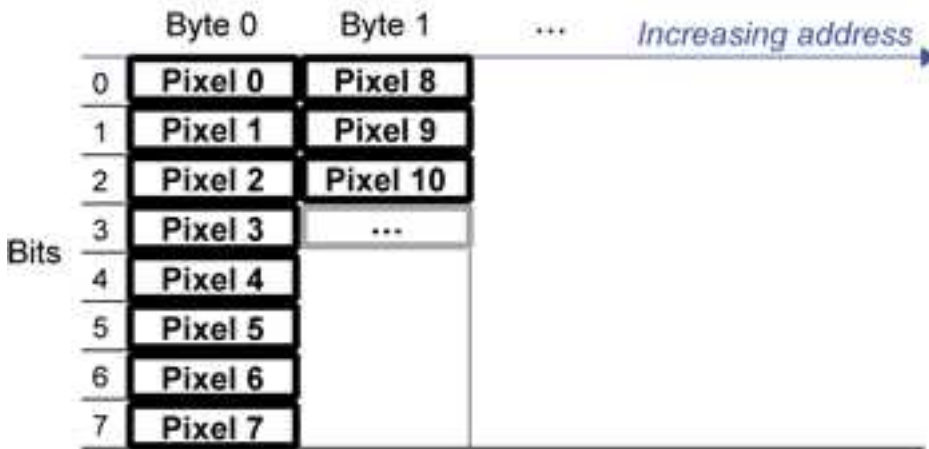
Image buffer pitch

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

Memory layout

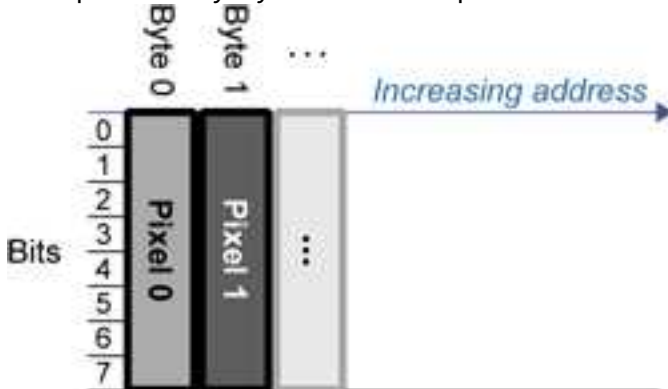
- **EImageBW₈** stores 8 pixels in one byte.

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



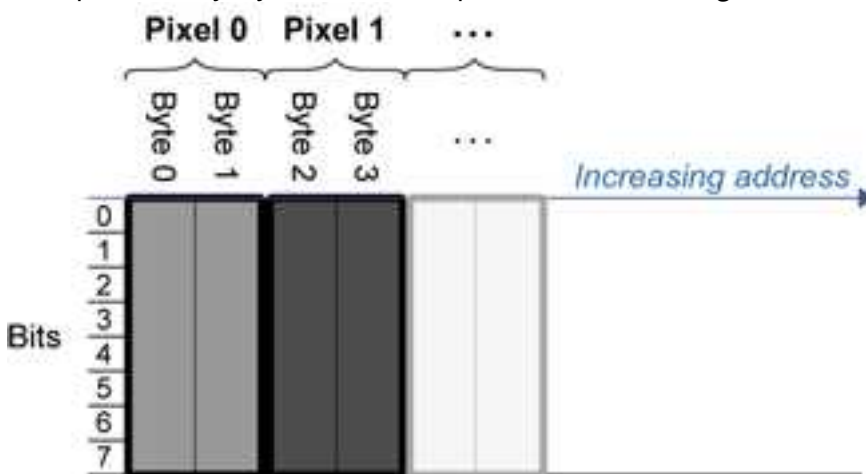
- **EImageBW₈** and **EDepthMap₈** store each pixel in one byte.

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



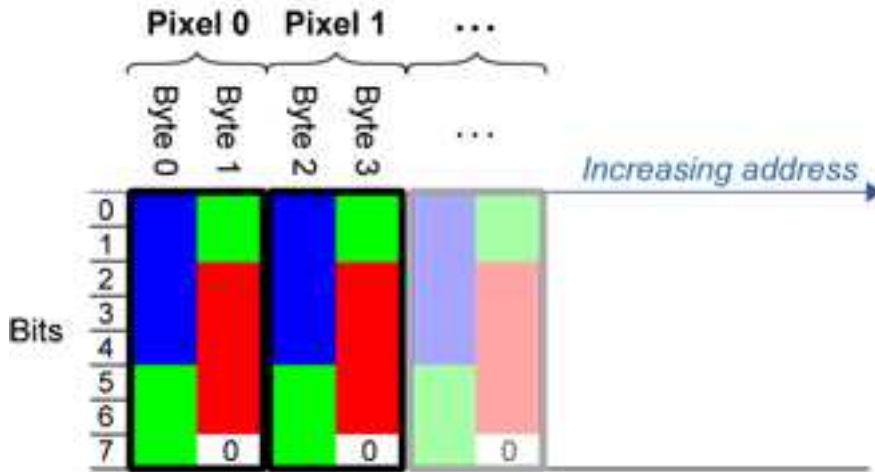
- **EImageBW₁₆** stores each pixel in a 16-bit word (two bytes).

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



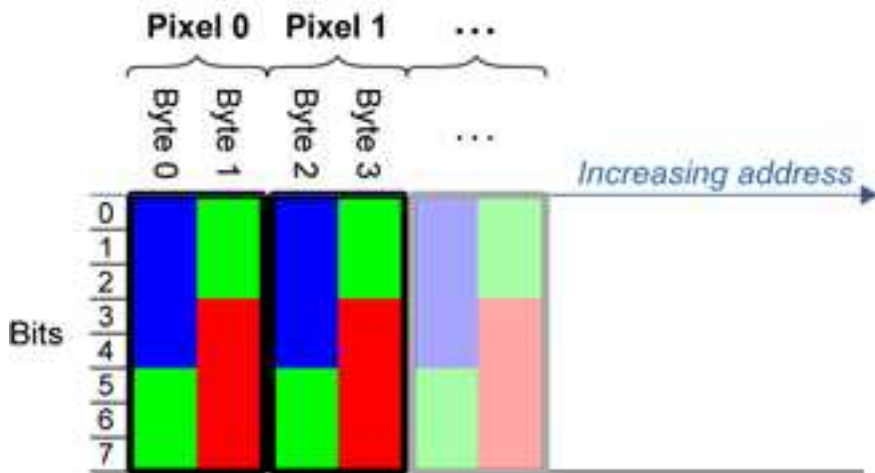
- **EImageC₅** stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

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



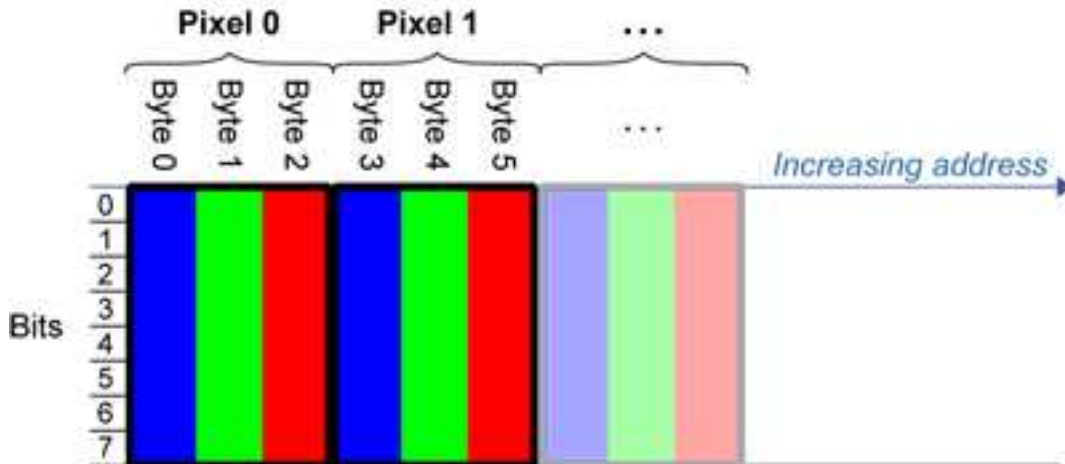
- [EImageC16](#) stores each pixel in 2 bytes. The first and third color components are coded with 5-bits. The second color component is coded with 6-bits.

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



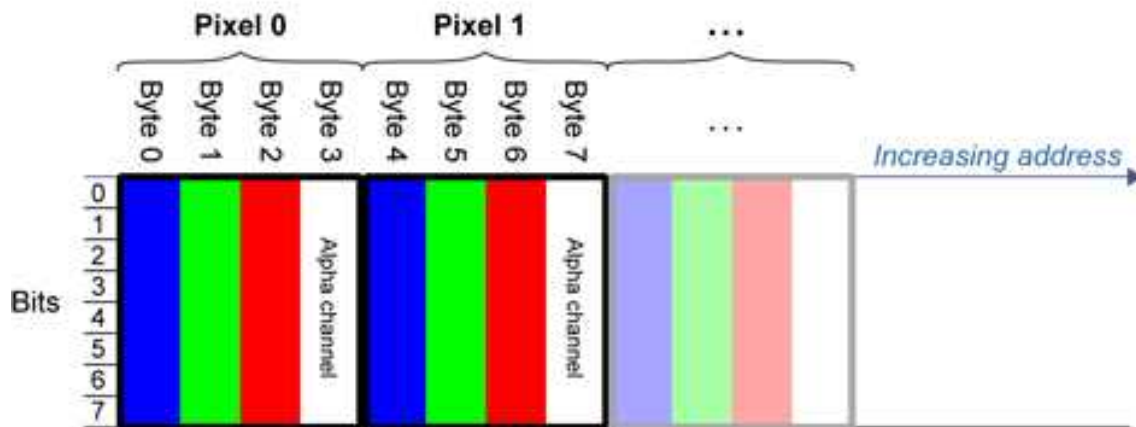
- [EDepthMap16](#) store each pixel in 2 bytes using a fixed point format.
- [EImageC24](#) stores each pixel in 3 bytes. Each color component is coded with 8-bits.

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.

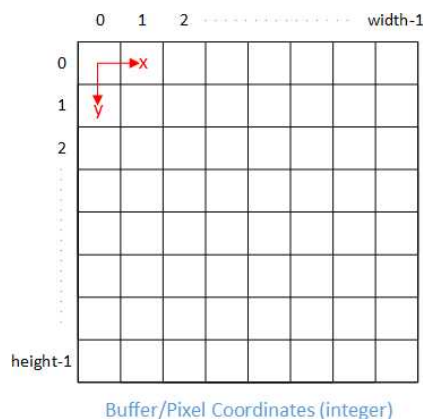
3.5. Image Coordinate Systems

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

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

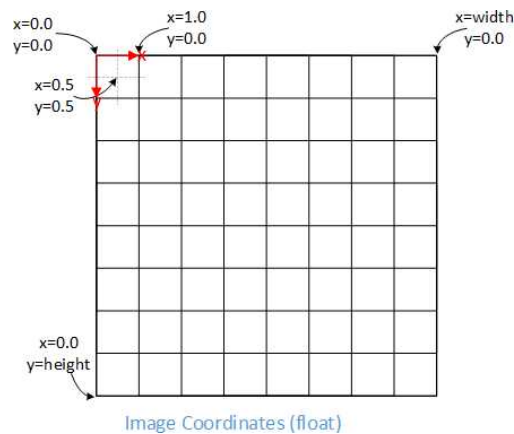
Integer coordinates

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



Real coordinates

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



3.6. Image Drawing and Overlay

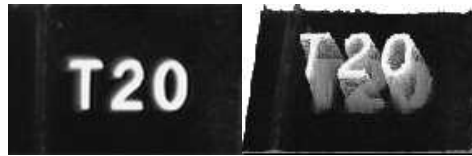
- Drawing uses Windows GDI (Graphics Device Interface) system calls.
 - MFC (Microsoft Foundation Class) applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
 - Borland/CodeGear OWL or VCL use a `Paint` event handler.
- The color palette in 256-color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

3.7. 3D Rendering of 2D Images

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

Gray 3D rendering

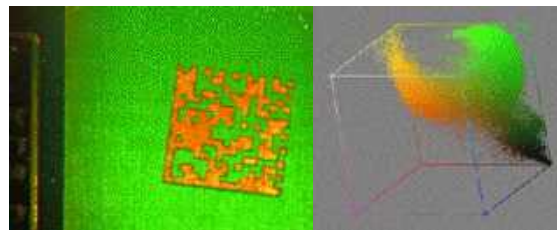
Easy: `Render3D` prepares a 3-dimensional rendering where gray-level values are altitudes. Magnification factors in the three directions (X = width, Y = height and Z = depth) can be given. The rendered image appears as independent dots whose size can be adjusted to make the surface more or less opaque.



3D rendering

Color histogram 3D rendering

Easy: `RenderColorHistogram` prepares a 3-dimensional rendering of a color image histogram. The pixels are drawn in the RGB space (not XY -plane) to show clustering and dispersion of RGB values. This function can process pixels in other color systems (using `EasyColor` to convert), but the raw RGB image is required to display the pixels in their usual colors. Magnification factors in all three directions (X = red, Y = green and Z = blue) can be given.

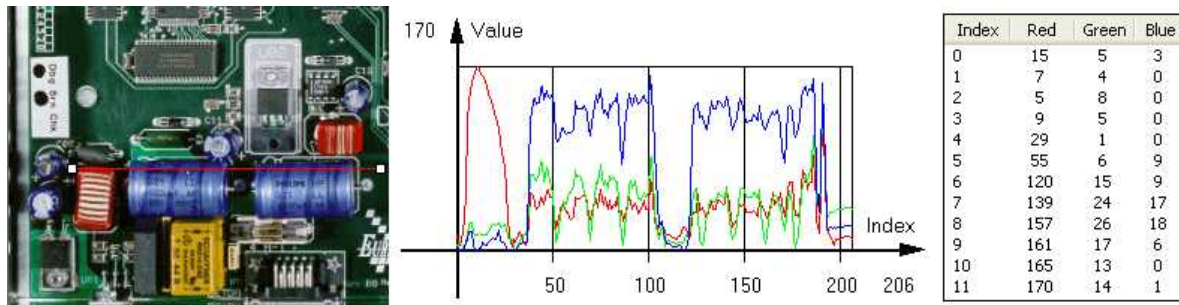


Color histogram rendering

3.8. Vector Types and Main Properties

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

[EVector](#) is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.



Profile in a C24 image, RGB values plot along profile and RGB values array ([EC24Vector](#))

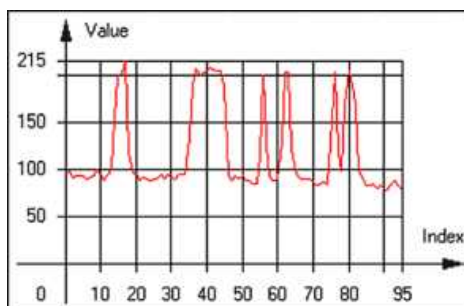
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

- To create a vector of the appropriate type:
 - Use its constructor and preallocate elements if required.
- To fill a vector with values:
 - Call the [EVector::Empty](#) member to empty it.
 - Call the [EC24Vector::AddElement](#) member to add elements one by one.
 - Use the indexing to access any element.
- To access a vector element, either for reading or writing:
 - Use the brackets operator [EC24Vector::operator\[\]](#).
- To determine the current number of elements:
 - Use the [EVector::NumElements](#) member.
- To draw the vector:
 - A pixel vector is a plot of the element values as a function of the element index, so its graphical appearance depends on its type. You can draw a vector in a window. For legibility, the drawing should appear on a neutral background.
 - Drawing is done in the device context associated to the desired window. By default, curves are drawn in blue and annotations in black. You can define: `graphicContext`, `width`, `height`, `originX`, `originY`, `color0`, `color1` and `color2`.
 - The [EC24Vector](#) has three curves drawn instead of one, each corresponding to a color component. By default the red, blue and green pens are used.

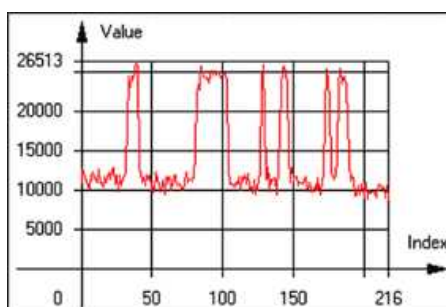
Vector types

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



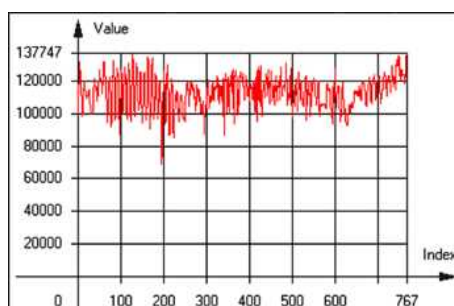
Graphical representation of an **EBW8Vector** (see [Draw method](#))

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



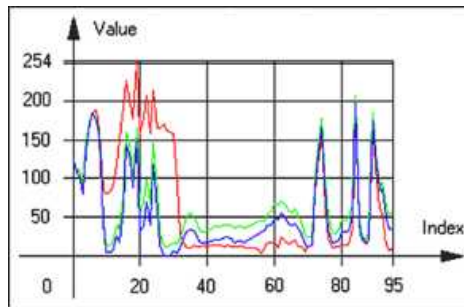
Graphical representation of an **EBW16Vector**

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



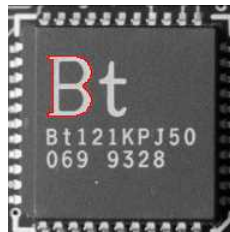
Graphical representation of an **EBW32Vector**

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



Graphical representation of an **EC24Vector**

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



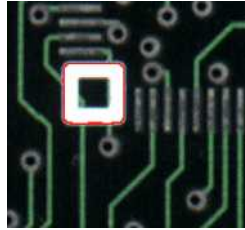
Graphical representation of an **EBW8PathVector** (see `Draw` method)

- **EBW6PathVector**: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).



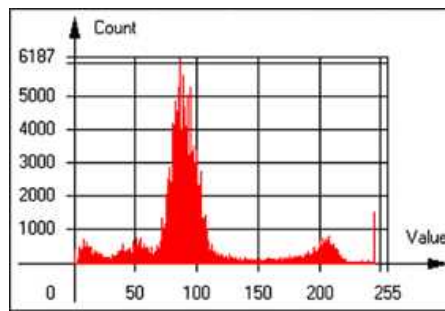
Graphical representation of an **EBW6PathVector** (see `Draw` method)

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



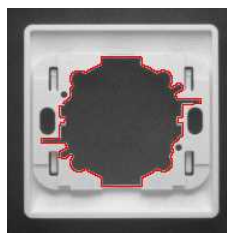
Graphical representation of an **EC24PathVector** (see [Draw method](#))

- **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`).

3.9. ROI Main Properties

ROIs are defined by a [width](#), a [height](#), and **origin x and y coordinates**.

The origins are specified with respect to the top left corner in the parent image or ROI.

The ROI must be wholly contained in its parent image.

The processing/analysis time of a BW1 ROI is faster if [OrgX](#) and [Width](#) are multiples of 8.

Save and load

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

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

ROI Classes

An Open eVision ROI inherits parameters from the abstract class [EBaseROI](#).

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

- [EROIBW](#)
- [EROIBW8](#)
- [EROIBW-6](#)
- [EROIBW32](#)
- [EROIC](#)
- [EROIC-5](#)
- [EROIC-6](#)
- [EROIC24](#)
- [EROIC24A](#)

Attachment

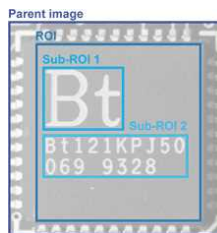
An ROI must be [attached](#) to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.

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

Nesting

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

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



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

Cropping

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



NOTE

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

Resizing and moving

ROIs can easily be resized and positioned by two functions and dragging handles:

- `EBaseROI::Drag` adjusts the ROI coordinates while the cursor moves.
- `EBaseROI::HitTest` informs if the cursor is placed over a dragging handle.
 - Once the handle is known, the cursor shape can be changed by an `OnSetCursor` MFC event handler. `HitTest` is unpredictable if called while dragging is in progress.
 - `HitTest` can be used in an `OnSetCursor` MFC event handler to change the cursor shape, or before a dragging operation like `OnLButtonDown`, (or `EvSetCursor` and `EvLButtonDown` in Borland/CodeGear's OWL) (or `FormMouseMove` and `FormMouseDown` in Borland/CodeGear's VCL).

3.10. Arbitrarily Shaped ROI (ERegion)

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

Regions or arbitrarily shaped ROI

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

In Open eVision:

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

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

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

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



TIP

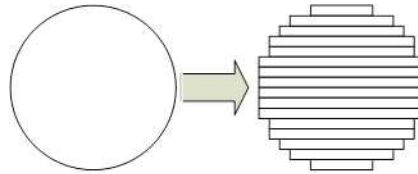
In the future Open eVision releases, the support of **ERegions** will be gradually extended to all operators.

Creating regions

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

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

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



Run-Length Encoding of a circle-shaped region

To create a region, either:

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

Geometry-based regions

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

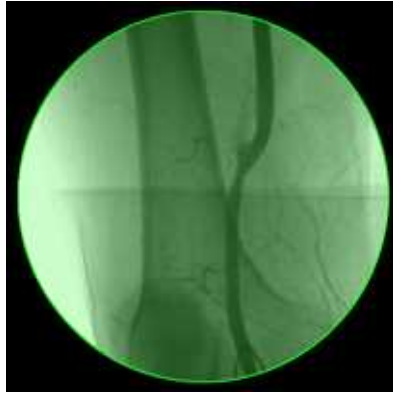
Use these classes to setup geometric regions and modify them with translation, rotation and scaling. The transformation operators return new regions, leaving the source object unchanged.

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



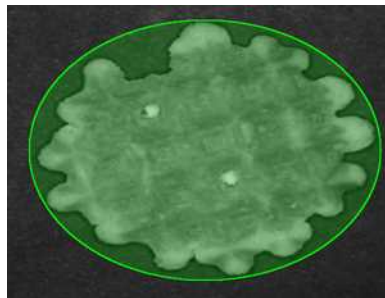
Rectangle region separating a bar code from the background

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



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

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



Ellipse region encompassing a waffle

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



Polygon region encompassing a key

Using the result of other tools

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

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



Open eVision provides constructors for the following tools:

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



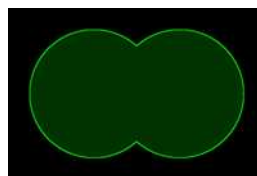
TIP

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

Combining regions

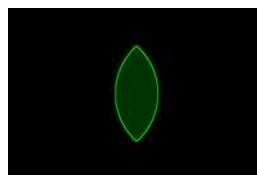
Use the following operations to create a new region by combining existing regions:

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



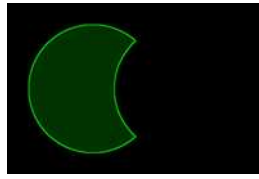
Union of 2 circles

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



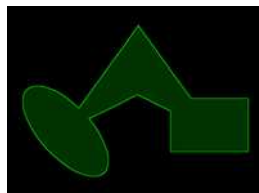
Intersection of 2 circles

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



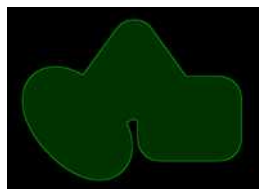
Subtraction of 2 circles

Morphological operations on regions



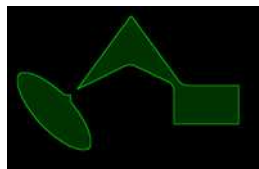
The initial arbitrary region used to illustrate the different morphological operations

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



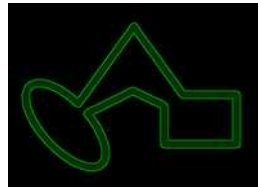
Grow of the arbitrary region

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



Shrink of the arbitrary region

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



Contour of the arbitrary region

Free-hand drawing a region

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

Using regions

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

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



NOTE

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

Preparing the region

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

Drawing regions

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

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

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

ERegions and EROIs

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



TIP

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

ERegion and 3D

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

3.11. Flexible Masks

ROIs vs flexible masks

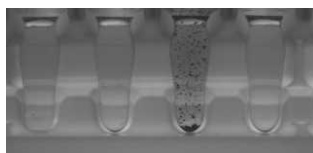
ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 26 apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
- Flexible Masks are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some `EasyObject` and `EasyImage` library functions.

Flexible Masks

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

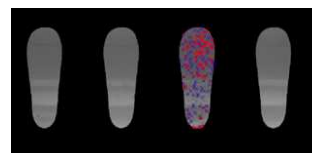
- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image



Associated mask

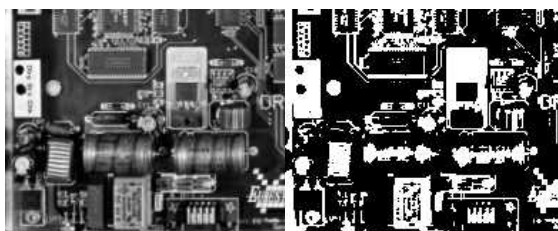


Processed masked image

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

Flexible Masks in EasyImage

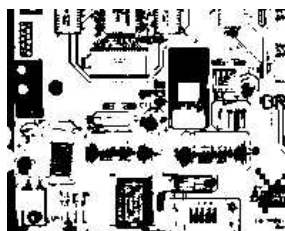
Code Snippets



Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

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



Resulting image

EasyImage Functions that support flexible masks

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

Flexible Masks in EasyObject

A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.

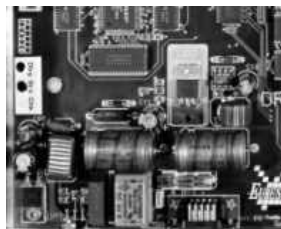
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

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

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

EasyObject functions that create flexible masks

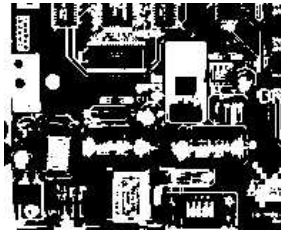


Source image

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

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

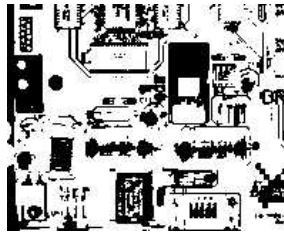
5. Exploit the flexible mask as an argument to `ECodedImage2::RenderMask`.



BW8 resulting image that can be used as a flexible mask

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

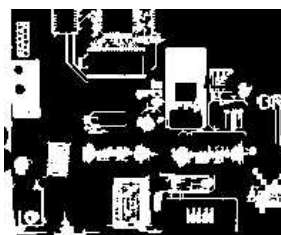
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

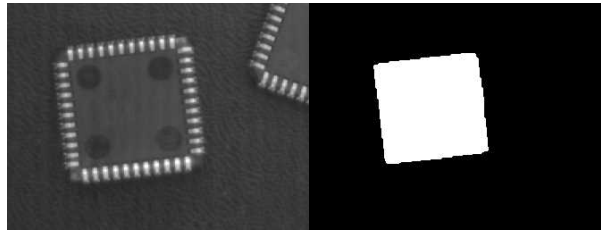
3) `EObjectSelection::RenderMask`: from a selection of blobs

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



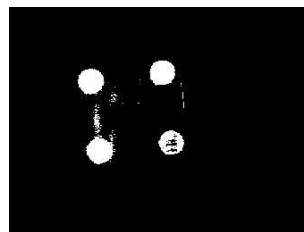
BW8 resulting image that can be used as a flexible mask

Example: Restrict the areas encoded by EasyObject



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

1. Declare a new `ECodedImage2` object.
2. Setup variables: first declare source image and flexible mask, then load them.
3. Declare an `EImageEncoder` object and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
4. Encode the source image. Encoding a layer with just the area in the flexible mask is then pretty straightforward.
We see that the circles are correctly segmented in the black layer with the [grayscale single threshold segmenter](#):



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

3.12. Profile

Code Snippets

Profile Sampling

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

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

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

Profile Analysis

The profile can be processed to find peaks or transitions:

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

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

Profile Insertion Into an Image

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

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

PART II
GENERAL PURPOSE LIBRARIES

1. EasyImage - Pre-Processing Images

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.

1.1. Intensity Transformation

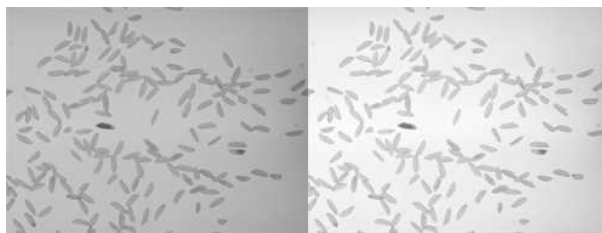
These EasyImage functions change the gray-levels of pixels to increase contrast.

Gain offset

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

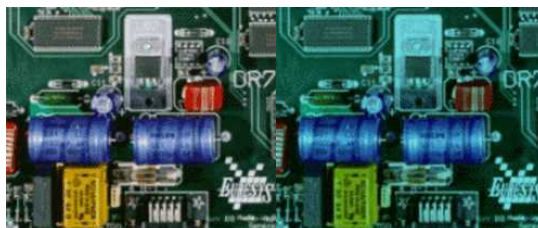
- **gain** adjusts **contrast**. It should remain close to 1.
- **offset** adjusts **intensity** (brightness). It can be positive or negative.
- The resulting values are always saturated to range [0..255].

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).

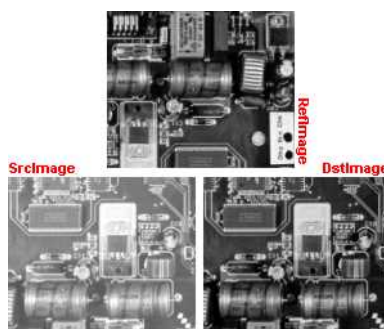


Example of gain/offset applied on a color image

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.

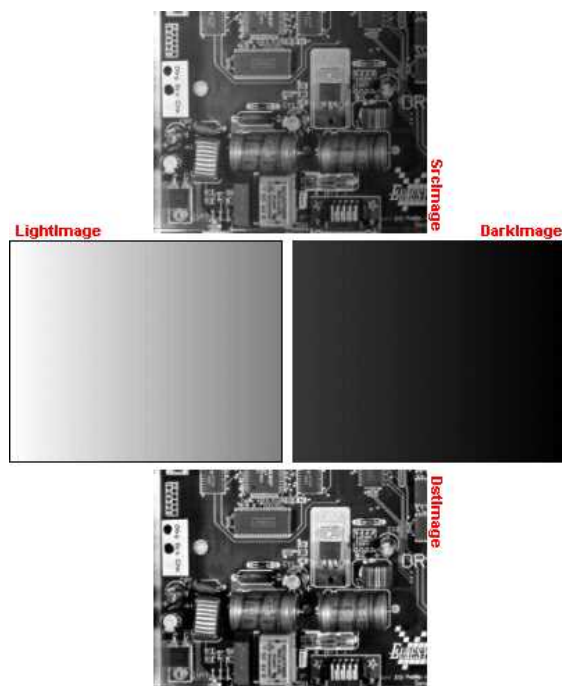


The reference image (from which the average and standard deviation are computed),
the source image (too bright),
and the normalized image (contrast and brightness are the same as the reference image)

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.

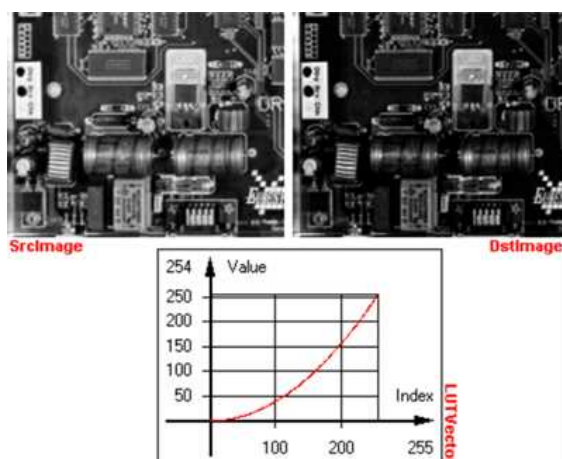
- When one reference image is used, the transformation is similar to an adaptive (space-variant) gain; each pixel in the reference image encodes the gain for the corresponding pixel in the source image.
- When two reference images are used, the transformation is similar to an adaptive gain and offset;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

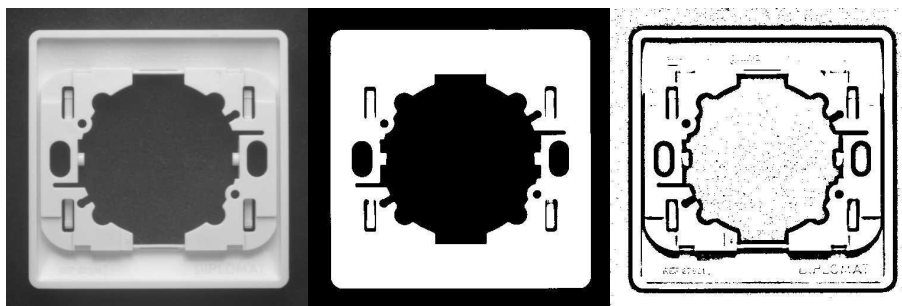
Lut uses a lookup table of new pixel values to replace the current ones - efficient for BW8 and BW16 images. If the transform function never changes, it is best to use a lookup table.



Example of a transform

1.2. Thresholding

Code Snippets



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

- "Automatic thresholding" on page 45 (BW8 and BW16 images only)
- "AutoThreshold" on page 46 (BW8 and BW16 images only)
- "Manual thresholding" on page 46 using one or two threshold values
- "Histogram based" on page 46 (computed before using the thresholding function)

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

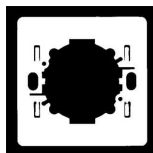
Keys to successful thresholding

- 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

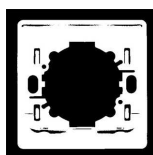
Automatic thresholding

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 the resulting image (default if the `Threshold` function is invoked without an argument).



Max Entropy: Maximizes the entropy (that is, 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 `HistogramThresholdBW6`) 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.

1.3. Arithmetic and Logic

Code Snippets

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 \neq 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 \neq 0

1.4. Non-Linear Filtering

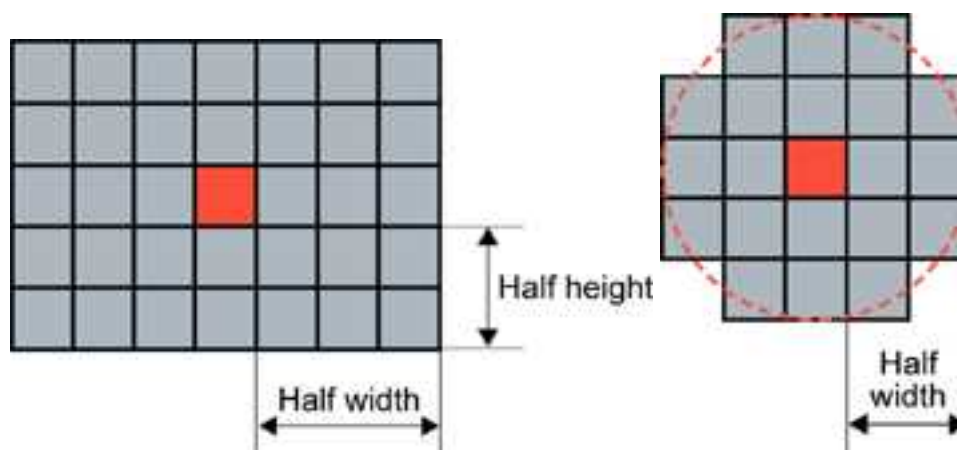
These functions use non-linear combinations of neighboring pixels to highlight a shape, or to remove noise.

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 page 49 (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).*

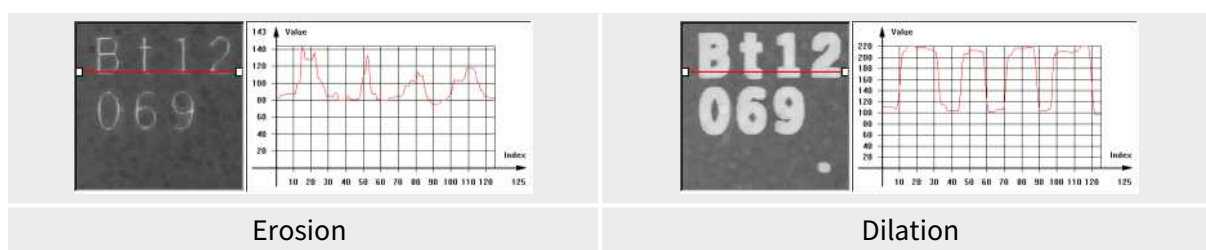
Erosion, Dilation, Opening, Closing, Top-Hat and Morphological Gradient operations all use rectangular or circular kernels of odd size. Kernel size has an important impact on the result.

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.

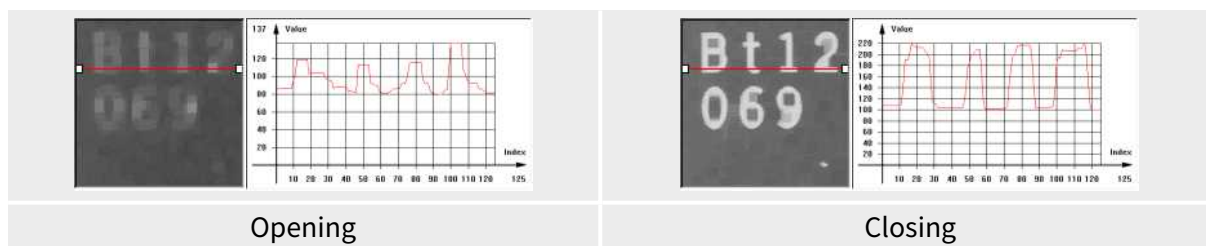


Erosion thins white objects by removing a layer of pixels along the objects edges: [ErodeBox](#), [ErodeDisk](#). As the kernel size increases, white objects disappear and black ones get fatter.

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.



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:

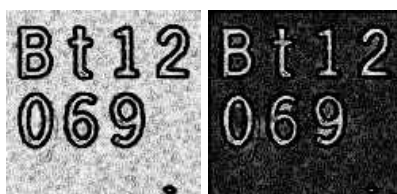
- Thinning: can help edge detectors by reducing lines to single pixel thickness.
- Thickening: can help determine approximate shape, or skeleton.

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: [BlackTopHatBox](#), [BlackTopHatDisk](#).

Morphological distance

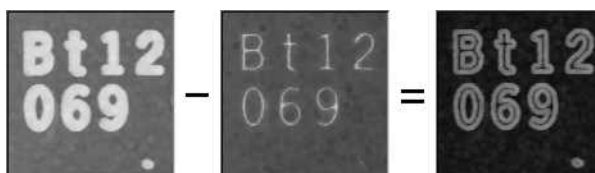
[Distance](#) computes the morphological distance (number of erosion passes to set a pixel to black) of a binary image (0 for black, non 0 for white) and creates a destination image, where each pixel contains the morphological distance of the corresponding pixel in the source image.

Morphological gradient

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

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

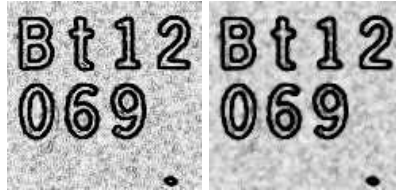
[MorphoGradientBox](#), [MorphoGradientDisk](#).



Dilation - Erosion = Gradient

Median

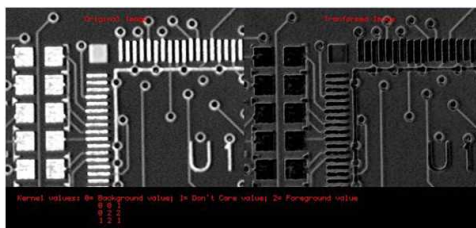
The **Median** filter removes impulse noise, whilst preserving edges and image sharpness. It replaces every pixel by the median (central value) of its neighbors in a 3x3 square kernel, thus, outer pixels are discarded.



Median filter: source and destination images

Hit-and-Miss transform

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



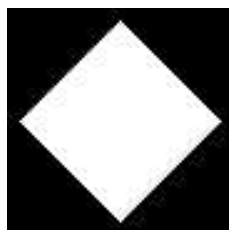
Hit-and-miss transform

The `HitAndMiss` function has three arguments:

- A pointer to the source image of type `EROIBW8`, `EROIBW-6`, 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:
 - $\text{kernel width} = (\text{endX} - \text{startX} + 1)$, $\text{kernel height} = (\text{endY} - \text{startY} + 1)$
 - `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:
 - $\text{kernel width} = ((2 \times \text{halfSizeX}) + 1)$, $\text{kernel height} = ((2 \times \text{halfSizeY}) + 1)$
 - $\text{kernel StartX} = -\text{halfSizeX}$, $\text{kernel StartY} = -\text{halfSizeY}$

example: detecting corners in a binary image.

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

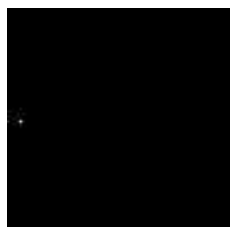


Binary source image

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:

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

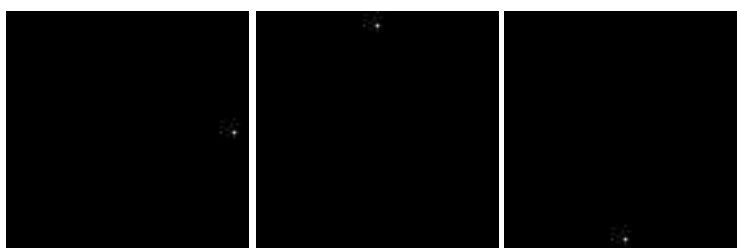
2. Apply the filter on the source image. Note that the resulting image should be properly sized.



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.



1.5. 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:

Re-alignment

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).

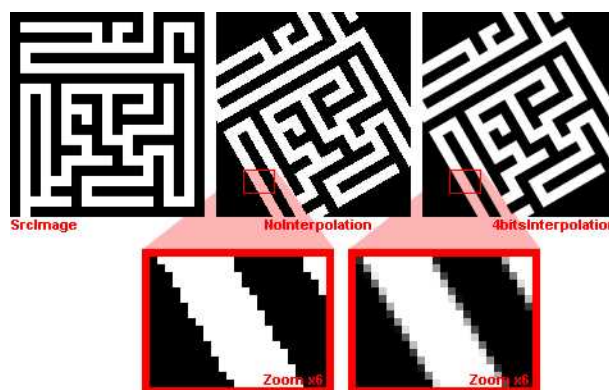
Translation, Scaling and Rotation

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 unwarped 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.

1.6. Noise Reduction and Estimation

Code Snippets

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.

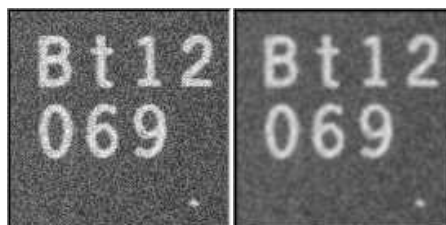
Spatial noise reduction (if you only have 1 image)

Reduces uniform and impulse noise but blurs edges.

Cannot distinguish noise from actual signal changes, so always spoils part of the signal.

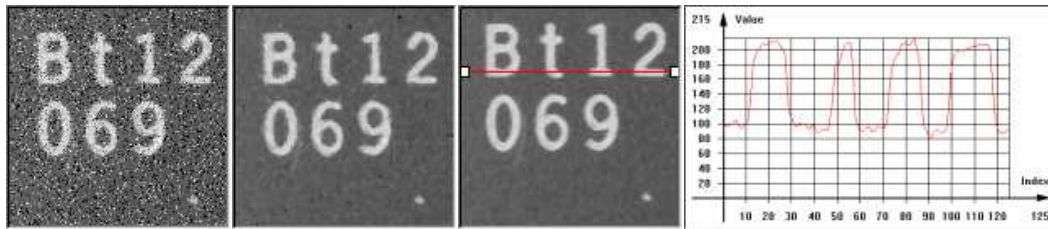
Uses the correlation between neighboring pixel values to perform convolution or median filtering:

- **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

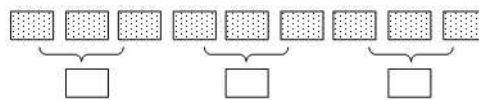
- [EasyImage::Median](#)
- [EasyImage::BiLevelMedian](#)

Temporal noise reduction (for several images, such as moving objects)

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

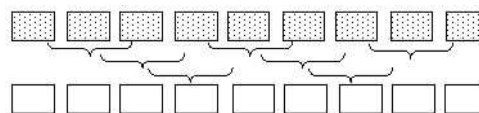
EasyImage provides three ways to minimize noise by means of several images:

- **Temporal average:** just accumulates N images and average them; using standard arithmetic operations, as illustrated below. Creates denoised 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 denoised image after N acquisitions only, is that fast display refresh is not possible.



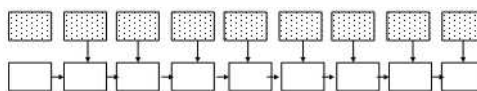
Simple average

- **Temporal moving average:** accumulates the last N images and updates the denoised 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 denoised 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:

$$Dst_N = a \times Src + (1 - a) \times Dst_{N-1}$$

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 precomputed 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.

1.7. 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/EImageBW6`.

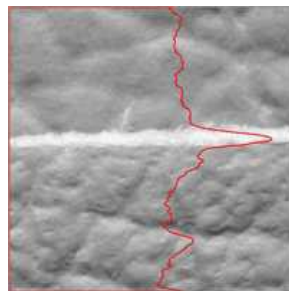
Use `EasyImage::ArgumentImage` or `EasyImage::ModulusImage` once before calling `GradientScalar`.

1.8. Vector Operations

Code Snippets / Code Snippets

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:

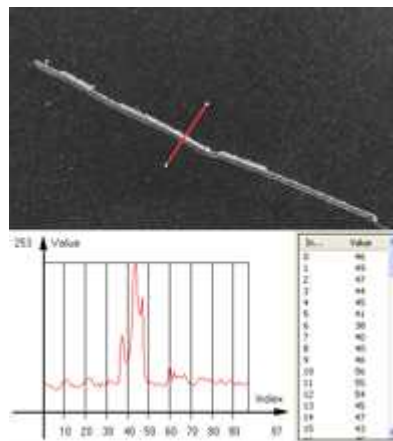
Projection



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`, `EBW6Vector` or `EC24Vector`). These functions support **flexible masks**.

- `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 masks.

`EasyImage::ImageToPath` copies the corresponding pixel values to the vector. This function supports flexible masks. 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.

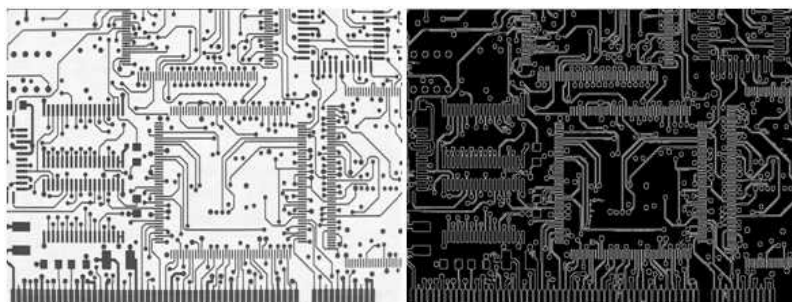
`EasyImage::PathToImage` copies pixel values from a vector or a constant to the pixels of a given path.

1.9. Canny Edge Detector

Code Snippets

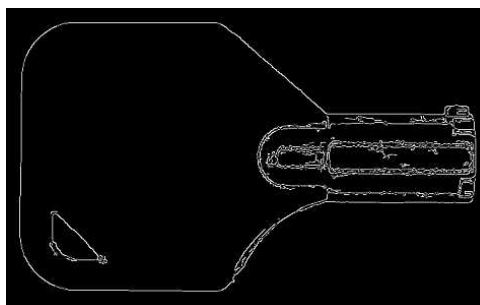
The Canny edge detector facilitates:

- 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:

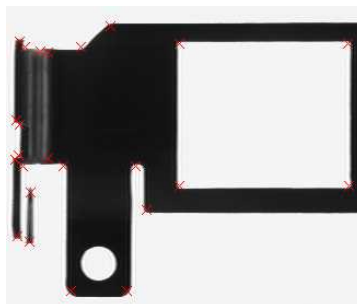
- [Apply](#): applies the Canny edge detector on an image/ROI.
- [GetHighThreshold](#): returns the high hysteresis threshold to consider that a pixel is an edge.
- [GetLowThreshold](#): returns the low hysteresis threshold to consider that a pixel is an edge.
- [GetSmoothingScale](#): returns the scale 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 to consider that a pixel is an edge.
- [SetLowThreshold](#): sets the low hysteresis threshold to consider that a pixel is 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.

1.10. Harris Corner Detector

Code Snippets

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 σ_i : the standard deviation of the Gaussian Filter used for scale analysis.
 $\sigma_d = 0,7 \times \sigma_i$, where σ_d 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.

1.11. 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.

1.12. 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.

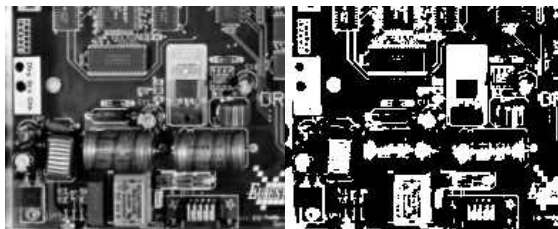
The same image should be used as source and destination because only the shifted rows are copied. To use a different destination image, the source image must be copied first in the destination image object.

The size of the destination image is determined as follows:

$$\begin{aligned} dstImage_Width &= srcImage_Width \\ dstImage_Height &= (srcImage_Height + 1 - odd) / 2 \end{aligned}$$

1.13. Flexible Masks in EasyImage

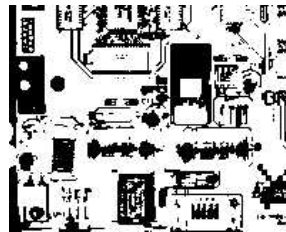
Code Snippets



Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

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



Resulting image

EasyImage functions that support flexible masks

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

1.14. Computing Image Statistics

Code Snippets

`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.

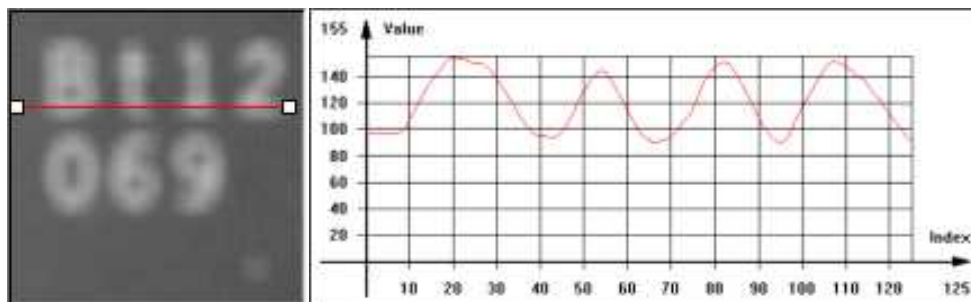


NOTE

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

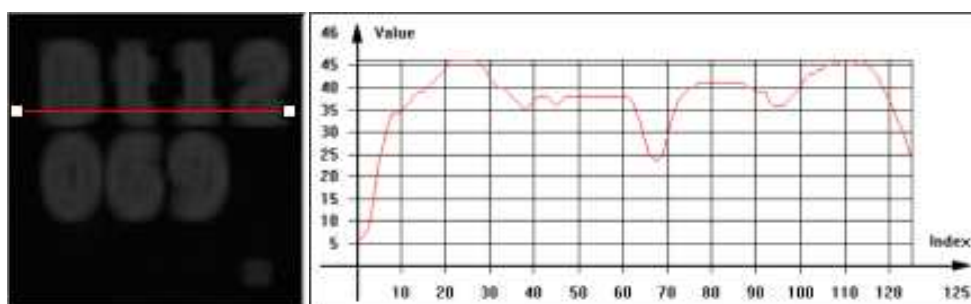
The result of the operation is another image.

The **local average**, [EasyImage::LocalAverage](#), corresponds to a strong low-pass filtering.



Sliding window average

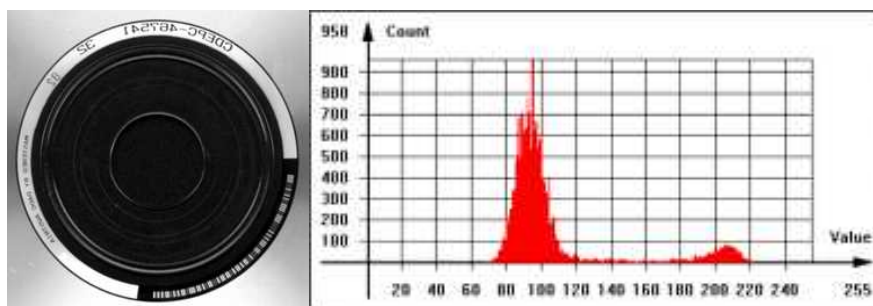
The **local standard deviation**, [EasyImage::LocalDeviation](#) enhances the regions with a high frequency contents, such as noisy or textured areas.



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 can take a flexible mask as input argument.

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

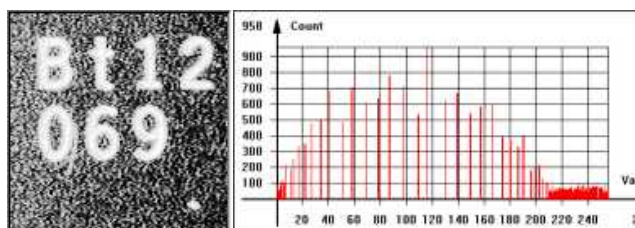
`EasyImage::AnalyseHistogramBW-6` provide statistics and thresholding values:

- 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.

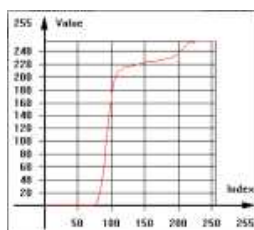
This may be useful to maximize image contrast, or reveal a lot of image details in dark areas.



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.

2. EasyColor - Pre-Processing Color Images

Code Snippets

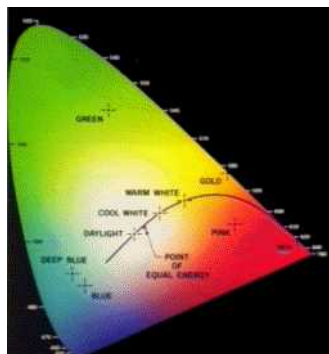
EasyColor makes color image processing as efficient as possible by detecting, classifying and analyzing objects. Several conversion functions mean that any color system can be processed.

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 76: `EasyImage::GainOffset`,
- "LUT for Color Calibration" on page 77: `EColorLookup::Calibrate`,
- "LUT for Color Balance" on page 77: `EColorLookup::WhiteBalance`,

- `EColorLookup::ConvertFromRGB`, `EColorLookup::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.

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.



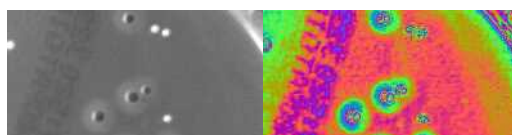
NOTE

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.

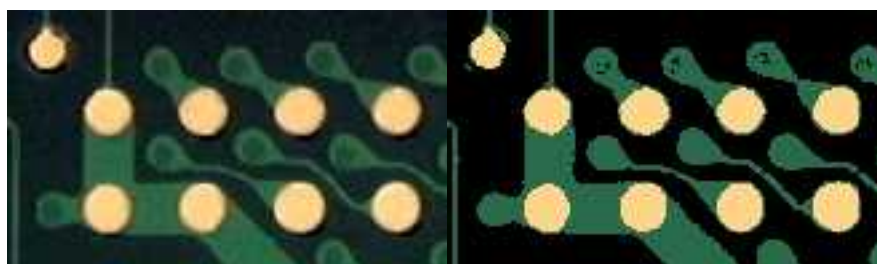
To define pseudo-color shades, you specify a trajectory in the color space of an arbitrary system. You can then pseudo-color using the drawing functions color palette (see [Image](#) and [Vector Drawing](#)) then save and/or transform it like any other color image.



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)

2.1. Bayer Conversion

[Code Snippets](#)

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. That filter is also named *Color Filter Array* or *CFA*.

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



Bayer vs. true color format



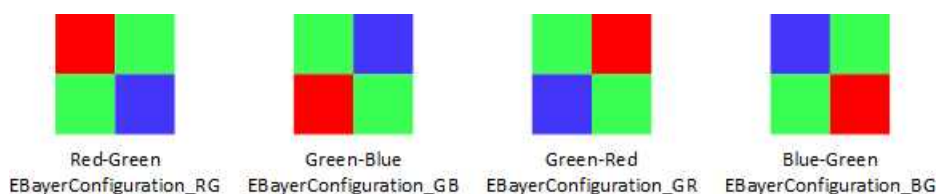
TIP

- 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.
- Parity adjustment is not necessary 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. That process is also known as *demosaicing*.

Along with the gray-level input image, the Bayer configuration is mandatory.

There are 4 different arrangements of the Bayer pattern, defined by the first 2 pixels of the first row of the image:



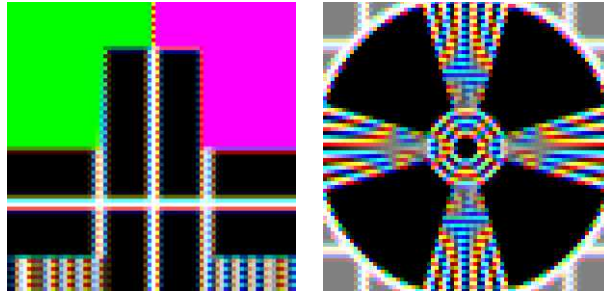
Several methods are available to reconstruct the missing pixels.

- Some are fast but the resulting image may have artifacts, like the zipper effect or color aliasing.
- Some are slower but achieve better interpolation and produce less artifacts.

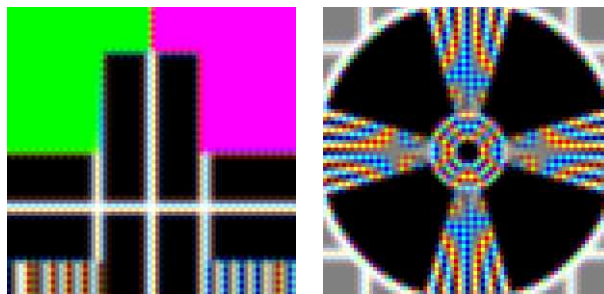
Interpolation modes

The frame rate is given for the conversion of a 1280 x 720 image on a single core Intel I7-6600U CPU.

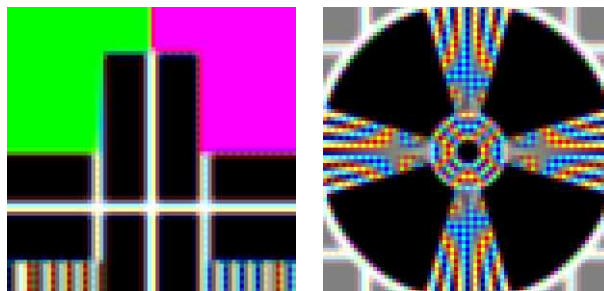
- Mode 0
 - No interpolation
 - Frame rate: 943



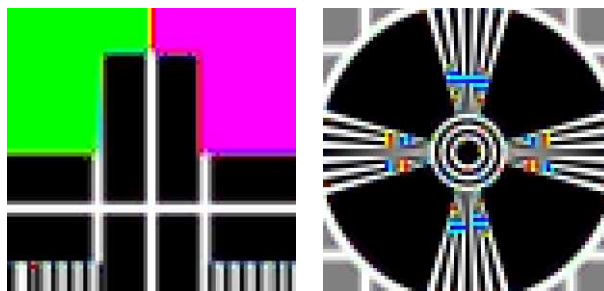
- Mode 1
 - Linear interpolation on a 3x3 kernel
 - Frame rate: 2159



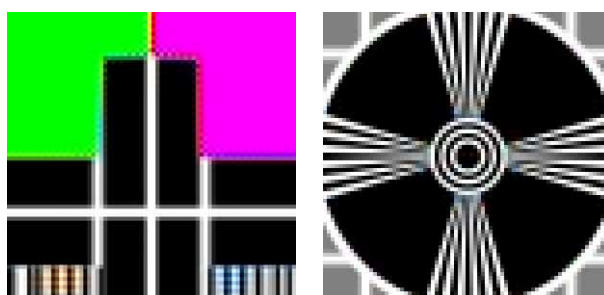
- Mode 2
 - Advanced interpolation on a 3x3 kernel
 - Frame rate: 1303



- Mode 3
 - Interpolation on a 5x5 kernel
 - Frame rate: 449



- Mode 4
 - Interpolation on 9x9 kernel
 - Frame rate: 22



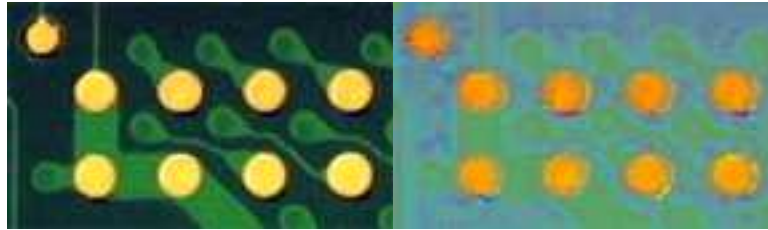
The method `EasyColor::C24ToBayer` is the reciprocal function of `EasyColor::BayerToC24`. It converts RGB color pixel images to Bayer images using the given `EBayerConfiguration`.

A Bayer encoded image is not compatible with a true color image (EC24), but you can apply white balance and gamma correction with the `EColorLookup` parameter in `EasyColor::TransformBayer`.

2.2. 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.

2.3. 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.

2.4. 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 only needs to be prepared once (it implements a decoupled color transformation).

Gamma precompensation

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

Gamma precompensation should be used after processing because using it before would change the result because of the nonlinearity introduced.

The precompensation process applies the inverse transform to the signal, so that the image renders correctly on the display. Three predefined 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**

Precompensation cancellation and pure precompensation correspond to exponents that are inverse of each other.

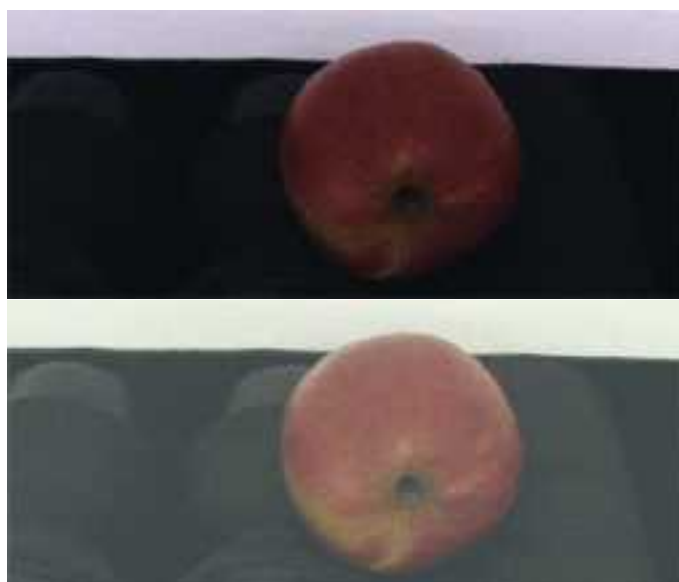
Gamma precompensation cancellation

Many color cameras have a built-in gamma precompensation 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 predefined 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, that is, 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 precompensation

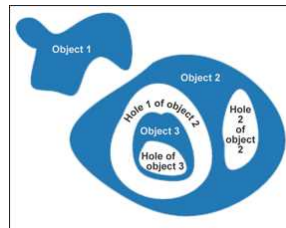
PART III

*MATCHING AND
MEASUREMENT TOOLS*

1. EasyObject - Analyzing Blobs

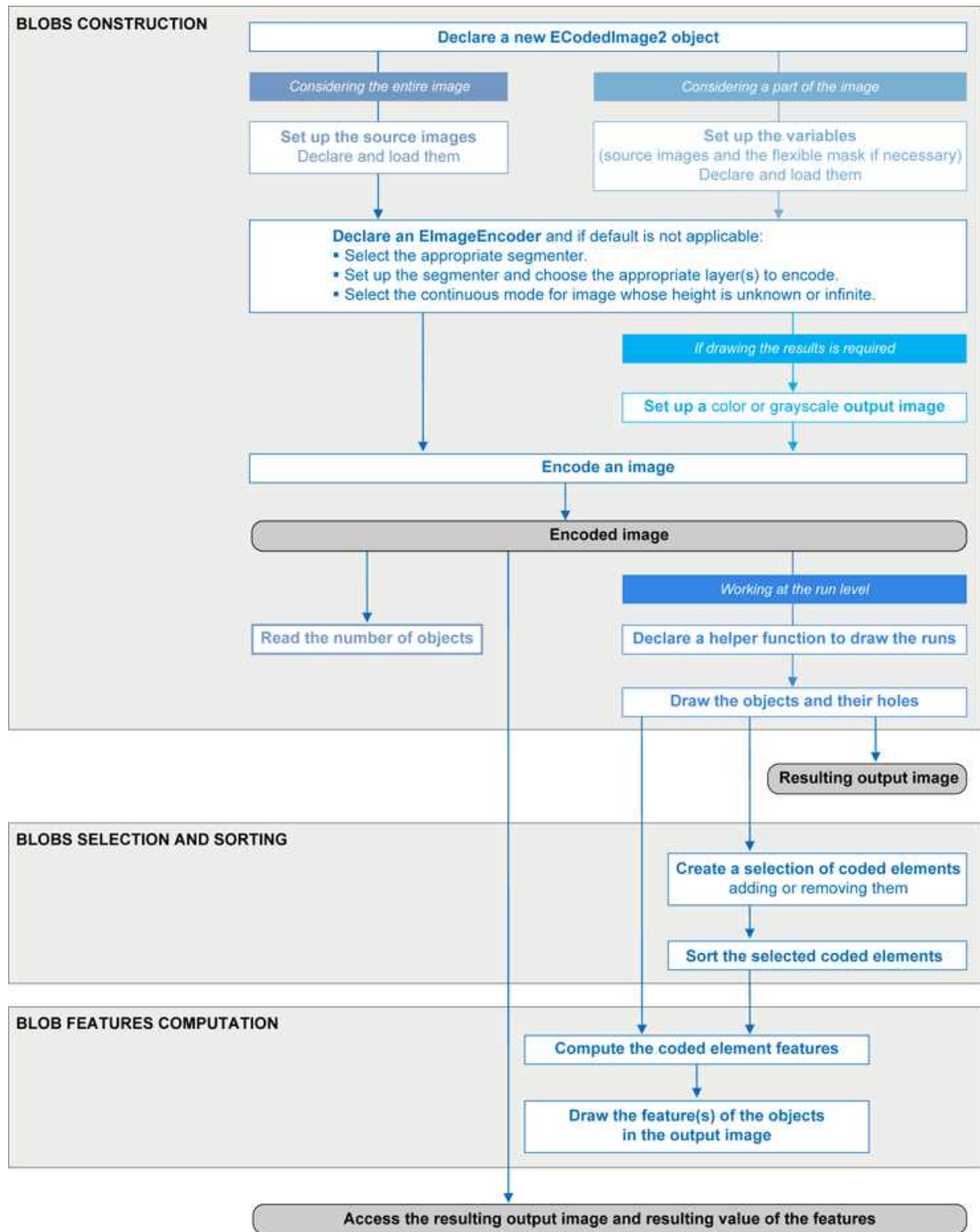
[Reference](#) | [Code Snippets](#)

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



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

Workflow



Blob Definition

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

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

Blobs are handled as independent entities:

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

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

Build Blobs

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

1. **Segment**: classifies the source image pixels, creates layers, and constructs the runs (a run is a sequence of adjacent pixels in a row, that share the same property).
2. **Encode**: assembles runs, to build blobs for each layer.

You select which objects or holes are kept.

`EImageEncoder::Encode` analyzes the blobs and stores the result into a coded image which has a set of superimposed, mutually exclusive image layers, where the pixels of each layer have properties in common, such as being above a threshold.

Flexible masks can restrict encoding to an arbitrary shaped area.

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

Functions

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


Pixel aggregation (encoder)

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

Extract objects (using geometric parameters)

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

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



Features computation and display

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

1.1. Image Segmenters

Code Snippets

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

1. Grayscale Single Threshold (default)

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

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

EasyObject provides 5 thresholding methods:

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

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

2. Grayscale Double Threshold

[EGrayscaleDoubleThresholdSegmenter](#) is applicable to BW8 and BW16 grayscale images and produces coded images with three layers:

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

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

3. Color Single Threshold

[EColorSingleThresholdSegmenter](#) is applicable to C24 color images; it produces coded images with two layers:

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

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

4. Color Range Threshold

[EColorRangeThresholdSegmenter](#) is applicable to C24 color images; it produces coded images with two layers:

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

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

5. Image Range

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

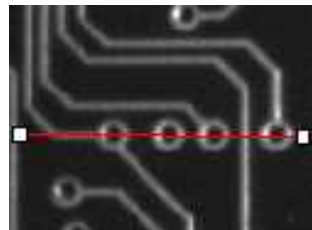
- when the background is not uniform enough,
- when the illumination is not uniform across the image,
- when only differences between the image and a reference image (ideal) are to be enhanced,

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

The difficulty is preparing suitable high and low reference images.

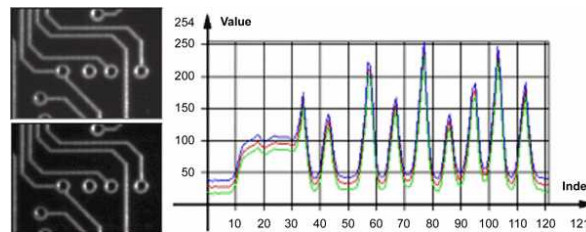
Preparing high and low reference images

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



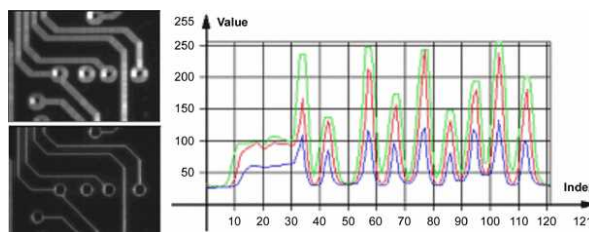
Source image

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



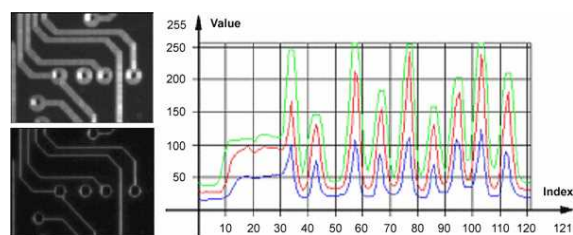
Gray-level tolerance margins

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



Geometric tolerance margins

Combining both kinds of tolerance margins gives the best results.



Combined margins

Image Segmenter

[EImageRangeSegmenter](#) and [EReferenceImageSegmenter](#) are applicable to BW8, BW16, and C24 images; and produce coded images with two layers.

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

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

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

- BW8: [SetLowImageBW8](#) [GetLowImageBW8](#)
- BW16: [SetLowImageBW-6](#) [GetLowImageBW-6](#)
- C24: [SetLowImageC24](#) [GetLowImageC24](#)

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

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

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

- BW8: [SetReferenceImageBW8](#), [GetReferenceImageBW8](#)
- BW16: [SetReferenceImageBW-6](#), [GetReferenceImageBW-6](#)
- C24: [SetReferenceImageC24](#) , [GetReferenceImageC24](#)

6. Labeled Image

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

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

7. Binary Image

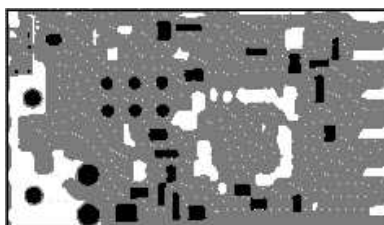
[EBinaryImageSegmenter](#) is applicable to BW1 binary images; it produces coded images with two layers:

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

1.2. Image Encoder

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

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



Object building

Selecting the Layers to Encode

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

Function [GetMaxLayerIndex](#) returns the highest Layer Index value. It is available for all segmenters.

Enabling/disabling layer encoding for each layer individually

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

Two-layer segmenters

Layer	Set LayerEncoded function	Get LayerEncoded function	Default value
Black layer	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

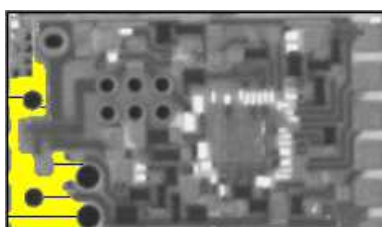
Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerIndex	IsBlackLayerIndex	0
White layer	SetWhiteLayerIndex	IsWhiteLayerIndex	1

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

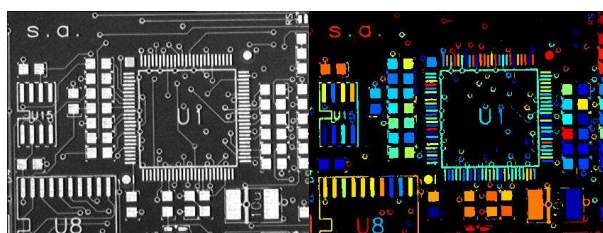
For the sake of computational efficiency, the objects are described as lists of runs. A run is a sequence of adjacent pixels that share homogeneous properties (such as being above a given threshold). These runs are merged in objects by the image encoder.



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



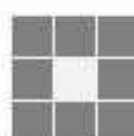
Source image (left) with the white layer rendered (right)

Connexity

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



4-connexity



8-connexity

Multiple images can be encoded in [continuous mode](#).

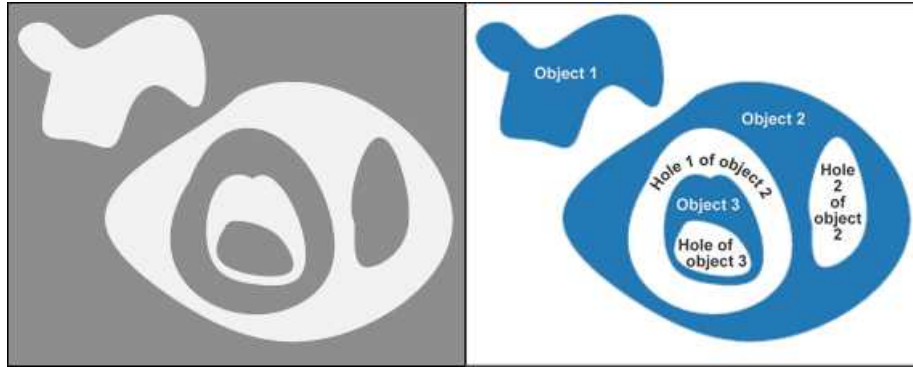
1.3. Holes Construction

Code Snippets

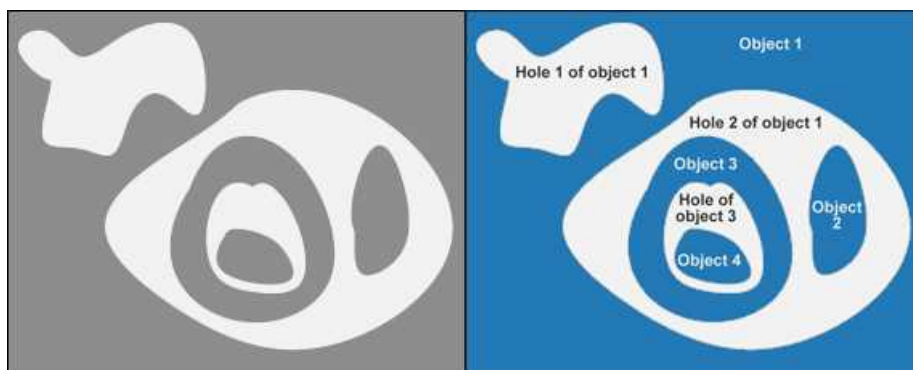
A hole is a set of connected pixels that are entirely surrounded by a parent object (4 or 8 pixels depending on the connexity mode).

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

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



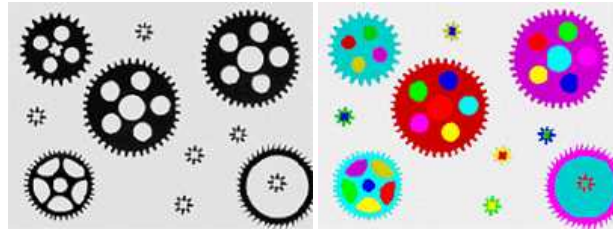
Encoding the white layer (3 objects and 3 holes)



Encoding the black layer (4 objects and 3 holes)

How to Color holes

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



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

1.4. Normal vs. Continuous Mode

Code Snippets

Normal Mode (default)

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

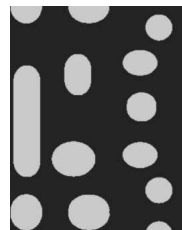
Continuous Mode

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

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

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

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

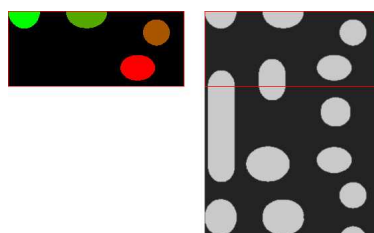


Original image

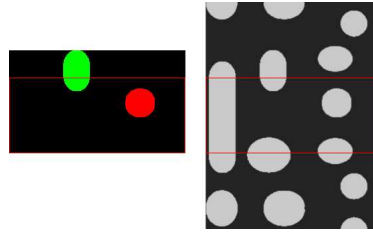


Three chunks of the image

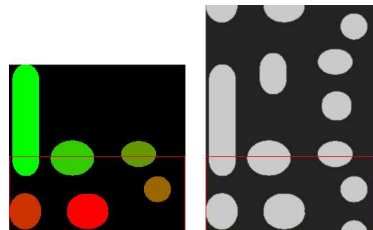
1. **Draw the objects encoded in a layer of a coded image.** This code is essentially the same as in "Browsing Runs" code snippet. The only difference is that an offset can be applied along the Y-axis.
2. **Define a function to draw the objects of a layer.** If a coded image contains objects that were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



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



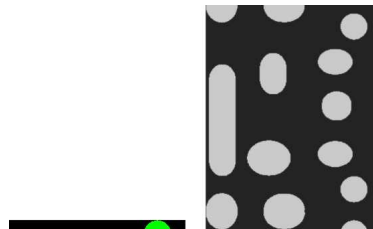
When count reaches 1, one of these two objects becomes completed, which leads to the encoding of the following image. Two other objects are not encoded yet at this time. Here is the result of the encoding of the last chunk (count = 2).



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

Flushing Continuous Mode

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



Result of the flush

1.5. Selecting and Sorting Blobs

Code Snippets

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

Create / modify a selection

You can use the methods [Add](#) and [Remove](#) of the class [EObjectSelection](#) to:

- Add or remove a single object , a hole or a whole layer to/from a selection.

- Add or remove objects or holes based on some specified **feature** (see the feature list in [Computing the Coded Element Features](#)).
- Add or remove objects or holes based on their specific **position**, or whether they lie within a specified ROI rectangle.
- Add or remove objects based on their specific position, or whether they lie outside, on or within a specified ROI rectangle or **ERegion** ([AddObjectsUsingRectangle](#) and [AddObjectsUsingRegion](#)).

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

Clear a selection

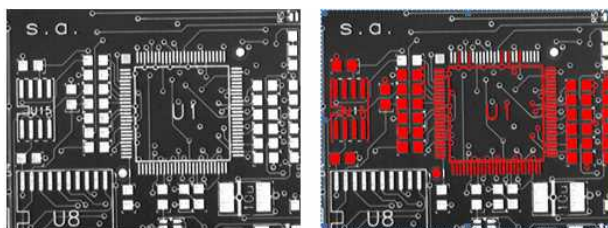
You can clear a previous selection using `EObjectSelection::Clear`.

Sort a selection

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

Example

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

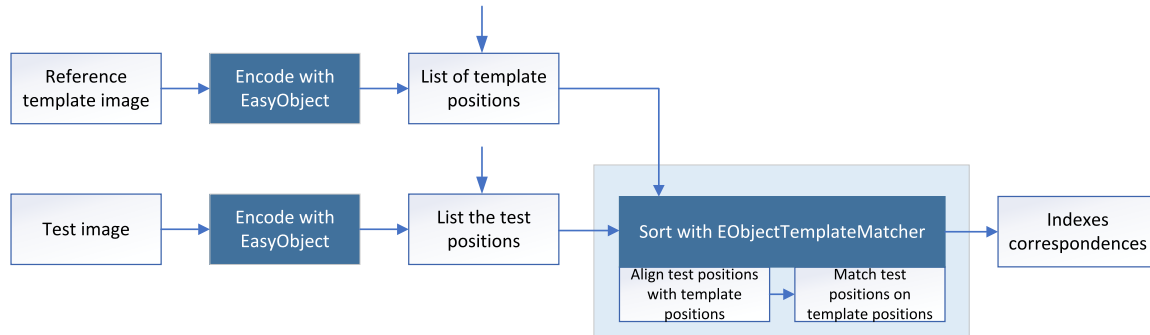


Source image, and selection of objects

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

1.6. Object Template Matcher

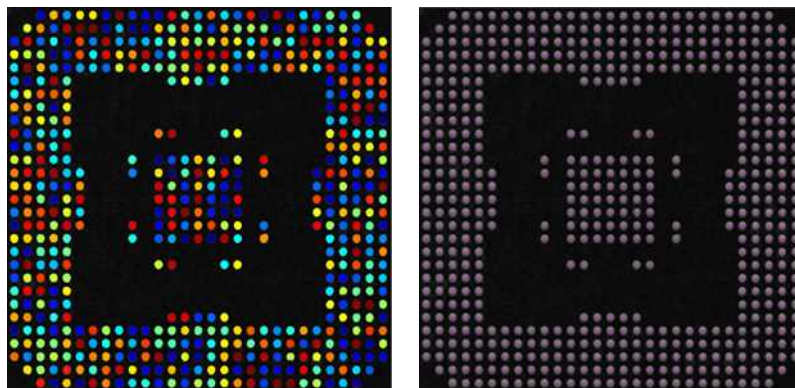
The class `EObjectTemplateMatcher` is a tool designed to align and match the output of EasyObject to a reference template. It is designed and developed to handle efficiently thousand of objects.



Creating the reference template

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

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

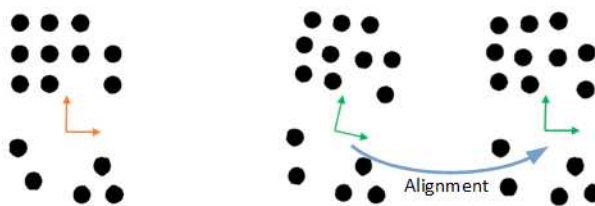


An encoding of the reference image for use as the template.
And the center positions of each object for use in the matching process.

Sorting the objects

- To perform the matching after setting up the template:
 - Use the method `SortSelection` with an `EObjectSelection` as parameter.
 - Or use the more generic method `SortPositions` with a `std::vector<EPoint>` as parameter.
- When you pass the objects in a selection as the sort method parameter, the bounding box center of the objects is the position used for the matching with the template.

- Before the sorting, [EObjectTemplateMatcher](#) performs an optional global rigid alignment of the submitted positions with the defined template.
 - This alignment only applies the translation and rotation transformations.
 - Use the method [SetEnableAlignment](#) to enable the alignment process.

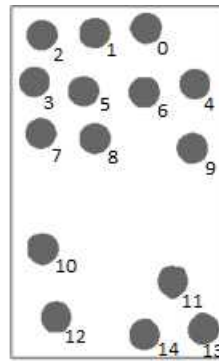


Left: the template.
Right: the alignment of the submitted selection.

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

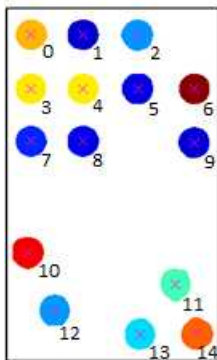
Retrieving the Sorting Results

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

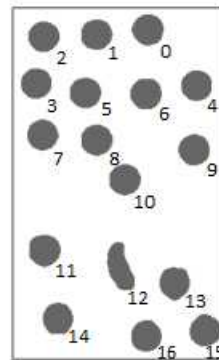


```
NumberOfPairedObjects = 15
SelectionIndexes =
2, 1, 0, 3, 5, 6, 4, 7, 8, 9, 10, 11, 12, 14, 13
TemplateIndexes =
2, 1, 0, 3, 6, 4, 5, 7, 8, 9, 10, 11, 12, 14, 13
UnpairedObjects =
  onlyInTemplate = <>
  onlyInSelection = <>
```

Selection with moved objects

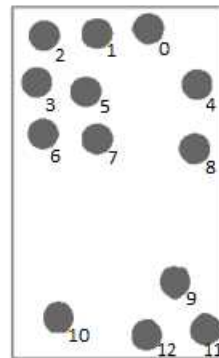


Template with object indexes



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

Selection with extra objects



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

Selection with missing objects

1.7. 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

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

Feret box

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

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

Miscellaneous

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

Ellipse of inertia



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

Ellipse (angle, height, width)	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):

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

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

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

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

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

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

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

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

Convex Hull

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

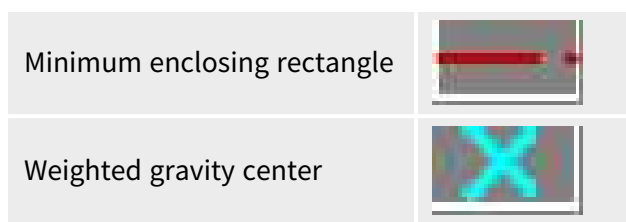
The corresponding function is [ECodedElement::ComputeConvexHull](#).



Graphic Representation

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

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



Coordinate System and Conventions

Coordinate system

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

Angles

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

Evaluating the features

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

Draw Coded Elements

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

To draw coded elements

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

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

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

Flexible Masks in EasyObject

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

A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.

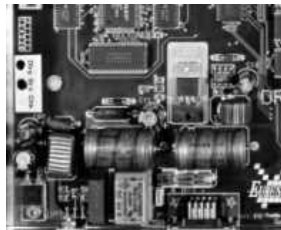
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

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

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

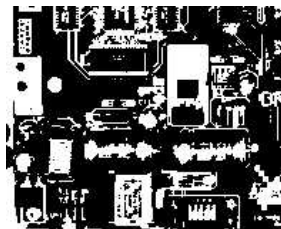
EasyObject functions that create flexible masks



Source image

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

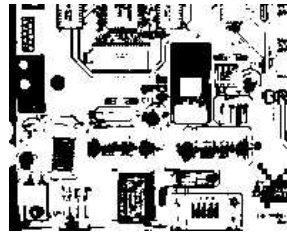
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers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



BW8 resulting image that can be used as a flexible mask

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

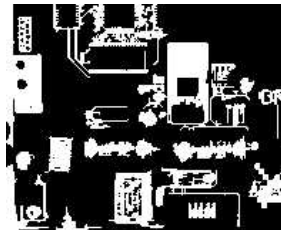
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

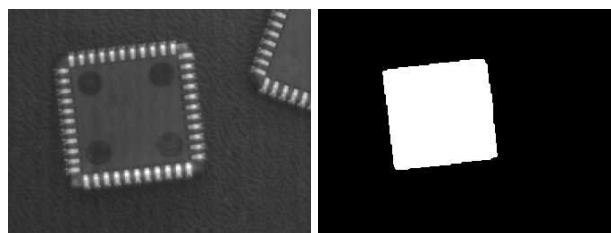
3. EObjectSelection::RenderMask: from a selection of blobs

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



BW8 resulting image that can be used as a flexible mask

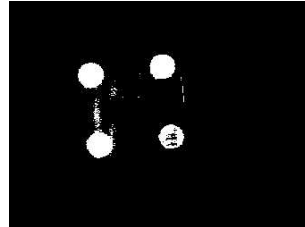
Example: Restrict the areas encoded by EasyObject



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

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

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



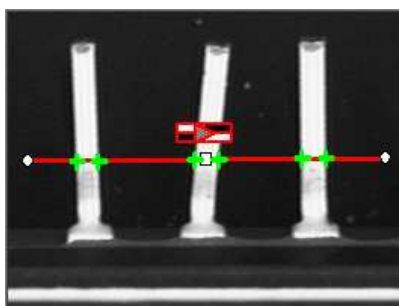
5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

2. EasyGauge - Measuring down to Sub-Pixel

2.1. Workflow

EasyGauge

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



TIP

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

Workflow

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

Choose the workflow that matches the complexity of your model and the accuracy required: uncalibrated, calibrated or grouped.

Uncalibrated Gauging: for a simple model

EasyGauge basic use is straightforward.

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

Uncalibrated gauging is easy to implement but has several drawbacks:

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

Calibrated gauging: for one or two simple measurement sites

Calibrated gauging is more accurate, and measures the inspected parts independently of the viewing conditions.

All measurements are taken in the calibrated units, with any distortion implicitly compensated. Refer to Calibration to learn how to master field-of-view calibration.

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

Complex Gauging

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

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

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



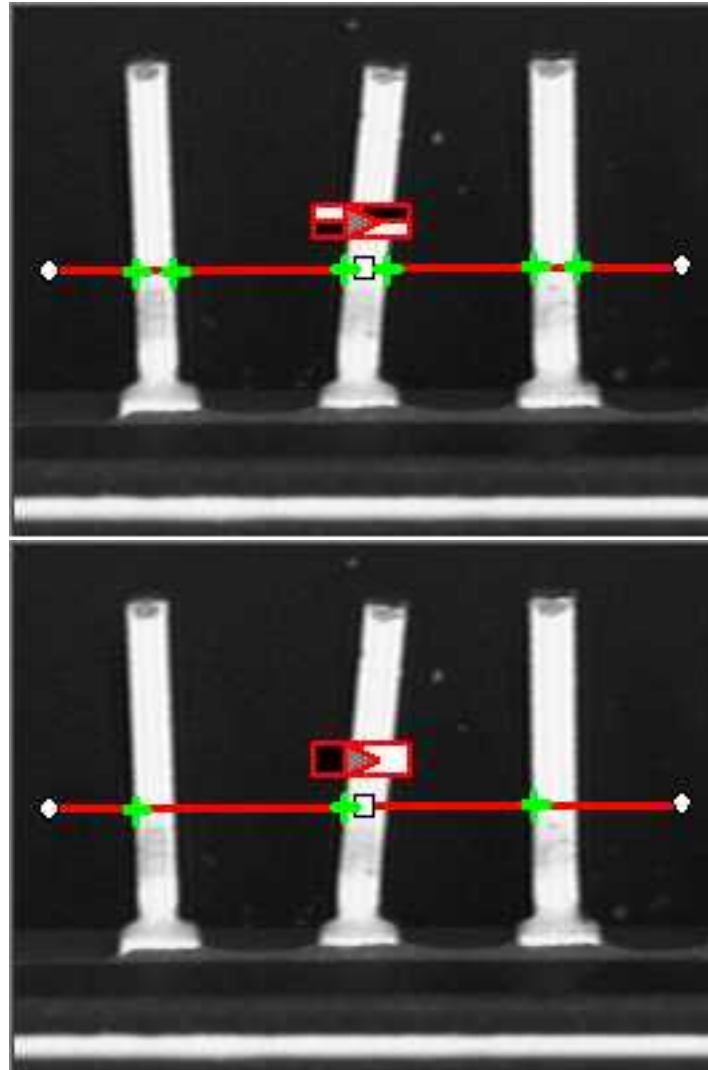
NOTE

Unlike the rest of Open eVision, EasyGauge uses a pixel center origin (see "Image Coordinate Systems" on page 19). The subpixel coordinate (0, 0) is the center of the upper left pixel of the image.

2.2. Gauge Definitions

Point gauge

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



Point location. Contrast-based selection

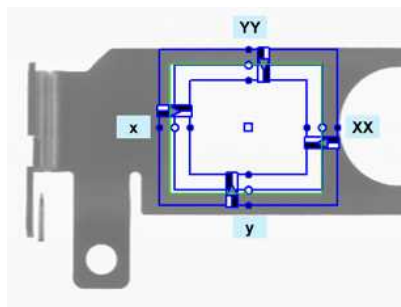
Rectangle Gauge

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

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

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

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



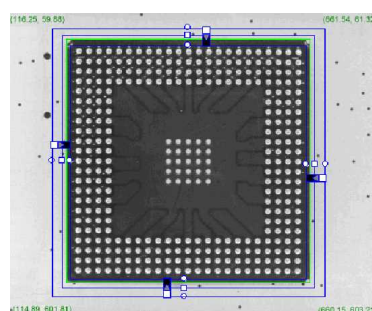
Naming conventions for the sides of a rectangle gauge

Usage

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

Alternatively, [MeasuredRectangle](#) provides the results as an [ERectangle](#) object.

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



Locating a rectangle's corners

Wedge gauge

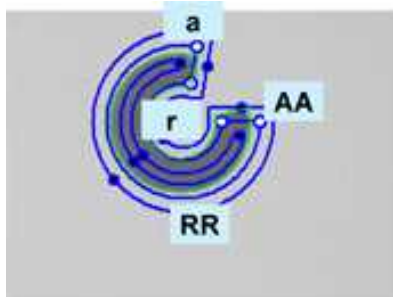
The placement of a [wedge gauge](#) is defined by its nominal position (given by the coordinates of its [center](#)), its nominal [inner](#) and [outer radius](#) ([inner](#) and [outer diameter](#)), its [breadth](#) (difference between radii), the [angular position](#) from where it extends and its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

The [Set](#) member can distinguish between a full ring, a sector of a ring and a disk.

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

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

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



Naming conventions for the sides of a wedge gauge

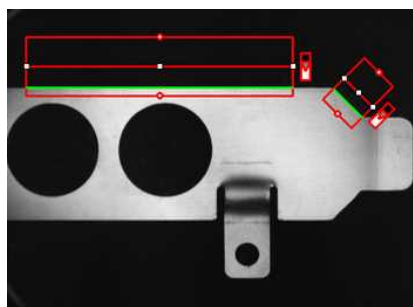
Usage

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

Alternatively, [MeasuredWedge](#) provides the results as an [EWedge](#) object.

Line gauge

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



Line fitting

Usage

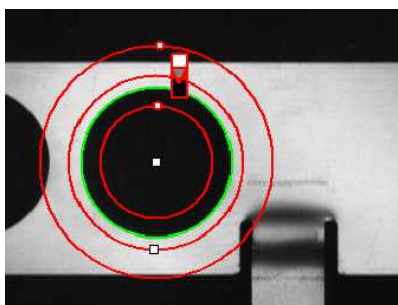
Define and position the gauge, then use [Measure](#) to fit the lines. To obtain the [line properties](#), set the [ActualShape](#) property to **TRUE** to return the fitted line (**TRUE** value) (instead of the nominal line position **FALSE** value, default).

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

Circle gauge

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

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



Circle fitting

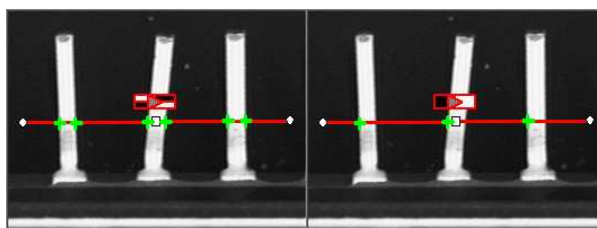
Usage

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

Alternatively, [MeasuredCircle](#) provides the results as an [ECircle](#) object.

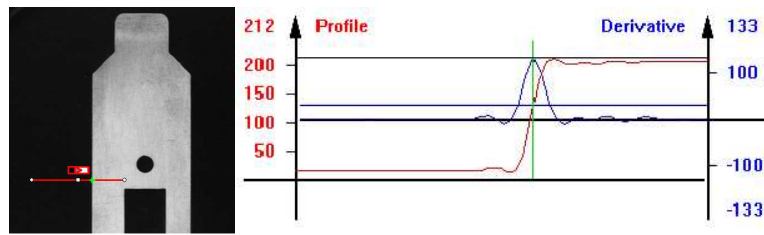
2.3. Find Transition Points Using Peak Analysis

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



Point location. Contrast-based selection

Point Location principle



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

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

EasyGauge extracts the pixel values along a profile (red curve) then uses peak analysis to determine the transition location. All the pixel values in the peak [area1](#) are used to compute the transition location.

- Sub-pixel accuracy is only possible if the transition is surrounded by almost uniform regions of at least 2 pixels wide.
- [BWB2](#) transitions have an increasing profile curve and the peak takes positive values. Otherwise, the curve decreases and the peak extends negatively.
- You cannot normally detect peaks using the default threshold value (20) as BWB or WBW transitions base the peak analysis on the gray level profile along the EPointGauge (or sample path) and not its first derivative.

[EPointGauge](#) contains all point measurement parameters, with default values that detect reasonably contrasted edges.

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

2Black / White / Black

EPointGauge parameters

Center: Nominal point position (will normally be different before and after measurement).

Tolerance: Tolerance value and gauge orientations.

TransitionType, TransitionChoice, TransitionIndex: Peak selection strategies.

Threshold: Noise immunity.

MinAmplitude, MinArea: Peak strength.

Thickness, Smoothing: Local filter widths.

RectangularSamplingArea Sets sampling area (rectangular by default) to transverse filtering mode.

Measure: Measures the object.

- In single transition mode, **Valid** returns True when an appropriate point was found. To obtain measurement results, set **ActualShape** to True so that **Center** returns the located point. (False default value returns nominal point position).

- In multiple transition mode, **NumMeasuredPoints** returns the number of points found,

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

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

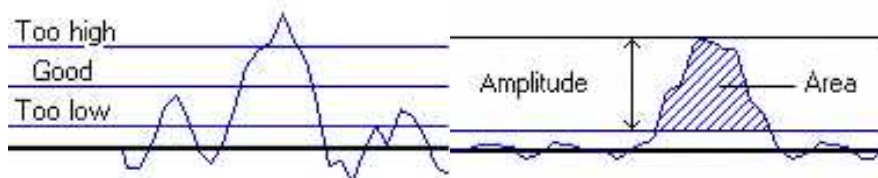
GetMeasuredPeak: Returns **EPeak** containing the peak's **Area** and **Amplitude**, and the delimiting coordinates along the probe segment (**Start**, **Length** and **Center** values).

Select Peaks to improve edge precision

The threshold level is very important:

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

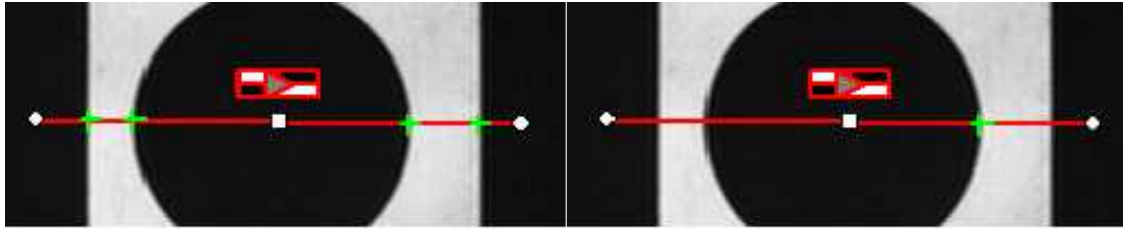
To resolve this dilemma, the EasyGauge peak selection mechanism can reject low contrast or false edges: transition strength is measured by peak *amplitude* and *area*. Every edge measurement determines peak amplitude and area. If either value falls below the **minimum amplitude** or **minimum area**,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



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

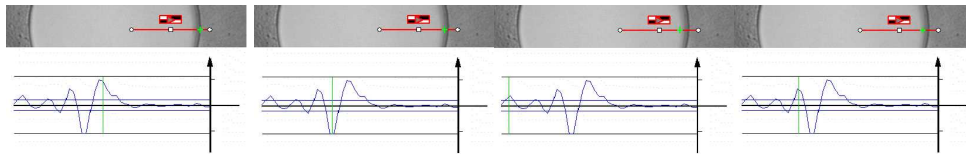
Multiple versus single transition

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



Multiple transition (left) versus single transition (right)

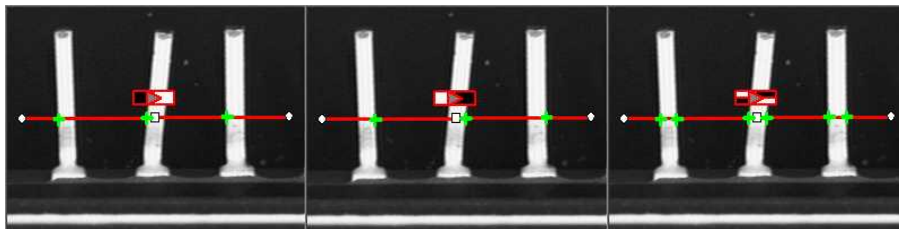
You can select the single most relevant transition based on 4 criteria: the highest peak, the peak with the largest area, the peak closest to the gauge center, or the N-th peak encountered starting from one tip of the gauge.



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

Positive or negative peak selection

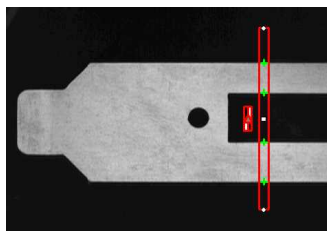
Peak selection can also be refined by choosing the transition polarity: White to Black or Black to White (i.e. positive or negative peak), or indifferent.



Black to white, white to black or indifferent polarities

Prefiltering

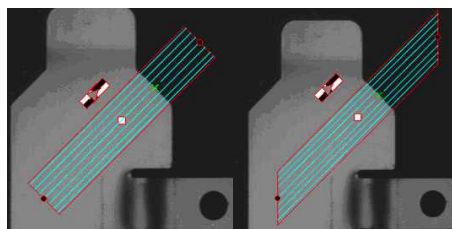
Prefiltering the image locally can reduce noise effects. Transverse (lengthwise) filtering averages several parallel lines when sampling the image. Longitudinal (crosswise) uniform filtering can also be applied to the resulting profile curve.



Thick point gauge for filtering

Transverse Filtering

Transverse filtering places parallel line segments in either a parallelogram or a rectangle (default). This behavior can be toggled. Parallelogram mode is faster than rectangular if the angle is close to 0° 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).

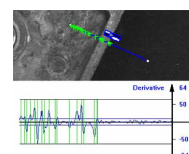
Low values indicate a weak edge, possibly corresponding to an unreliable or inaccurate measurement.

Tuning Point Measurement Parameters for unclear edges

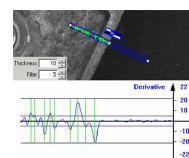
The EasyGauge default parameters and working modes are good for clear edges. More complex situations may need parameter tuning.

1. Set the gauge point location and tolerance.

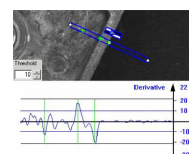
The center position and orientation are easy to decide based on a sample image or on coordinate considerations. The tolerance depends on the edge position variations. A larger tolerance increases the likelihood of hitting an edge, but it may be a false edge or extraneous feature.



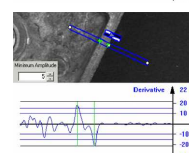
2. Decide whether noise reduction is required. Lay the gauge over the desired location and observe the profile curve and its derivative (play with the filtering parameters while looking at the plotted curve). The curve regularity gives an indication of the spread of the gray-level values. When these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



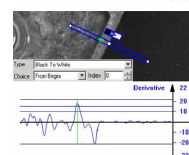
3. Set the threshold value to be low enough for useful parts of the peaks to cover enough pixels (to achieve better sub-pixel accuracy), but not lower than the ambient image noise.



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



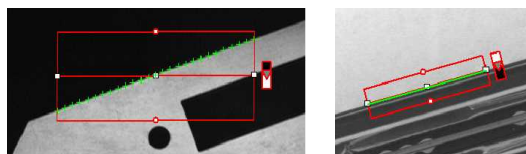
5. Choose whether all transition points are needed or just the most relevant. If all are required, they can be queried one after another. Otherwise, a point selection strategy should be chosen based on strength, order or transition polarity (black to white and/or conversely).



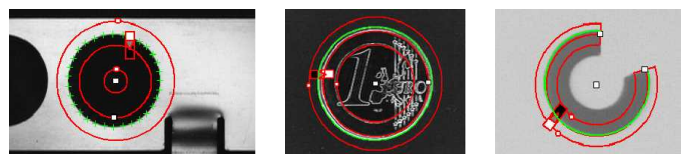
2.4. Find Shapes Using Geometric Models

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

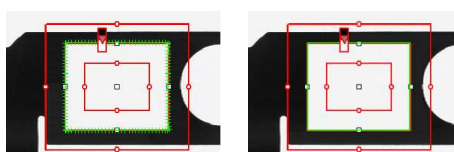
- **Line:** Measures position and orientation of straight edges.



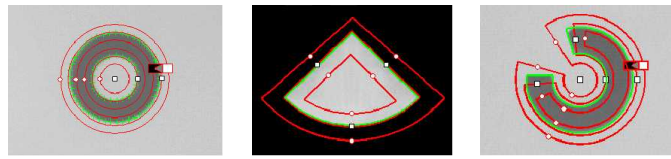
- **Circle:** Measures position and curvature of a circle or arc.



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



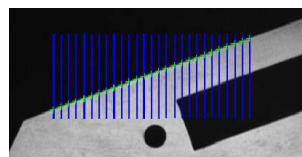
- **Wedge:** Measures position, orientation and size of a ring/ disk sector / curvilinear rectangle.



All gauge types share these common features:

- **Point sampling**

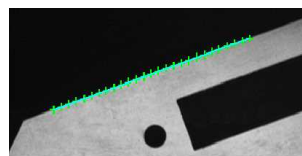
- Point gauges are placed along the edges and point measurement carried out at regularly spaced spots, which can be adjusted differently per side in rectangle and wedge gauges. All point measurement parameters and operating modes are available.
- [SamplingStep](#) sets the spacing of point location gauges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



Sampling paths and sampled points

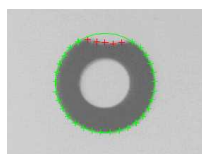
- **Model fitting**

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



- **Outlier rejection**

- After model fitting, some points will be too far away from the fitted model and may harm location accuracy. EasyGauge can tag them as outliers to be ignored using the [FilteringThreshold](#) property.
- The outlier elimination process can be repeated several times using [NumFilteringPasses](#). The number of valid sample points remaining after a model fitting operation is kept in [NumValidSamples](#). The average distance of these points to the fitted model is returned by [AverageDistance](#).



2.5. Gauge Manipulation: Draw, Drag, Plot, Group

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

Draw

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

Drag

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

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

Plot

EasyGauge can `Plot` gray-level values along the sampled paths and/or its derivative - useful for parameter tuning.

Point measurement gauges can plot after calling `Measure`.

Model fitting gauges can plot after calling `MeasureSample` with an index argument that lies between **0** and **GetNumSamples-1** (included).

To view the corresponding sampling path, use the method `Draw` with mode **EDrawingMode_SampledPath**.

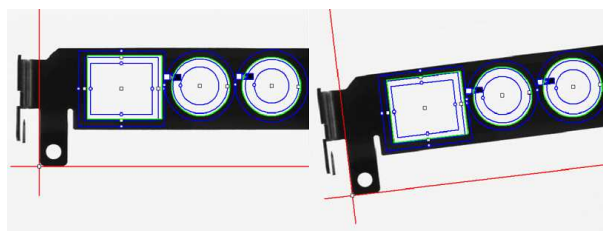
Group

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

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

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

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



2.6. Calibration and Transformation

Field-of-view calibration

<p>Calibration establishes the relationship between real-world point coordinates and image pixels. A simple calibration model computes faster, a repeatable part position is easier to locate.</p>	
<p>The Raw sensor coordinate system starts from upper left and extends rightwards and downwards. The range of abscissas is 0 to width-1 and the range of ordinates is 0 to height-1 where integer coordinate values correspond to pixel centers.</p>	
<p>The Centered sensor coordinate system starts at the center ($[(width-1)/2, (height-1)/2]$ in the Raw system) and extends rightwards and upwards.</p>	
<p>The real world 3D coordinates are defined in a 2D reference frame tied to a reference plane. The origin and direction of the axis are normally aligned with major features of the inspected parts.</p>	

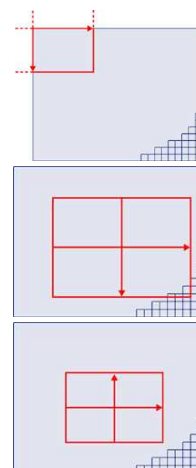
Before World-to-Sensor Transform

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

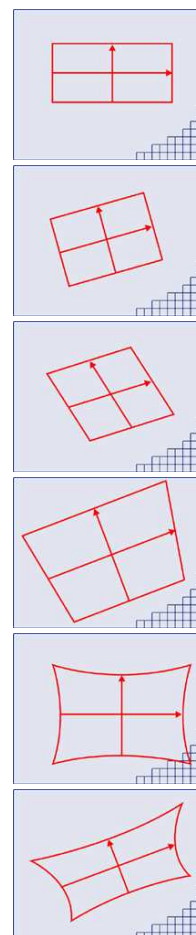
- adjust sweep frequency or scanning speed to avoid non-square pixels.
- adjust optical alignment to minimize perspective effect. The field of view should be parallel to the sensor plane.
- use long focal distances and good quality lenses to minimize Optical distortion.
- use appropriate scale factor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

Effects of World-to-Sensor Transform

- **No calibration.** World and sensor coordinates are identical.
- **Translated calibration:** The coordinate origin can be moved. World coordinates correspond to pixel units.
- **Isotropic scaling** (square pixels). A scale factor converts pixel values to physical measurements.



- **Anisotropic scaling** (non-square pixels). Uses two scale factors with pixel aspect ratio (X/Y) in the range $[-4/3, -3/4]$ (or $[3/4, 4/3]$). Pixels are always displayed as square, so the image appears stretched.
- **Scaled and skewed** (square pixels). Real-world axis aligns with rotated inspected part using translation, rotation and scaling.
- **Scaled and skewed** (non-square pixels). Distortion is apparent. Occurs when camera scan speed does not match pixel spacing.
- **Perspective distortion** causes further away objects to look smaller; lines remain straight but angles are not preserved.
- **Optical distortion** causes cushion or barrel appearance of rectangles.
- **Combined distortions** result in a complex, non linear, transform from real-world to sensor spaces.



2.7. Calibration Using EWorldShape

The EWorldShape object can calibrate the whole field of view (in given imaging conditions with fixed camera placement and lens magnification), if the optical setup is modified.

EWorldShape computes appropriate calibration coefficients and transforms measurement gauges that are tied to it.

It can set world-to-sensor transform parameters, perform conversions from and to either coordinate system, determine unknown calibration parameters, and save the parameters of a given transform for later reuse.

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

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

There are several ways to obtain the calibration coefficients:

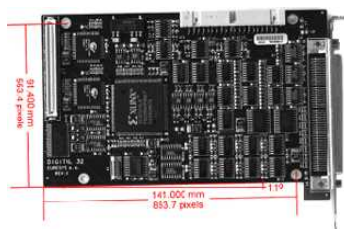
Estimate (feasible if no distortion correction is required and accuracy requirements are low)

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

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

To estimate a skew angle, apply this formula to two points on the X-axis in the world system:

$$\theta = \arctan \frac{j_1 - j_0}{i_1 - i_0}$$



Estimating scale factors and skew angle

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

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

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

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

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

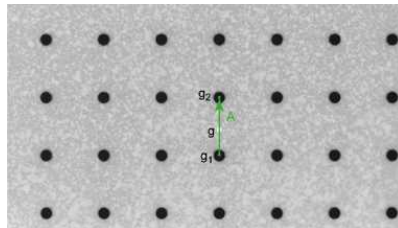
Analyze a Calibration target

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

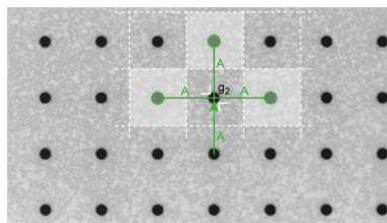
Open eVision relies on the use of a specific target holding a rectangular grid of symmetrical dots (of any shape) with no other object on the grid.

Dot Grid based calibration example

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



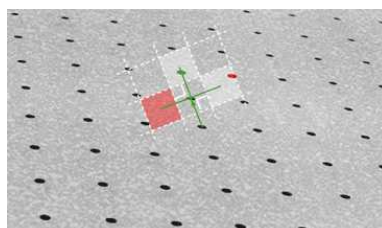
- b. Starting from the extremity of the reference segment (g_2), the algorithm determines 3 tolerance areas (white squares in the figure), in perpendicular directions. The tolerance areas are centered at a distance A (length of the reference segment) from (g_2). They are square, with a side-length of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



- c. The 3 perpendicular segments are the references of the next iterative searches. The algorithm goes back to step 2.
5. Call [Calibrate](#).

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

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



TIP

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

2.8. Advanced Features

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

Sensor width and height

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

Field-of-view width and height

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

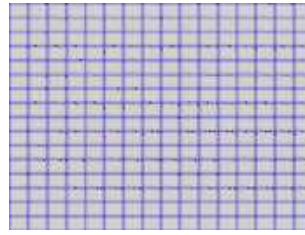
$$\begin{aligned} f\text{-}o\text{-}v \text{ width} &= \text{pixel width} * \text{sensor width} \\ f\text{-}o\text{-}v \text{ height} &= \text{pixel height} * \text{sensor height} \end{aligned}$$

or

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

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

Ratio



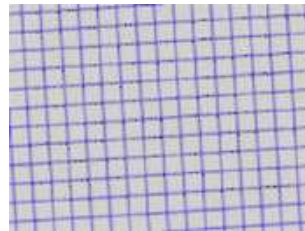
Anisotropic aspect ratio

Center abscissa and ordinate

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

Skew angle

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



Skew angle

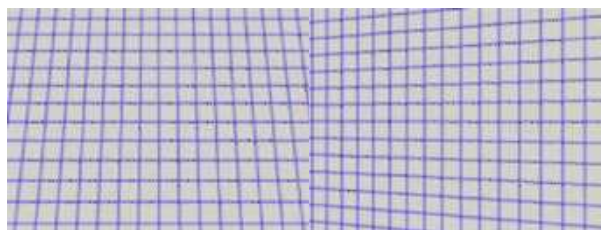


NOTE

When the pixels are not square, the [EWorldShape](#) object can convert the angle between the world and sensor spaces.

X and Y tilt angles

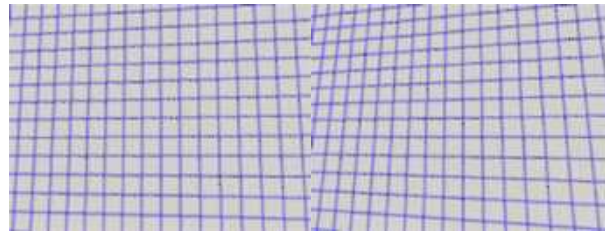
The **X** and **Y** tilt angles describe the viewing plane direction. They correspond to the required rotations around X and Y axis that bring the Z axis parallel to the optical axis.



Tilt X and tilt Y angles

Perspective strength

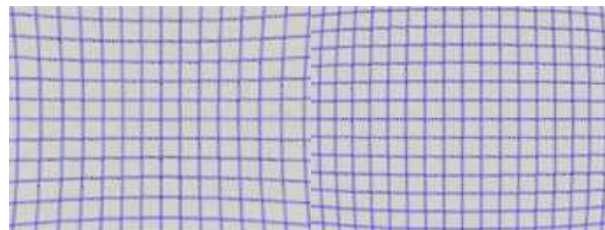
The **perspective strength** gives a relative measure of the perspective effect. The shorter the focal length, the larger the value.



Weak and strong perspective

Distortion strength

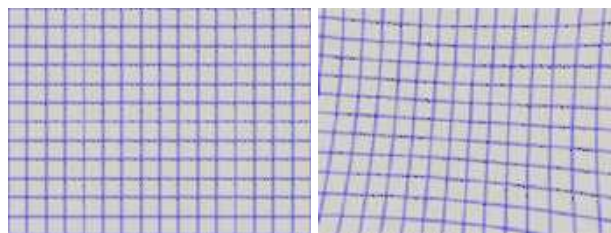
Distortion strength gives a relative measure of radial distortion in the image corners, that is the ratio of image diagonal length with and without distortion.



Positive and negative distortion

Calibration mode, expressed as a combination of options, can be accessed via [CalibrationModes](#).

Effect of the Calibration Coefficients



No calibration coefficient: All coefficients combined.

2.9. Unwarp an Image

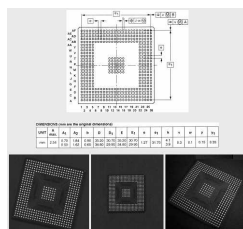
An **EWorldShape** object manages a field-of-view calibration context. Such an object is able to represent the relationship between world coordinates (physical units) and sensor coordinates (pixels), and account for the distortions inherent in the image formation process.

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

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

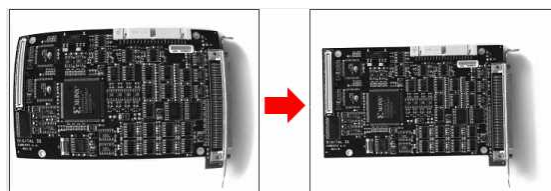
The goal of calibration is twofold:

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



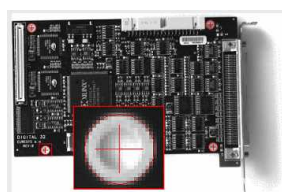
Single model versus multiple viewing conditions

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



Removal of image distortion

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

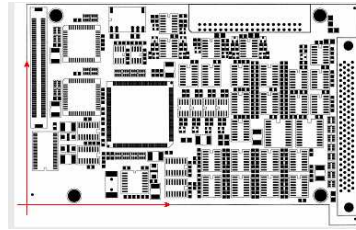


Feature point in sensor space

The world coordinates describe the location of points on the inspected item are expressed in an appropriate length measurement unit.

The world coordinates are actual dimensions, usually gathered from design drawings or by mechanical measurements.

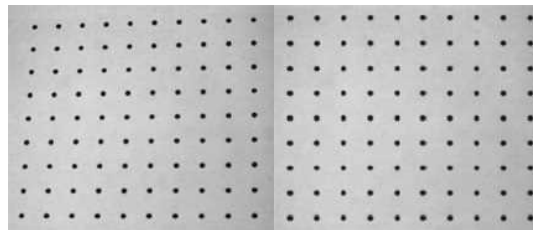
They require a reference frame to be defined.



Reference frame in world space

Unwarp

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

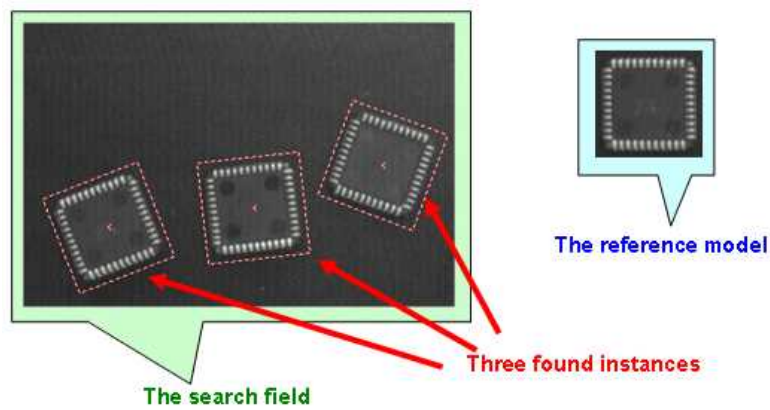


Distorted vs. Unwarped image

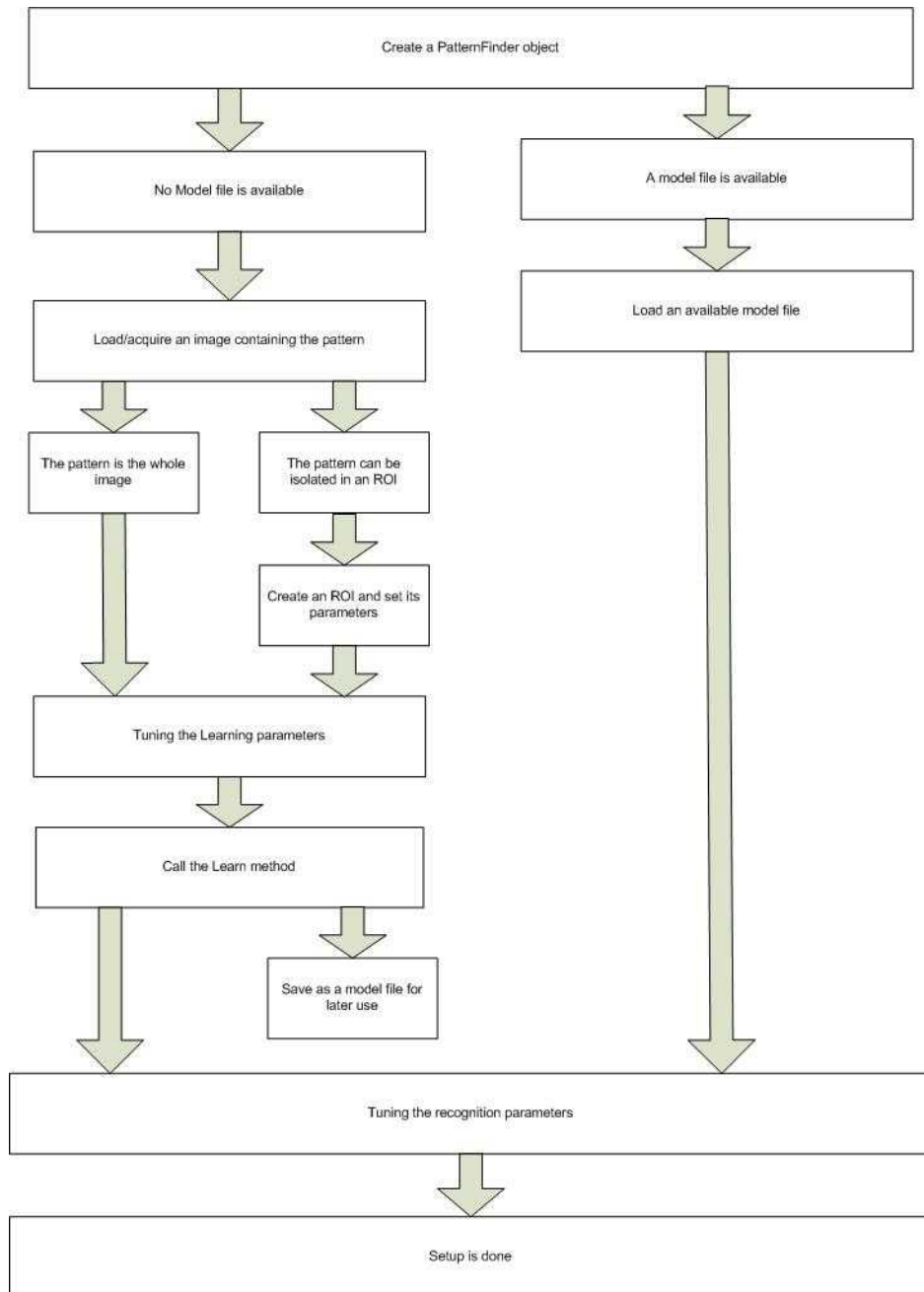
3. EasyFind - Matching Geometric Patterns

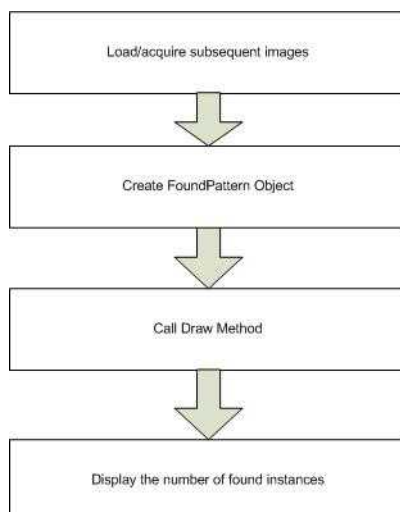
3.1. Workflow

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



Workflow

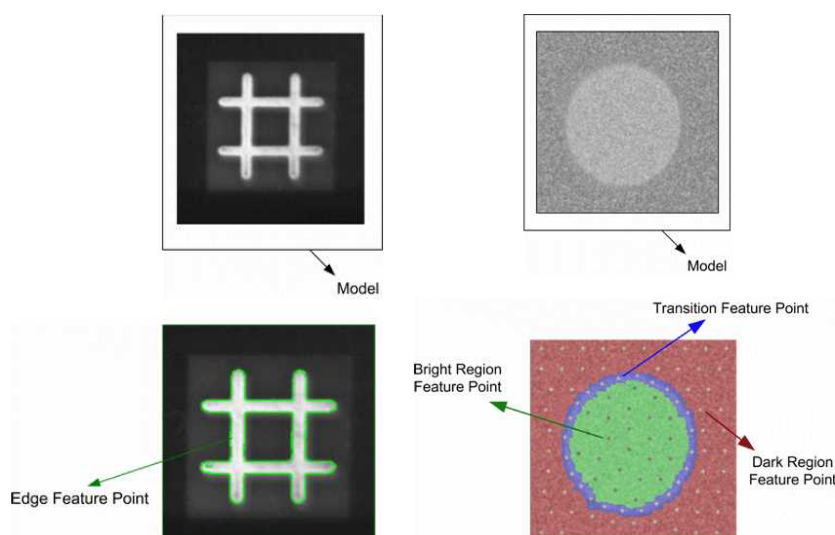




Feature points definition

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

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

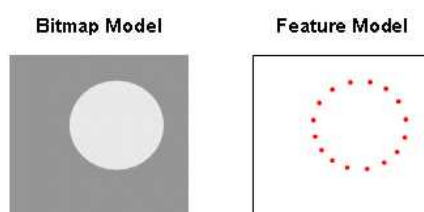


- **Edge feature points:** an abrupt change of gray level between two regions indicates an edge at this location in the search fields.
- **Transition feature points:** a smooth change of gray level between two regions indicates a transition area in their neighborhood (represented by dots in the blue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

3.2. 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.

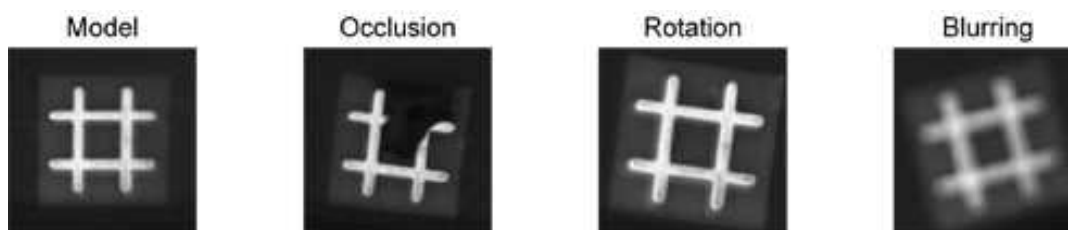
Consistent Edges

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:

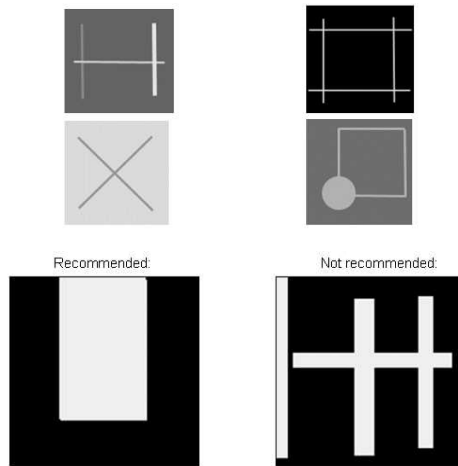
- `PointByPointNormal`: if points share the same contrast polarity.
- `PointbyPointInverse`: if points exhibit opposite contrast polarity.
- `PointByPointAny`: regardless of their respective contrast polarity.



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.

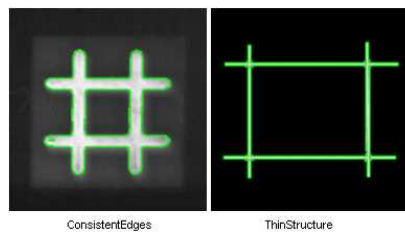
Good for models containing thin elements.



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.

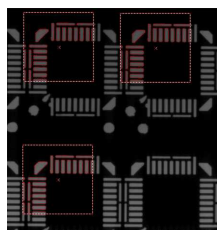


3.3. Finding Process

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.

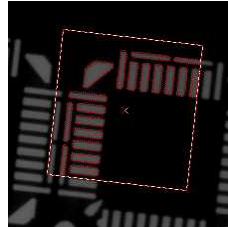
Maximum number of expected instances

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)

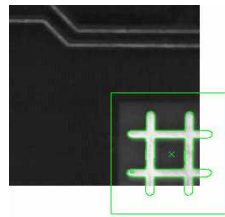
Ranges have a bias and a tolerance. For instance, for an angle bias of 20° and an angle tolerance of 5° , EasyFind returns instances with an angle between 15° and 25° with respect to the learned model ($20^\circ \pm 5^\circ$).



3.4. Advanced Features

Find partial patterns

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



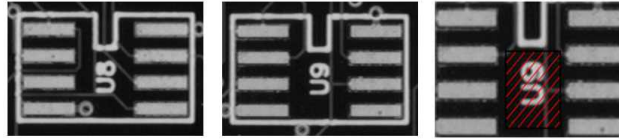
Tune Parameters

These parameters can be tuned for all models:

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



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



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

4. EasyMatch - Matching Area Patterns

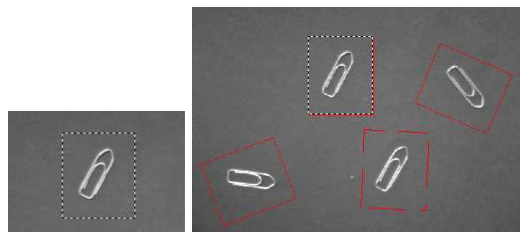
4.1. Workflow

EasyMatch

Reference

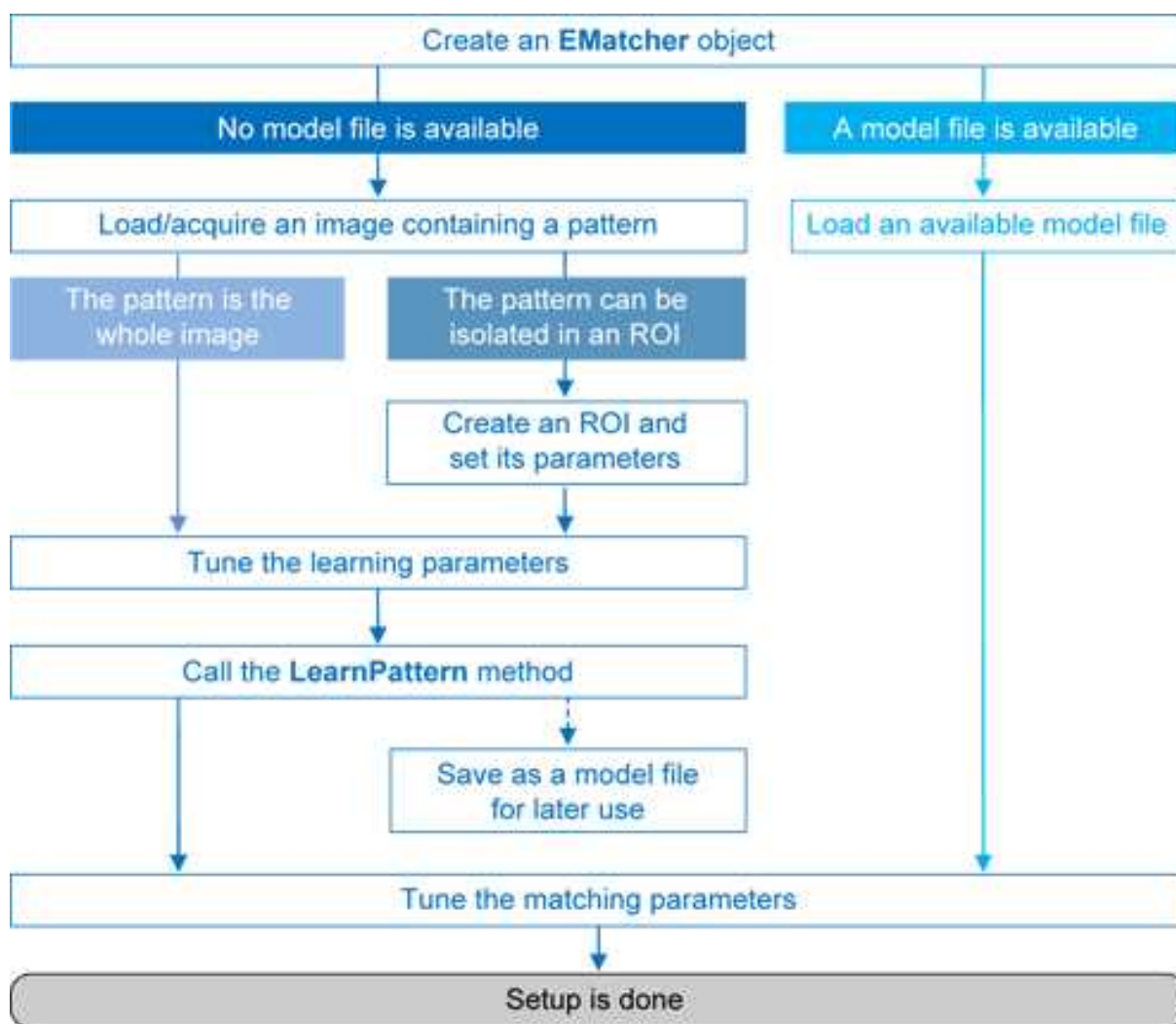
EasyMatch learns a pattern and finds exact matches:

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

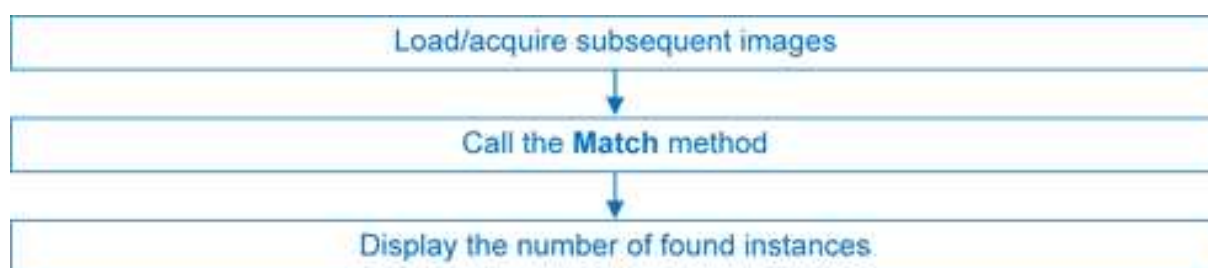


Learning and Matching a pattern

Learning workflow



Matching workflow



4.2. Learning Process

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

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

Best pattern characteristics

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

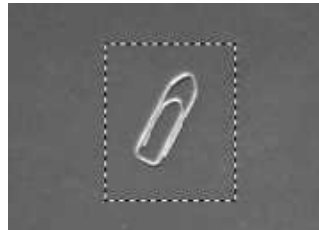
Customize Parameters

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

- **DontCareThreshold:** If don't care areas are required, the corresponding pixels must hold a value below the **DontCareThreshold**.
If all the background can be ignored, merely adjusting the **DontCareThreshold** to the right thresholding value can do.
Otherwise, when the don't care area is unrelated to the threshold pattern image, the **DontCareThreshold** should be set to 1 and all pixels belonging to the don't care area should be set to black (value 0).

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

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



Learning a pattern

4.3. Matching Process

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

- **Match**: receives the target image/ROI as its argument and locates the desired occurrences of the pattern. You can pass an arbitrary shaped region of interest (**ERegion**) to ignore the pixels located outside of this region.

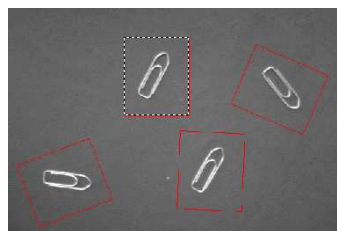
You can set these parameters:

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

The following functions return the result of the matching:

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

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



Matching a pattern

4.4. Advanced Features

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

- Learning time:

- Optimize number of searches: Searching all positions takes too long, so a sequence of searches is performed at various scales (reductions). The coarsest reduction is quick and approximate. Subsequent reductions work in a close neighborhood to improve location, drastically reducing the number of positions to be tried. The location accuracy is given by $2K$, where K is the reduction number.
- [MinReducedArea](#). Indicates how small the pattern can be made for rough location.
- Matching time:
 - Correlation mode (way to compare the pattern and the image): [CorrelationMode](#). **Can be standard, offset-normalized, gain-normalized and fully normalized:** the correlation is computed on continuous tone values. Normalization copes with variable light conditions, automatically adjusting the contrast and/or intensity of the pattern before comparison.
 - Contrast mode (way to deal with contrast inversions): [ContrastMode](#). Lighting effects can cause an object to appear with inverted contrast, you can choose whether to keep inverted instances or not, and whether to match positive occurrences only, negative occurrences only or both.
 - Maximum positions (expected number of matches): [MaxPositions](#), [MaxInitialPositions](#). You can compel EasyMatch to consider more instances than needed at the coarse stage using the [MaxInitialPositions](#) parameter (this number is progressively reduced to reach [MaxPositions](#) in the final stage).
 - Minimum score (under which match is considered as false and is discarded): [MinScore](#), [InitialMinScore](#).
 - Sub-pixel accuracy: [Interpolate](#). The accuracy with which the pattern is measured can be chosen (the less accurate, the faster). By default, the position parameters for each degree of freedom are computed with a precision of a pixel. Lower precision can be enforced. One tenth-of-a-pixel accuracy can be achieved.
 - Number of reduction steps: [FinalReduction](#). Can speed up matching when coarse location is sufficient, range [0...[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.

5. EChecker2 - Validating Golden Templates

5.1. EChecker2

- EChecker2 is a tool that allows to inspect an image using a Golden Template validation. It works in 2 steps:
 - a. The *Model Creation* involves pre-processing a set of reference images to compute a model. You can create the model once and archive the results in a Golden Template model for later use.
 - b. The *Inspection* involves processing an image and checking its quality using the previously computed model.These 2 operations are totally independent and can even be programmed in separate applications.
- EChecker2 is part of the EasyObject library.
- The following sections present the relevant API functions for use in the training and inspection steps.

EChecker2 vs. EChecker

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

5.2. Creating a Model

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

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

Initialization

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



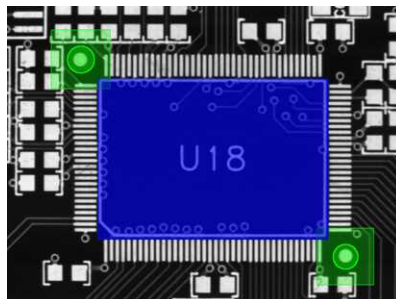
TIP

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



NOTE

- As per Open eVision 2.15, you can only use either 1 or 2 regions.
 - The possibility to use more regions will be added in the future.
- An **ERegion** is defined to delimit the area to be inspected. This area should only include pixels of the rigid part (that moves with the fiducial marks), and not the background.
- To perform the initialization, use the method `EChecker2::Initialize`.



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

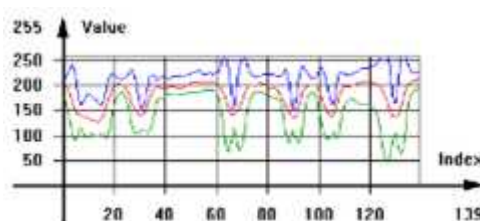
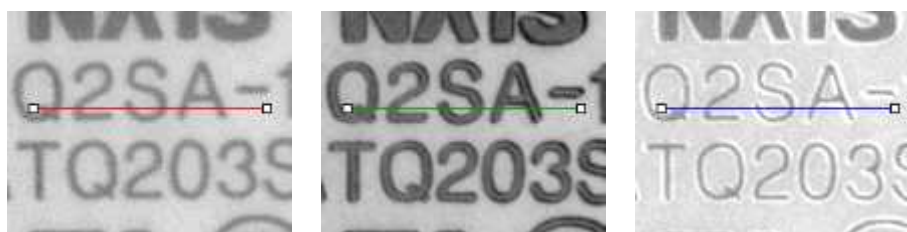
Training

- After the initialization, the main training phase begins.
 - All the training images are processed and are averaged using statistical training (see below).
 - The training uses realignment to deal with displacement of the inspected part in the field of view.
 - The training uses gray-level normalization to deal with global illumination changes.
- Ideally, use 16 images or more in training to create the low and high threshold images that serve as the basis of the inspection process.

- To perform the training:
 - Use the method `EChecker2::Train` with class instances.
 - Use the method `EChecker2::TrainFromImageFiles` with a list of image files.

Statistical training

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



Accepted gray-level ranges

Model creation parameters

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

- Set `FiducialHorizontalTolerance` and `FiducialVerticalTolerance` to adjust the search distance for the fiducials from the reference position (after realignment).
 - Default: 30 pixels.
- Set `InspectionTolerance` to adjust the acceptance ranges during the inspection.
 - Use higher values to make the inspection process more tolerant to noise and/or texture.



NOTE

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

Model serialization

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

5.3. Inspecting an Image

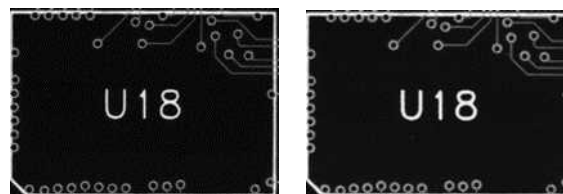
- Use the inspection on an image to assess it towards the trained model.

The process is straightforward:

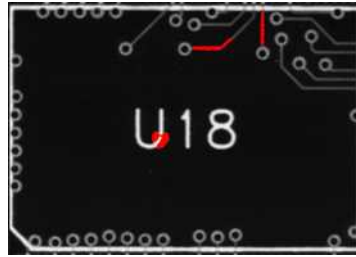
- a. The sample image is realigned with the model.
- b. The gray-level is normalized.
- c. This gray-level is combined with the high and low threshold images to populate an `ECodedImage2`.
- d. The computed blobs are made of pixels that fall out of the range defined by the threshold images and thus represent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



The realigned Image



The low and high threshold images



Detected defects after inspection

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

PART IV
TEXT AND CODE READING
TOOLS

1. EasyBarCode - Reading Bar Codes

1.1. Reading Bar Codes

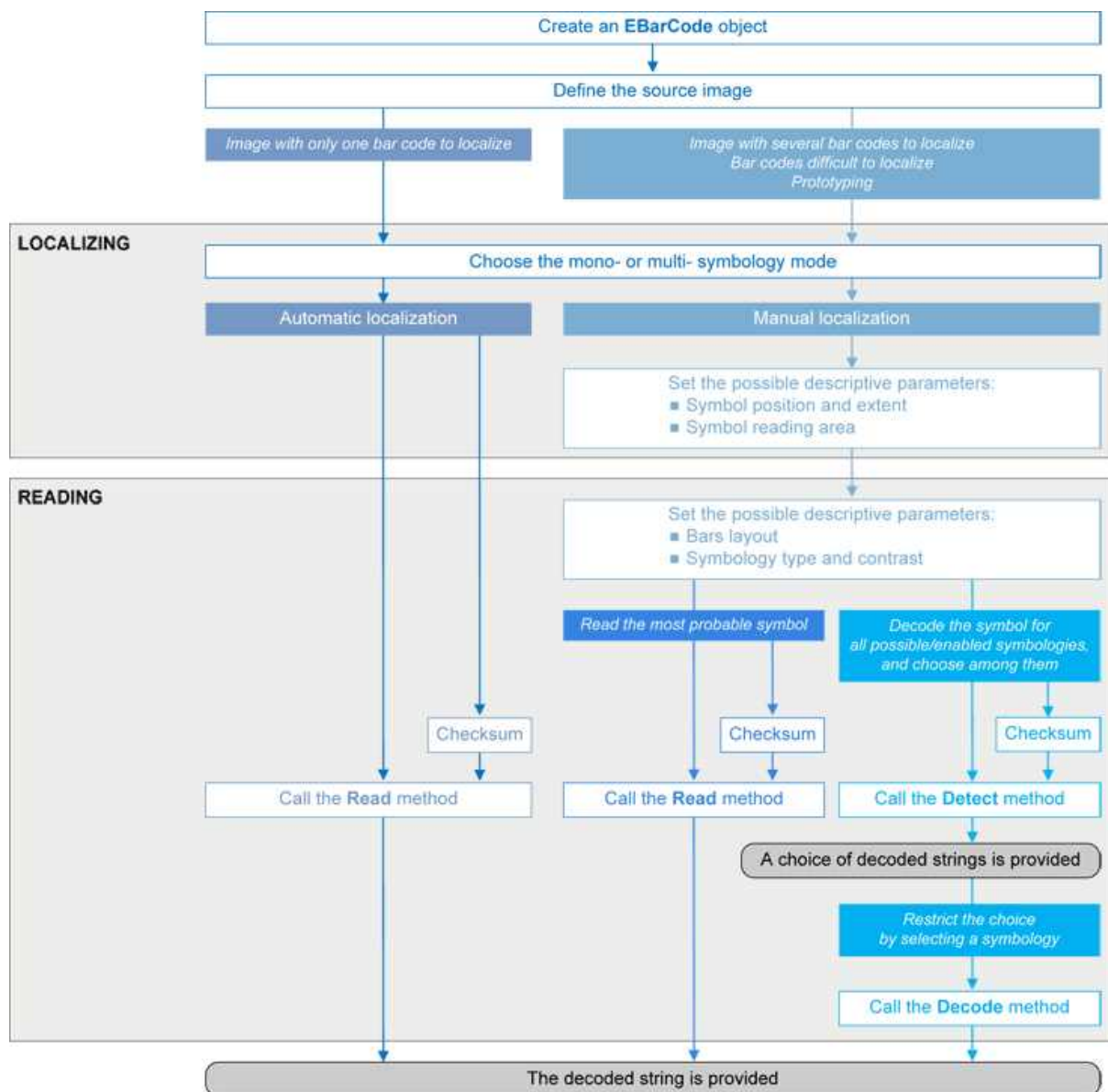
[Reference](#) | [Code Snippets](#)



Bar code (EAN 13 symbology)

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

Workflow



Bar code definition

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

- The bar code may be black ink on white background or inversely 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.
- Bars and spaces 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:

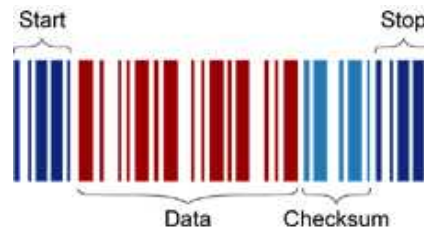
- ADS Anker
- Binary code
- Code 11
- Code 13
- Code 32
- Code 39 Extended (a super-set of Code 39)
- Code 39 Reduced (a subset of Code 39)
- Code 93
- Code 93 Extended
- Code 412 SEMI
- Code 2/5 3 Bars Datalogic
- Code 2/5 3 Bars Matrix
- Code 2/5 5 Bars IATA
- Code 2/5 5 Bars Industry
- Code 2/5 5 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

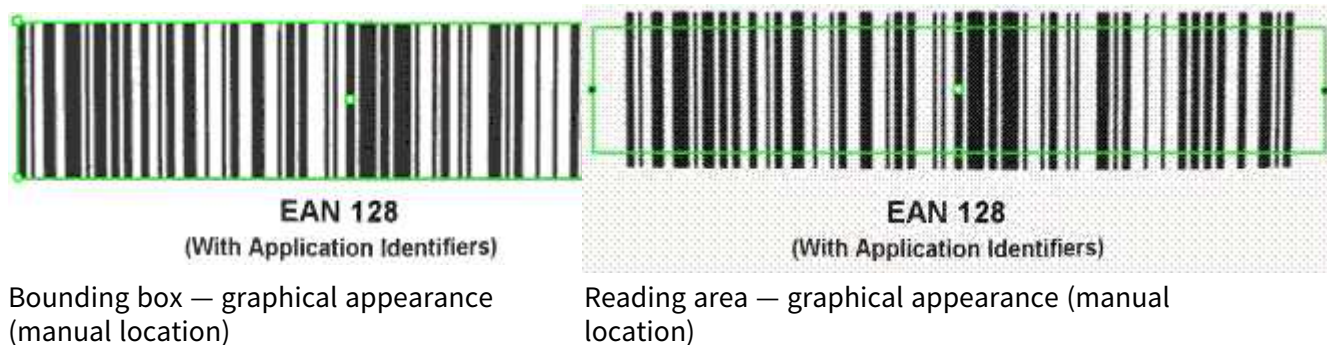
Note: When the bar code contains `\0x00` characters, the `std::string::c_str` method should not be used (since C-strings are terminated by the `\0x00` character). An iterator over the characters should be used instead of a C-string.

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!).



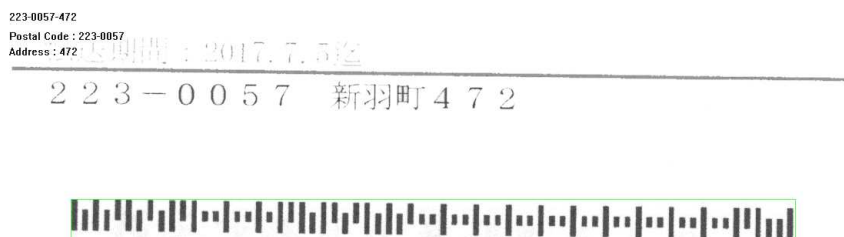
Read all interpretations (multi-symbology mode)

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.

1.2. Reading Mail Bar Codes

Reference | Code Snippets



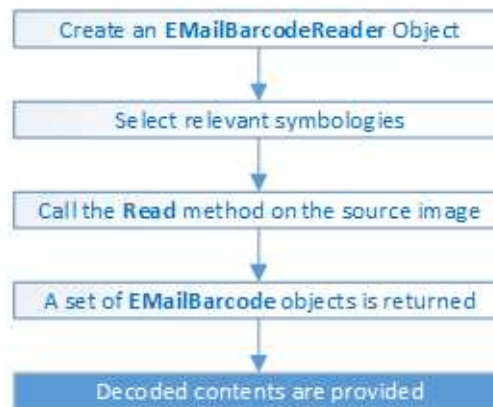
Mail bar code example

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



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.



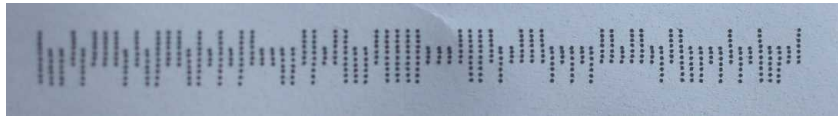
US Intelligent Mail bar code with highlighted position and decoded information

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.

2. EasyMatrixCode - Reading Matrix Codes

2.1. EasyMatrixCode vs EasyMatrixCode2

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

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.

2.2. EasyMatrixCode

Specifications

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



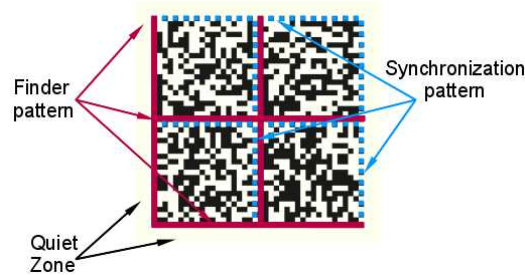
ECC 200, 26x26 cells data matrix code (left) and finder pattern (right)

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)..

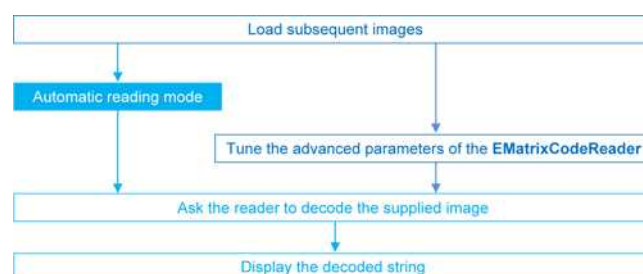


NOTE

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

Workflow

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



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:

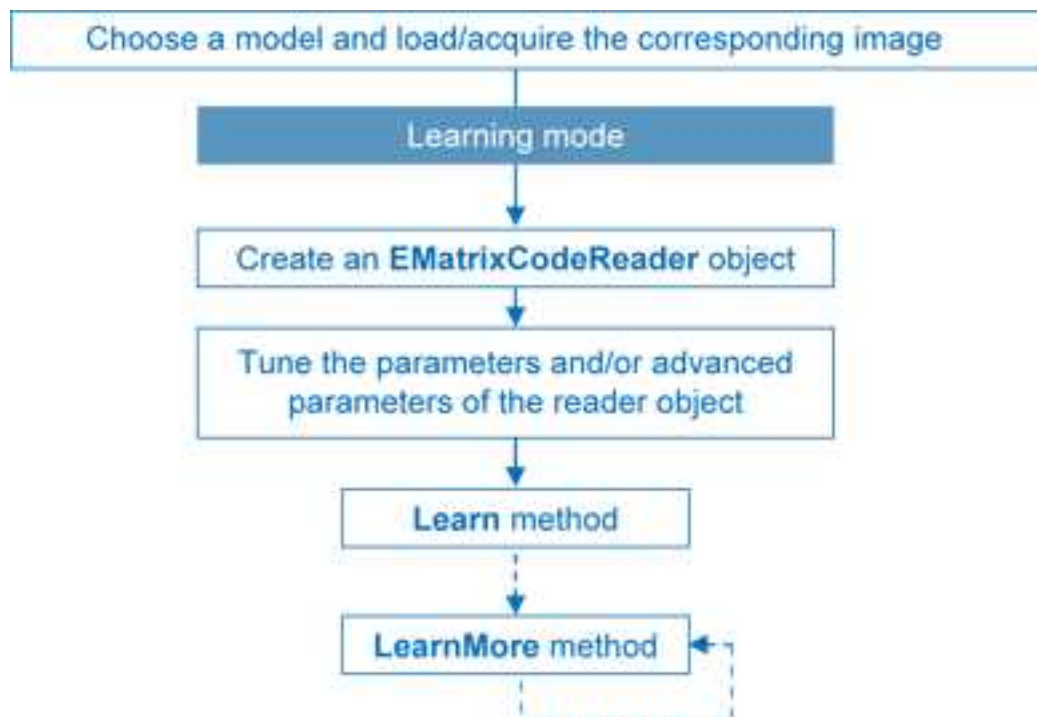
- Its decoded string,
- Its position in the image,
- Its logical size,
- Its encoding type,
- Its grading results,
- Methods to draw the data matrix code on the source image.

Learning a Matrix Code

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

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

Workflow



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 [GetIso545GradingParameters](#), [GetIso2958GradingParameters](#) 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:

$$]d2 [GS1]{Id1} {Value1} [GS1]{Id2} {Value2} \dots$$

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 with 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.



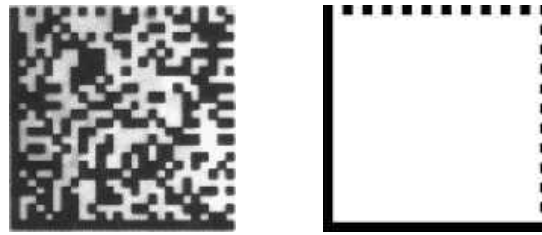
TIP

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

2.3. 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).

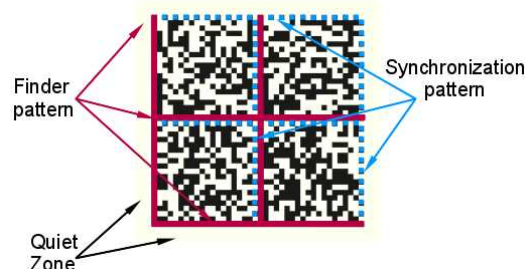


NOTE

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.



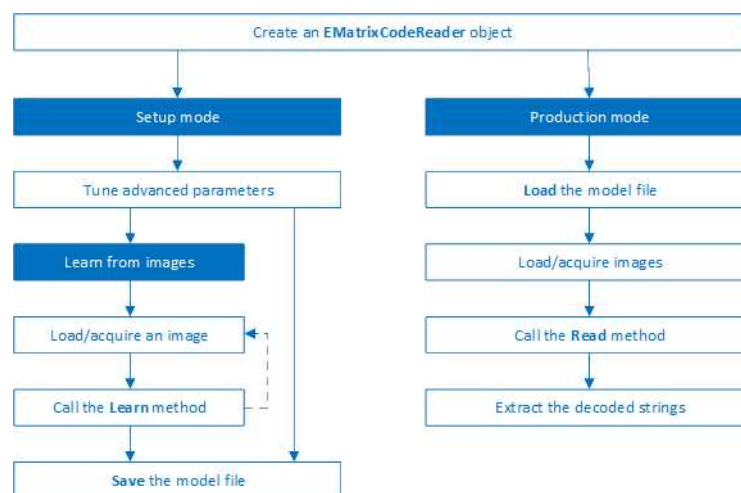
- A data matrix code is characterized by:
 - Its **logical size** (number of cells).
 - Its **encoding type**: ECC 000 (odd symbol sizes, deprecated) or ECC 200 (even symbol sizes)..

**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 [GetIso545GradingParameters](#), [GetIso2958GradingParameters](#) 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:

$$]d2 [GS1]]{Id1} {Value1} [GS1]]{Id2} {Value2} \dots$$

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 with 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, it 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](#).

3. EasyQRCode - Reading QR Codes

3.1. Workflow

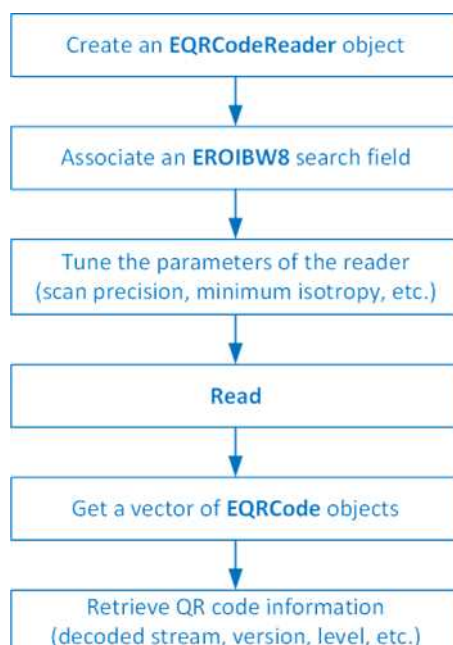
EasyQRCode



EasyQRCode detects QR (Quick Response) codes in an image, decodes them, and returns their data.

Error detection and correction algorithms ensure that poorly-printed or distorted QR codes can still be read correctly.

Workflow



3.2. QR Codes Specifications

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).

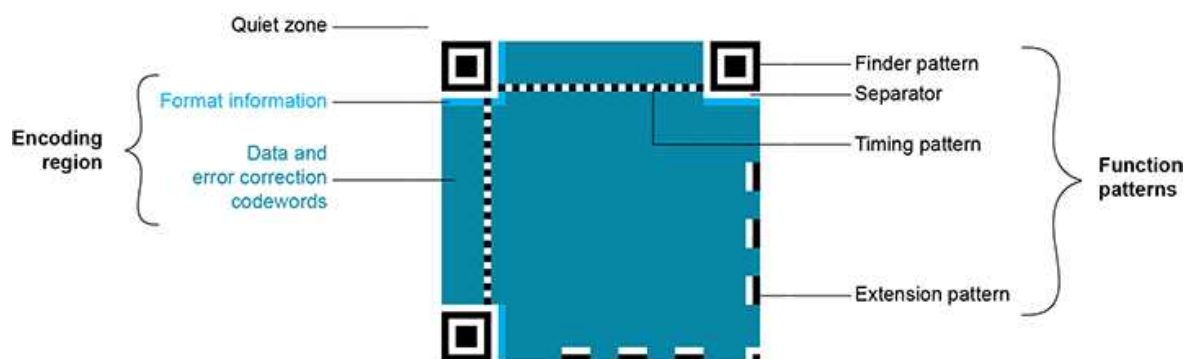
QR code structure

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.

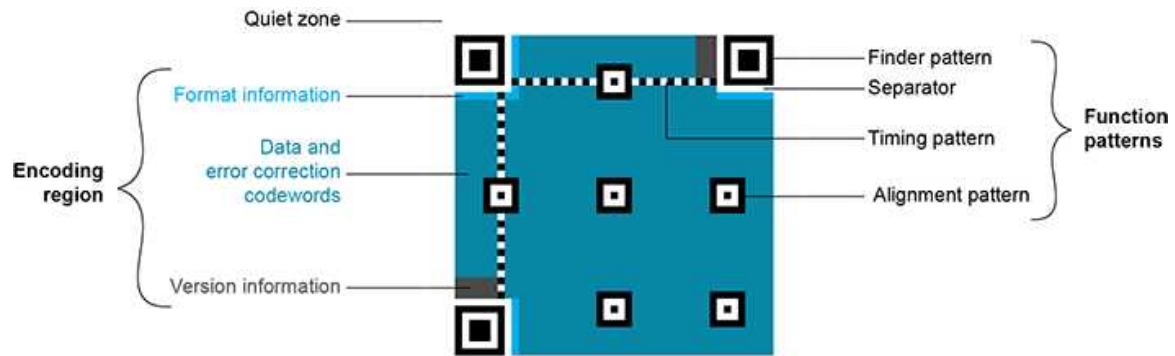
A QR code must be structured with the following elements:

- *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

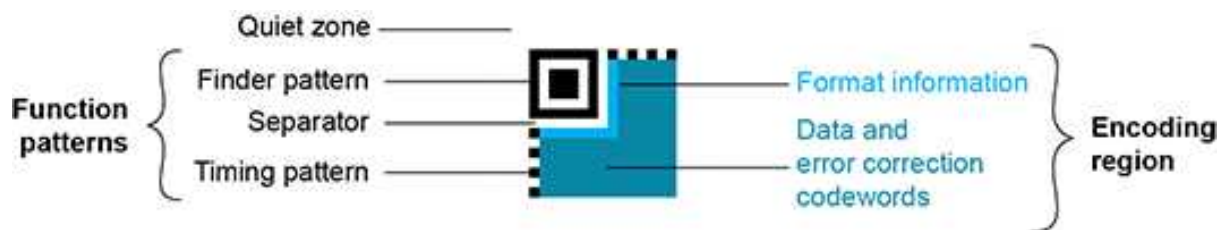
Variants of this structure exist, according to the model, format, or version of the QR code. For instance, model 1 QR codes do not feature alignment patterns but extension patterns. Micro QR codes include only one finder pattern, and no alignment pattern.



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. (ECI is not available for Micro QR code symbols.)
- *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 ["Reading QR Codes" on page 170](#) 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*: 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*: 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)

For Micro QR code symbols, the available error correction levels depend on the version:

- M1 has only error detection
- M2 and M3 support L and M levels
- M4 supports L, M and Q levels

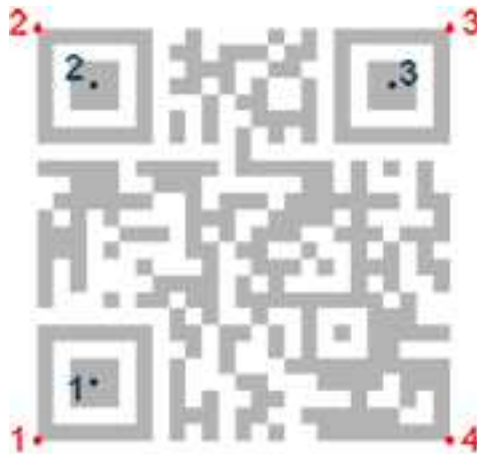
QR code geometry

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).
- It contains the coordinates of the **finder pattern centers** (bottom left, top left, top right).
- For a Micro QR code symbol, the coordinates for a single **finder pattern center** (link) are returned.

EasyQRCode uses a float coordinate system and the origin (0.0, 0.0) is the top left corner of the top left pixel of the image.



QR code geometry

Read a QR code

Reading a QR code returns information about every QR codes found in the given search field (see "Reading QR Codes" on page 170).

3.3. Reading QR Codes

Read 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 QR code reader uses the **gravity center** of the **QR code geometries** to sort the QR code candidates in line then columns order starting from the top left corner of the image.
5. The QR code reader decodes the QR candidates and returns the **QR code: model, version, level, geometry** and the **decoded data** as described below.
6. 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**: By default, the QR code reader searches for QR codes of model 1 and 2, and all versions.
 - You can shorten the process by specifying the QR code **model(s)** and a range of **versions** (from 1 (**minimum**) to 40 (**maximum**)) to search for.
 - By default, the QR code reader does not search for Micro QR code symbols.

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 is available).
- 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 `0xEFBB` escape character.
- See the code snippet [Retrieving Information of a QR Code](#).

The `decoded stream` class consists of:

- A coding mode (`basic`, `ECI`, `FNC/GS` or `FNC/AIM`).
- An application indicator (if the coding mode is `FNC/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 `-`).
- 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 \(Advanced\)](#).

Computing the print quality

- The print quality indicators as defined by ISO 15415 and ISO/IEC TR 29158 (formerly known as AIM DPM-1-2006) are computed during the `Read` operation, but only if `EQRCodeReader::ComputeGrading` is set to `TRUE`.
 - The print quality is not yet supported for Micro QR code models.
- Use the `EQRCode::GetIso545GradingParameters` and `EQRCode::GetIso2958GradingParameters` methods to retrieve the grades.
- Using the grading:
 - For more accurate results, it requires modules to be at least 10 pixels in width.
 - It requires a 1-module quiet zone for grading.
 - It evaluates the `Fixed Pattern Damage` with a 4-module quiet zone around the finder patterns. If this condition is not met, the `Fixed Pattern Parameter Grade` is returned as `-`. This result affects the overall symbol grade.
 - The `Version Additional Parameter` is returned as `-` when it is not applicable. In this case, this result is ignored in the overall symbol grade.
 - The implementation follows closely the standard but the grades also depend on the decoding algorithm. So the results may slightly differ according to the Open eVision version.
- The print quality computation is not yet available for Micro QR codes.

Multithreading

The `Read` method supports multithreading.

- Multithreading splits the load between the detection methods (such as `AdaptiveThreshold` and `Gradient`) and decodes multiple candidates in parallel. This is useful when there are several codes in the image.

4. EasyOCR - Reading Texts

4.1. Workflow

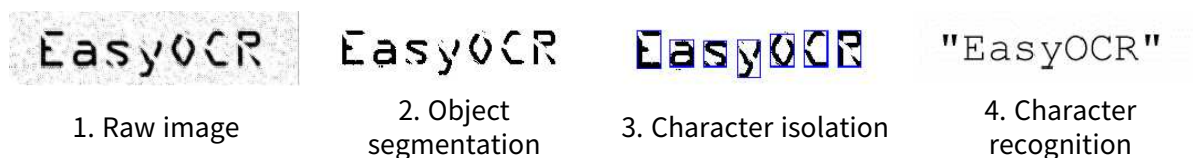
[Reference](#) | [Code Snippets](#)

EasyOCR

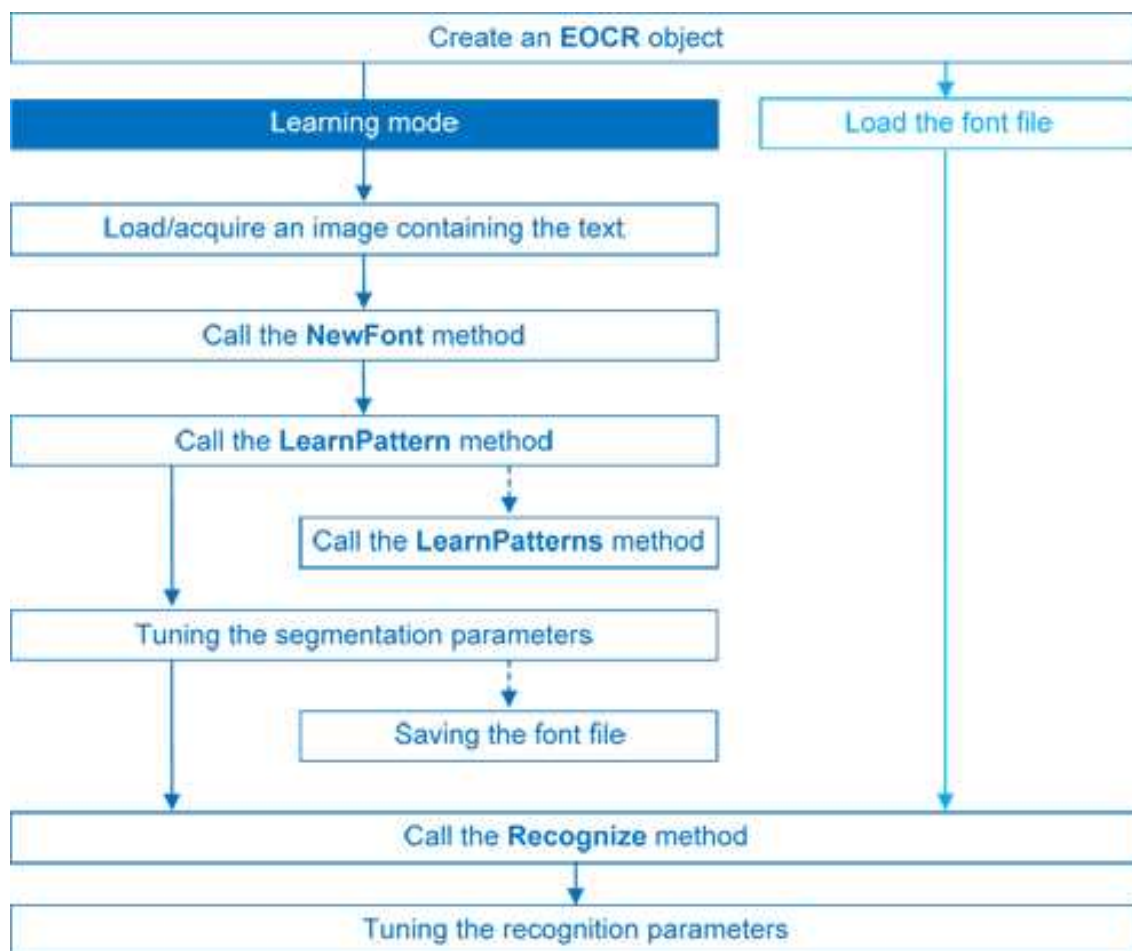
EasyOCR optical character recognition library reads short texts (such as serial numbers, part numbers and dates).

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

There are 4 steps to recognizing characters:



Workflow



4.2. Learning Process

You can learn characters to create a font file if required.

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

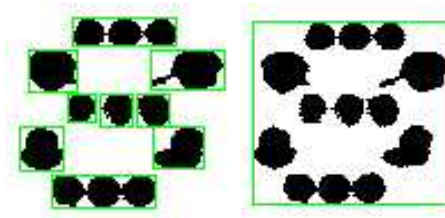
Font files are created as follows:

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

4.3. Segmenting

Segmenting

- 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](#).



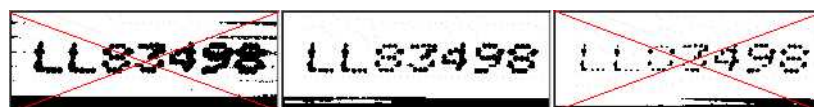
Character segmentation by blob grouping

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

4.4. Recognition

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.

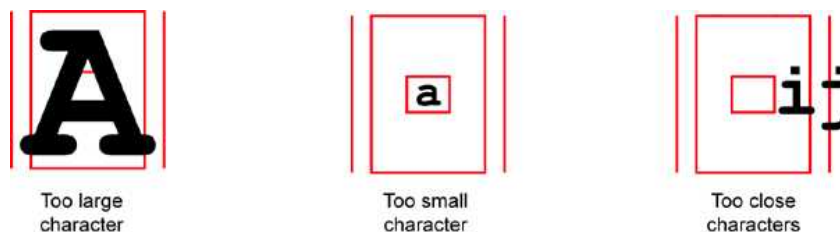
Steps 2 to 4 can be repeated at will to process other images or ROIs. The **Recognize** method can be used as well.

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

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

Recognition parameters

- **MaxCharWidth, MaxCharHeight**: if a blob does not fit within a rectangle with these dimensions, it is not considered as a possible character (too large) and is discarded. Furthermore, if several blobs fit in a rectangle with these dimensions, they are grouped together, forming a single character. The outer rectangle size should be chosen such that it can contain the largest character from the font, enlarged by a small safety margin.
- **MinCharWidth, MinCharHeight**: if a blob or a group of blobs does fit in a rectangle with these dimensions, it is not considered as a possible character (too small) and is discarded. The inner rectangle size should be chosen such that it is contained in the smallest character from the font, shrunk by a small safety margin.
- **RemoveNarrowOrFlat**: Small characters are discarded if they are narrow **or** flat. By default they are discarded when they are both narrow **and** flat.
- **CharSpacing**: if two blobs are separated by a vertical gap wider than this value, they are considered to belong to different characters. This feature is useful to avoid the grouping of thin characters that would fit in the outer rectangle. Its value should be set to the width of the smallest gap between adjacent letters. If it is set to a large value (larger than **MaxCharWidth**), it has no effect.
- **CutLargeChars**: when a blob or grouping of blobs is larger than **MaxCharWidth**, it is discarded. When enabled, the blob is split into as many parts as necessary to fit and the amount of white space to be inserted between the split blobs is set by **RelativeSpacing**. This is an attempt to separate touching characters.
- **RelativeSpacing**: when the **CutLargeChars** mode is enabled, setting this value allows specifying the amount of white space that should be inserted between the split parts of the blobs.



Invalid recognition settings

Advanced tuning

These recognition parameters can be tuned to optimize recognition:

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

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

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

ESegmentationMode, character isolation mode defines how characters are isolated:

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

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

5. EasyOCR2 - Reading Texts (Improved)

5.1. Workflow

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

EasyOCR2

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

It can use 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.

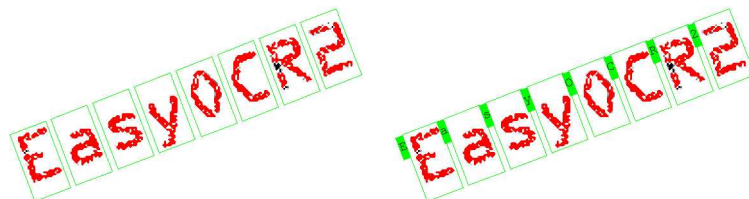
You can perform the recognition in two different ways:

- Use a pre-trained classifier powered by Deep Learning technologies.
- Or use a character database built from sample images or from a TrueType font (.ttf) file.

Text recognition with EasyOCR2 follows four phases:



Input image (left) and image segmentation (right)



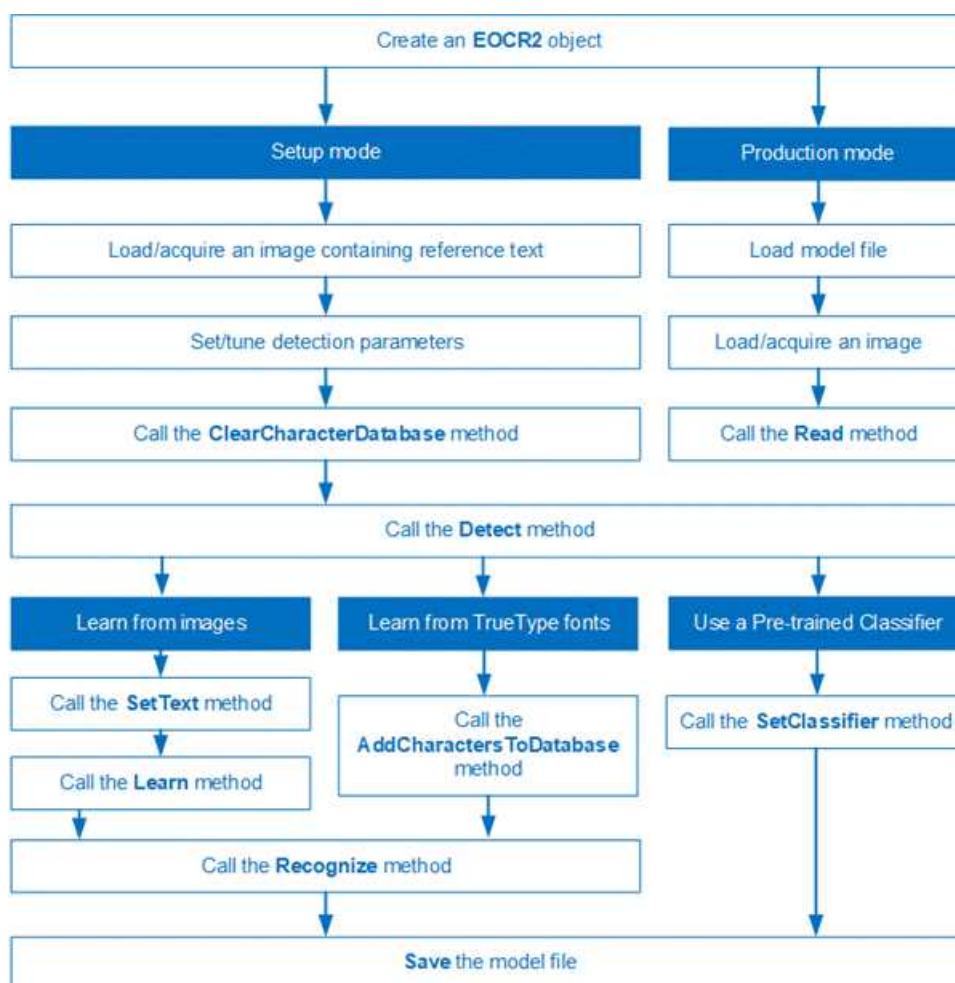
Fitting textboxes (left) and recognition (right)

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
 - Complex text topologies
- When TrueType font files that match the text to be read are available, EasyOCR2 allows the user to use those font files directly for recognition, while EasyOCR does not.
 - EasyOCR2 can perform recognition without any preliminary training using the provided pre-trained classifiers.

Workflow



5.2. Detection

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 as parts 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.
 - If the topology is not required, EasyOCR2 fits boxes to the blobs as best as it can.
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.

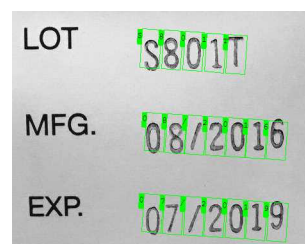
See example in code snippet: [Detecting Characters](#)

FIXED WIDTH

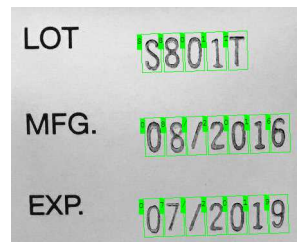
An example of a fixed-width font, processed with the `detectionMethod` `'EOCR2DetectionMethod_FixedWidth'`

PROPORTIONAL

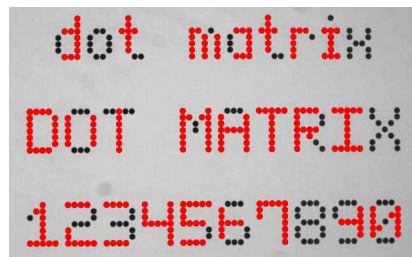
An example of a proportional font, processed with the `detectionMethod` `'EOCR2DetectionMethod_Proportional'`



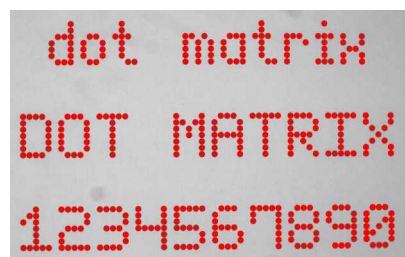
The text angle estimate for this image is slightly off when `NumDetectionPasses=>`



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 `EnableTopology` indicates if the parameter `Topology` is required. In simple cases, the topology may not be necessary.
- 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 parameter `SegmentationMethod` selects the algorithm used for segmentation.
 - The setting `EOCR2SegmentationMethod_Global` is faster and suited for clear background.
 - The setting `EOCR2SegmentationMethod_Local` (default) is more complex and suited for non uniform background.
- 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:

$$\text{minArea} = \text{CharsMaxFragmentation} * \text{CharsHeight} * \text{min}(\text{CharsWidthRange})$$

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

Advanced parameters for local segmentation (optional):

- 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 global segmentation (optional):

- The parameter `EnableSecondPassGlobalSegmentation` performs the segmentation twice during the first pass to have more accurate results when the text background is not just plain.

Advanced parameters for detection (optional)

- If the topology is required, 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 lower bound of this parameter is also used when the topology is not required.
- The parameter `CharsWidthBias` biases the optimization toward wider or narrower character boxes.
- The parameter `CharsSpacingBias` biases the optimization toward smaller or larger spacing between characters boxes.

Advanced parameters, specific for the setting `EOCR2DetectionMethod_Proportional`:

- The parameter `EnableCutLargeCharacter` tells if the detection should try to split too wide segmented blobs into different characters.
 - This parameter is also used when the topology is not required.

Advanced parameters, specific when the topology is not required:

- The parameter `EnableOffSizeCharacter` tells if the detection should allow or not the detection of characters whose size (width and height) is out of the size parameters if they are in the vicinity of characters in valid size range.

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.

5.3. Learning

In order to recognize characters, EasyOCR2 can use a database of known reference characters. You can generate this character database from images and/or from TrueType system fonts.

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

1. Set the required detection parameters or call [Load](#) to read the model (.o2m) file from disk.
2. Optionally, call [ClearCharacterDatabase](#) to clear the current character database.
3. Call [Detect](#) to extract the text from the image.
4. Call [SetText](#) in the extracted text structure to set the correct value for each character.
5. Call [Learn](#) to add the detected characters and their correct value to the current character database.
6. Call [SaveCharacterDatabase](#) to save the current character database to disk.
7. 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](#)

5.4. Recognition

Recognition

EasyOCR2 recognizes characters using a pre-trained classifier or 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 **Classifier** parameter specifies the classifier used by **EOCR2** for recognition. By default, its value is **EOCR2Classifier_DatabaseClassifier** and it means that it uses the current Character Database. Other values represent pre-trained classifiers for different context.

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 alphanumeric 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 alphanumeric characters, followed by a punctuation character and 4 digits.
- The second line has 2 words. The first word comprises 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 V
DEEP LEARNING INSPECTION
TOOLS

1. Deep Learning Tools - Inspecting Images with Deep Learning

1.1. 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 Supervised segments defects and/or various elements in images. In the supervised mode, the training images must be precisely annotated with their expected segmentation (also called the *ground truth*).
- EasySegment Unsupervised 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 advantages are that the tool can detect and segment defects that are not in your dataset or that are unexpected and that it doesn't require to annotate the images with their expected segmentation (this can be very time consuming).
- 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.
- You can use EasySegment Unsupervised to produce a rough annotation of the images required by EasySegment Supervised.
- EasyLocate locates objects and/or defects in images. The neural network predicts the location and the label of the object and/or defect.
 - The location of the object is represented by its bounding box.
 - EasyLocate works well with partially occluded objects.
 - You can use it for defect detection and counting applications.
 - Compared to EasySegment, it detects two objects or defects with the same label and that are overlapping or touching each other as two different objects or defects and not as a single blob.

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

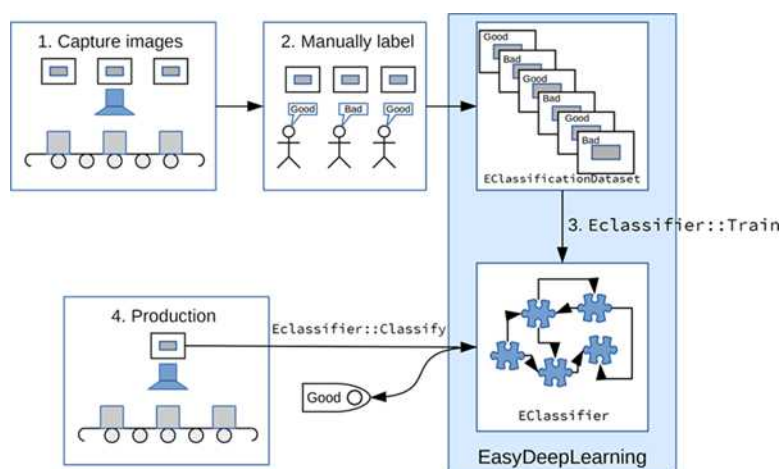
	EasyClassify	EasySegment Unsupervised	EasySegment Supervised	EasyLocate
Minimum image size	128 × 128	64 × 64		128 × 128
Maximum image size	1024 × 1024	10 000 × 10 000		500 000 pixels (for ex. 707 × 707 for square image)
Best image size	256 × 256 - 600 × 600	n.a.		n.a.
Number of channels	1 or 3 (grayscale and color images)			
Bit depth	8 bits, 16 bits			
Number of labels	2 - 1000	2 (good and defective)	2 - 64	
Minimum number of images per label	2	1 for the good label 0 for the defective label	1	
Supported formats	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 to 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.
 - With EasyClassify, you can use the training with as few as 10 images per label. Nevertheless, complex tasks may require more than 100 images per label.
 - With EasySegment Unsupervised, you can use less than 10 “good” images. Nevertheless, complex tasks may require more than 100 “good” images.
 - With EasySegment Supervised, the required number of images depends on the size and the number of elements to segment in each image. You can use as few as 10 elements per label. Nevertheless, complex tasks may require more than 100 elements per label.
 - With EasyLocate, the number of images depends on the number of objects/defects in the images. You can use as few as 10 objects per label. Nevertheless, complex tasks may require more than 100 objects per label.
 - 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 Unsupervised:
 - A single category for images of good samples.
 - As many categories as you want (including none) for images of defective samples.
- EasySegment Supervised:
 - You must annotate the pixels of the images with a ground truth segmentation
 - There must be at least one segmentation label in addition to the **Background** label.
- EasyLocate:
 - You must annotate the defects or the objects with a bounding box and a label.

Use the [EClassificationDataset](#) class to compile your labeled images.

3. Choose the deep learning tool that suits your needs.

All deep learning tools are child classes of the [EDeepLearningTool](#) class:

- EasyClassify: [EClassifier](#) class
- EasySegment: [EUnsupervisedSegmenter](#) and [ESupervisedSegmenter](#) classes.
- EasyLocate: [ELocator](#) class.

4. Train the deep learning tool on the dataset.

5. Apply the trained tool in production.

Each tool returns a specific object.

1.2. Tools and Resources

Deep Learning Studio

Deep Learning Studio is a graphical user interface that you can use to:

- Create datasets (load, label and segment images),
- Configure and visualize the data augmentation transformations,
- Train a deep learning tool,
- Analyze the performance of the tool,
- Apply the tool to new images.

**TIP**

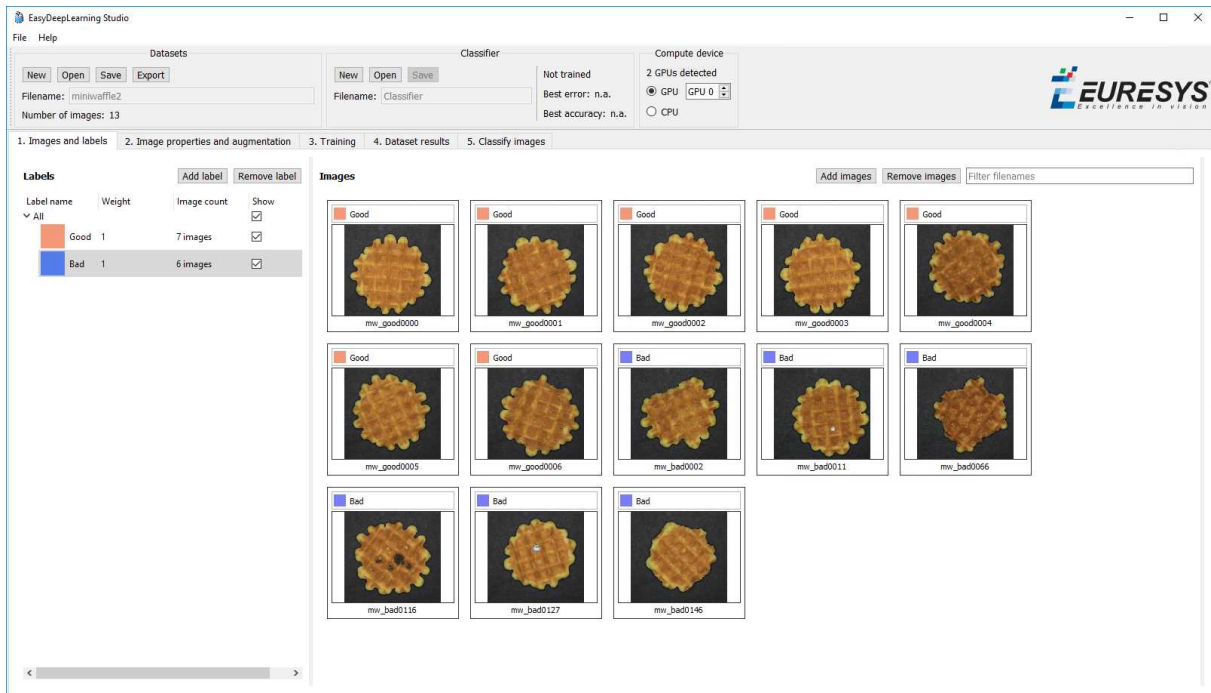
The Deep Learning Studio is available in the installation folder of Open eVision.

Resources and code snippets

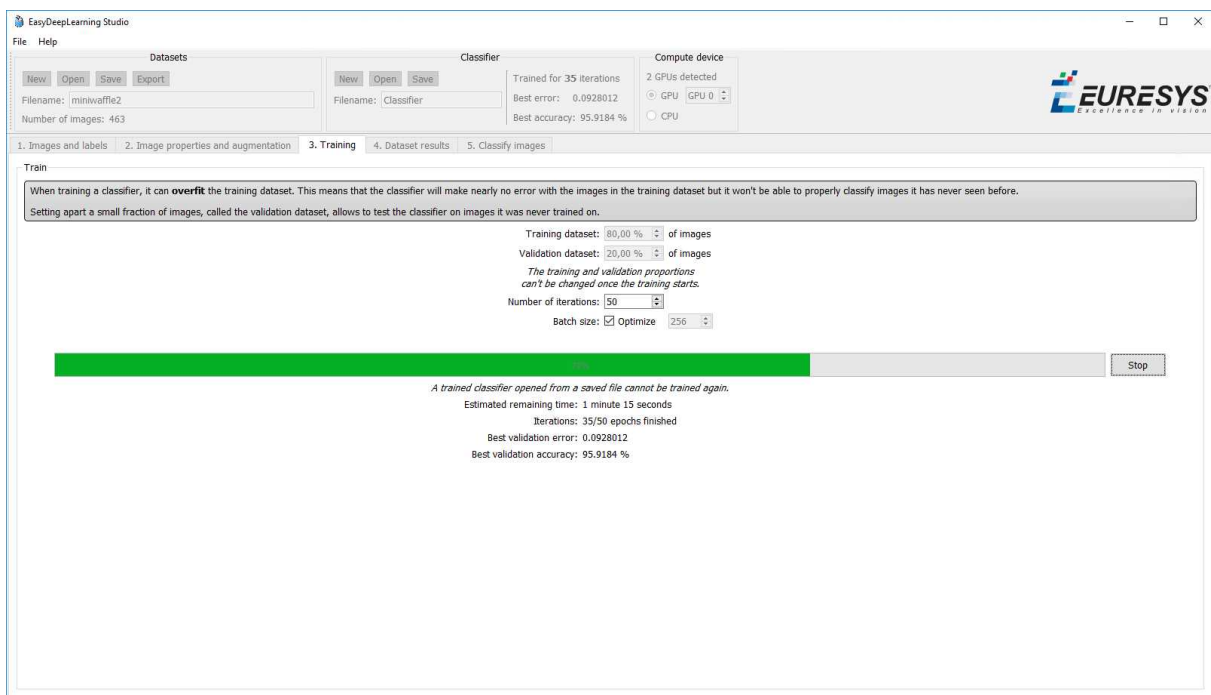
- The Deep Learning Additional Resources package, separate from the Open eVision installer, provide several sample datasets and deep learning tools trained with these datasets:
 - For EasyClassify: the `MiniWaffle` and `Stone Tiles` datasets
 - For EasySegment Unsupervised: the `Fabric` dataset
 - For EasySegment Supervised: the `Coffee` dataset
 - For EasyLocate: the `ElectronicComponentsBag` and `CeramicCapacitor` datasets
- 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

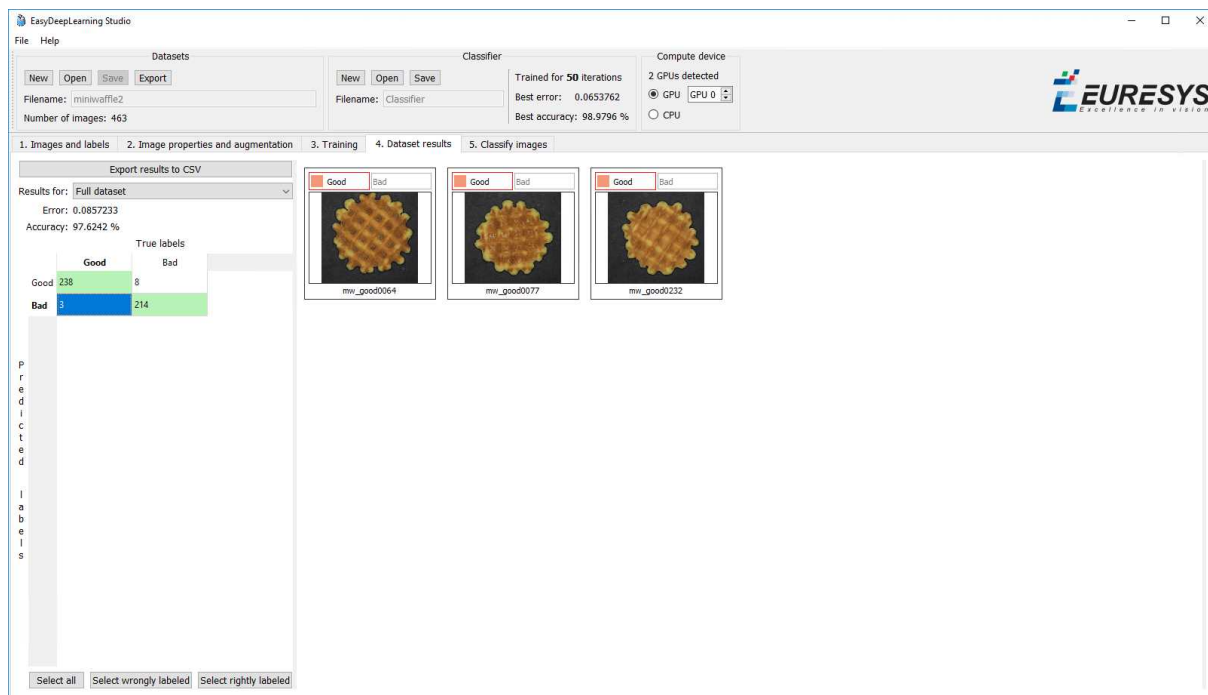
You can use Deep Learning Studio to perform steps 2 and 3 of the process described in section "Purpose and Workflow" on page 189.



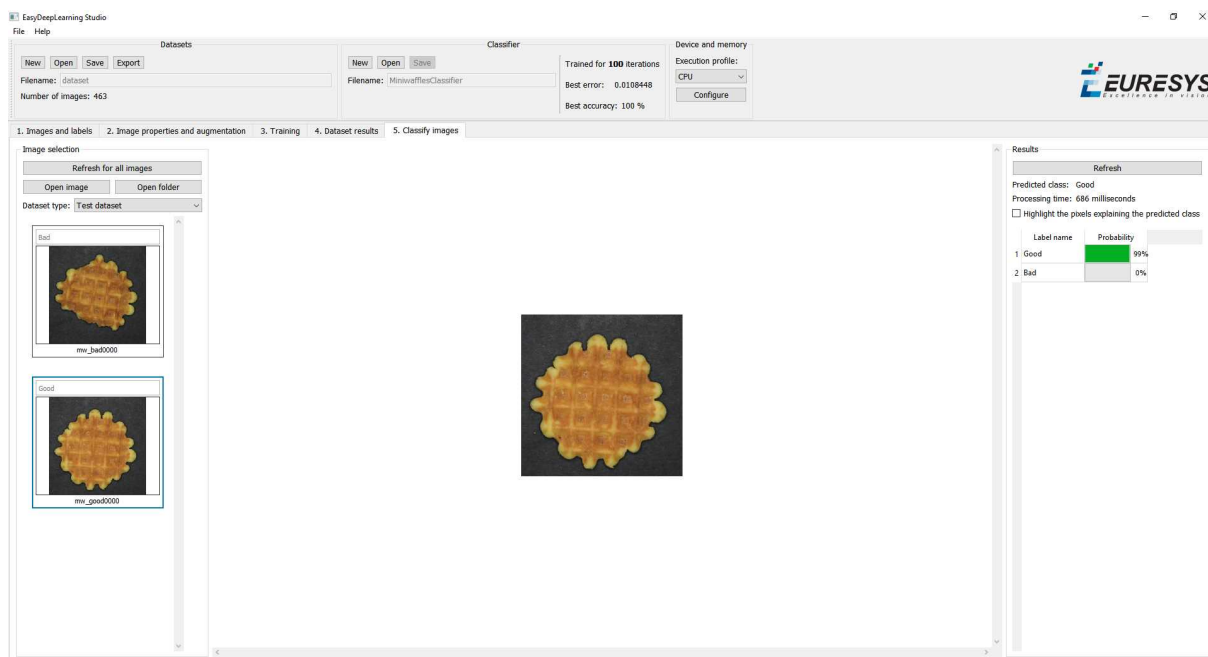
Manual labeling of images (step 2) and creating the dataset (steps 3a and 3b)



Splitting the dataset (step 3c) and starting the training (steps 3d and 3e)



Analyzing the performance (step 3f)



Classifying images (step 4)

1.3. 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 v11.1 (<https://developer.nvidia.com/cuda-toolkit>)
 - NVIDIA CUDA® Deep Neural Network library (cuDNN) v8.1 for CUDA 11.1 (<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\), a deep learning tool automatically finds what it needs.
 - Otherwise, copy the DLLs `cusolver64.dll`, `curand64_0.dll`, `cufft64_0.dll` and `cublas64_0.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 all the DLL files `cuda*.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.

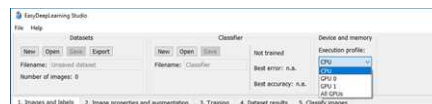


NOTE

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.



- You can configure these execution profiles to match your needs.
- GPU processing is not possible with 32-bit applications.

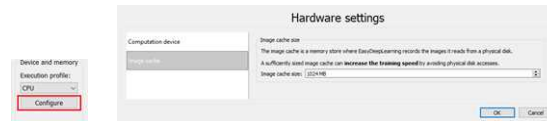
Image cache

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:

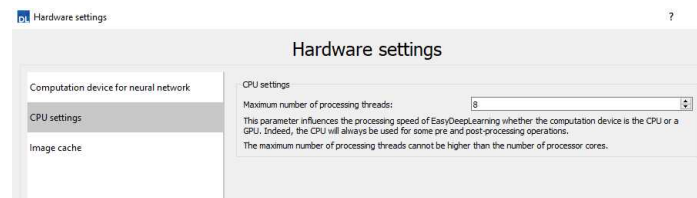
- In the API, use the `EDeepLearningTool::SetImageCacheSize` method.
- In Deep Learning Studio, click on the **Configure** button below the **Execution profile control** and select **Image cache** in the menu.



Multicore processing

The deep learning tools support multicore processing (see "[Multicore Processing](#)" on page 343):

- 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.



1.4. Managing the Images

Images and Labels

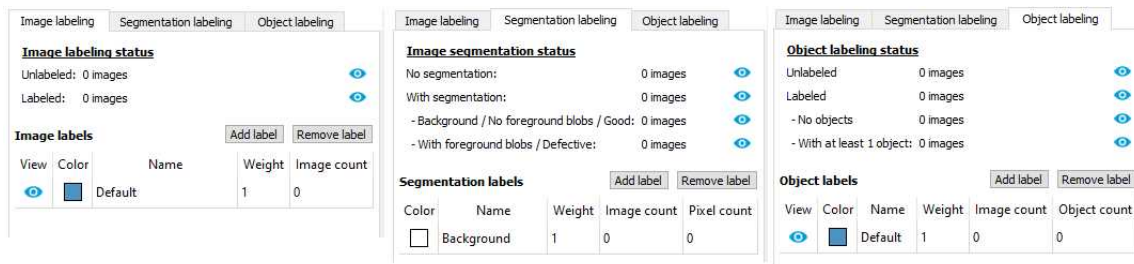
Images

- 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_BW-6`
 - `EImageType_C24`

Labels

- There are 3 types of labels:
 - The *image labels* represent a characteristic of an image and its content. Use them to annotate images for EasyClassify or EasySegment Unsupervised.
 - The *segmentation labels* represent a characteristics of pixels. Use them to annotate image pixels for EasySegment Supervised.
 - The *object labels* represent a characteristic of a region of an image delimited by a bounding box. Use them to annotate images for EasyLocate.
- Images have the following labeling states:
 - *Labeled or Unlabeled* if the image is or is not associated with an image label.
 - Only labeled images are used to train an EasyClassify or an EasySegment Unsupervised tool.
 - In the API, use `EClassificationDataset::HasLabel(imageIndex)`.
 - *With or without segmentation* if the image has or has not a ground truth segmentation.
 - Only images with segmentation are used to train an EasySegment Supervised tool.
 - In the API, use `EClassificationDataset::HasSegmenation(imageIndex)`.
 - *With or without object labeling* if the image has or has not a ground truth object labeling.
 - Only images with object labeling are used to train an EasyLocate tool.
 - In the API, use `EClassificationDataset::HasObjectLabeling(imageIndex)`.
- The ground truth segmentation of an image has the following state:
 - *Background* when all the pixels of the image are associated with the `Background` segmentation label.
 - In defect detection applications, a background segmentation means that the image contains no defect.
 - *With foreground blobs* when the segmentation contains at least one pixel associated with a segmentation label different from `Background`.
 - In defect detection applications, a segmentation with foreground blobs means that the image contains defects.
 - In the API, use `EClassificationDataset::HasForegroundSegments(imageIndex)`.
- The ground truth object labeling of an image has the following state:
 - *No objects* when there is no object in the image.
 - In defect detection applications, an image with no object means that the image contains no defect.
 - *With objects* when there is at least one object in the image.
 - In defect detection applications, an image with objects means that the image contains defects.
 - In the API, use `EClassificationDataset::GetImageNumObjects(imageIndex)` to determine if the image has objects or not.

- In Deep Learning Studio, the icons (visible) and (hidden) represent the visibility state of the images with the corresponding state and/or image label in the image list.
 - Click on these icons to toggle the visibility state.

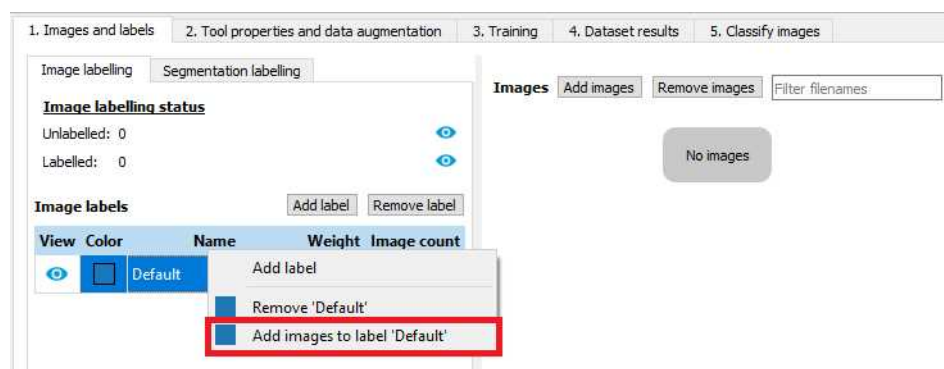


Adding Images

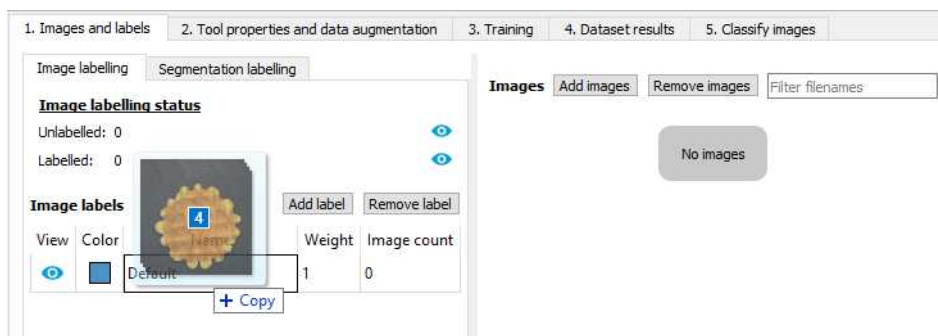
- In Deep Learning Studio, add image files (PNG, TIFF, JPEG, BMP and J2K types) to your datasets in one of the following ways:
 - Click on the **Add images** button to add images without any label nor segmentation.



- Right-click on an image label and click **Add images to label** to directly associate these images to the label.



- Drag and drop your files directly on an image label to directly associate these images to the label..



- Add a single image 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.
 - You can specify a label to immediately associate the image with the label. Otherwise, the image is unlabeled.
- Add several images with the `EClassificationDataset::AddImages(filter[, label])` method.

`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

The `EClassificationDataset` class automatically generates the set of labels from the labels of the images that you add to the dataset.



NOTE

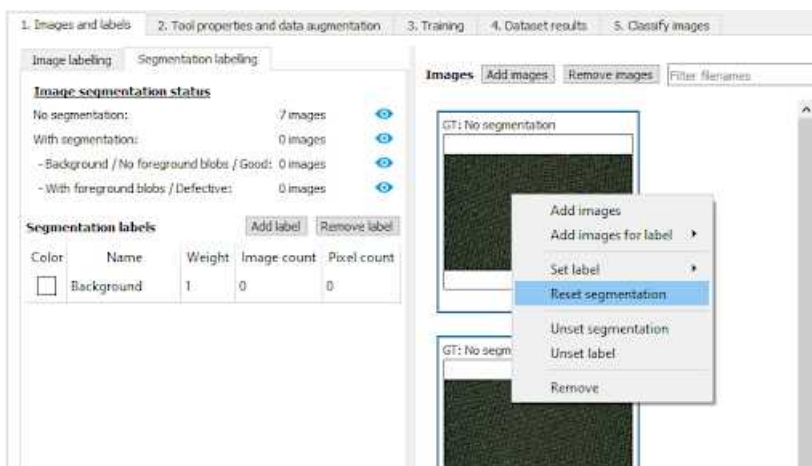
By default, all images are unlabeled and have no ground truth segmentation.

Editing the Segmentation of an Image

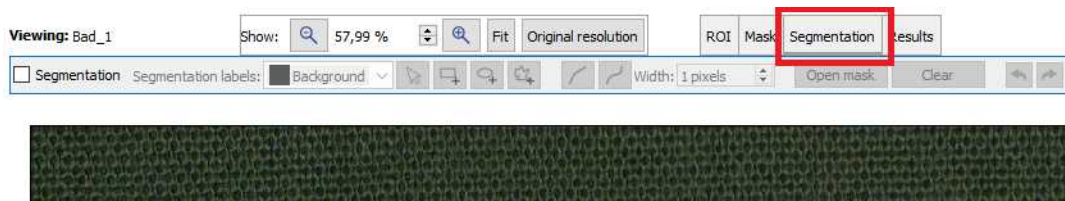
In Deep Learning Studio:

- To initialize or reset the segmentation of an image to all **Background** pixels:

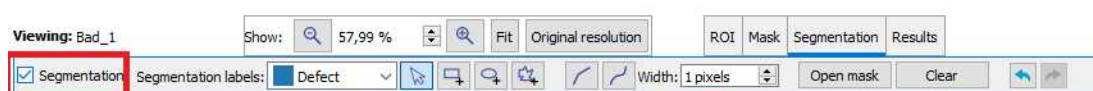
- a. Select one or more images in the image list.



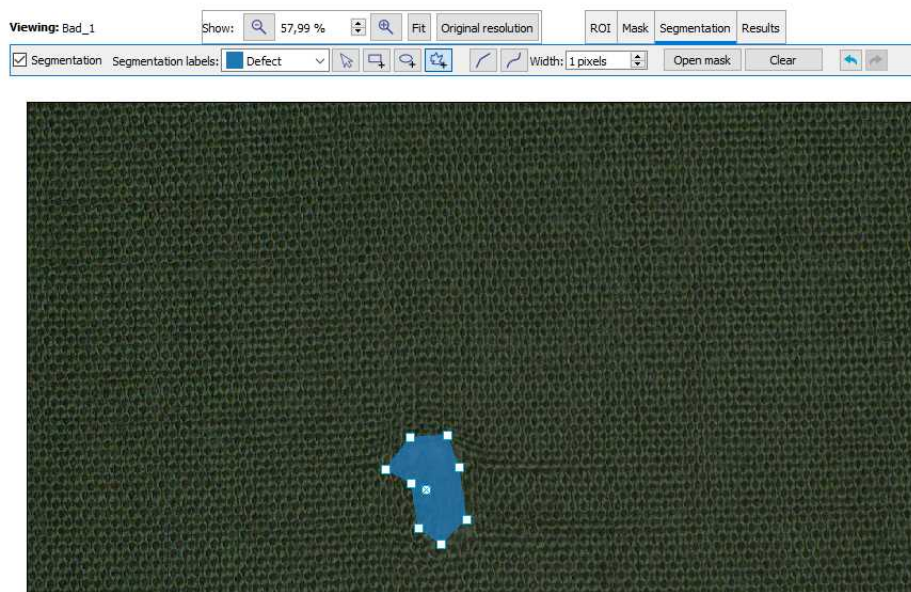
- b. Right click on the selection.
- c. Click on Reset segmentation.
- To edit the segmentation:
 - a. Double click on the image to open it in the image editor.
 - b. Click on the Segmentation button (ALT + S).



- c. To reset or unset the segmentation, uncheck the Segmentation checkbox (CTRL + S).



- d. Select a segmentation label, a drawing tool and enclose the segmentation.



- e. Change the segmentation label of a blob by right-clicking on it



Editing the Objects of an Image

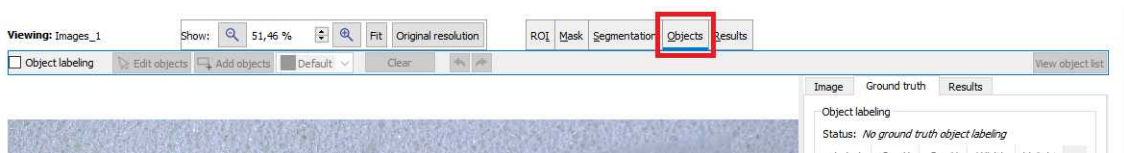
In Deep Learning Studio:

- To initialize or reset the object labeling of an image:

- a. Select one or more images in the image list.



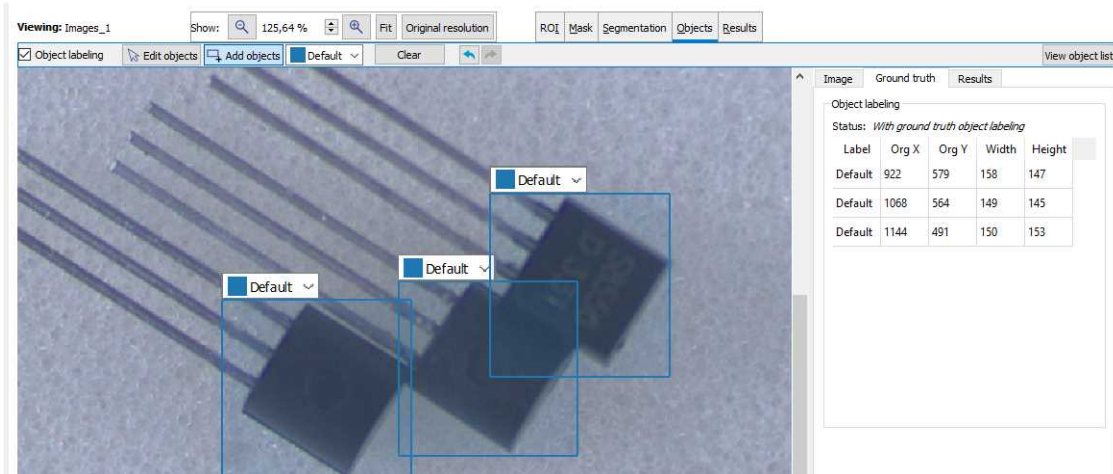
- b. Right click on the selection.
- c. Click on Reset object labeling.
- To edit the objects:
 - a. Double click on the image to open it in the image editor.
 - b. Click on the Objects button (ALT + 0).



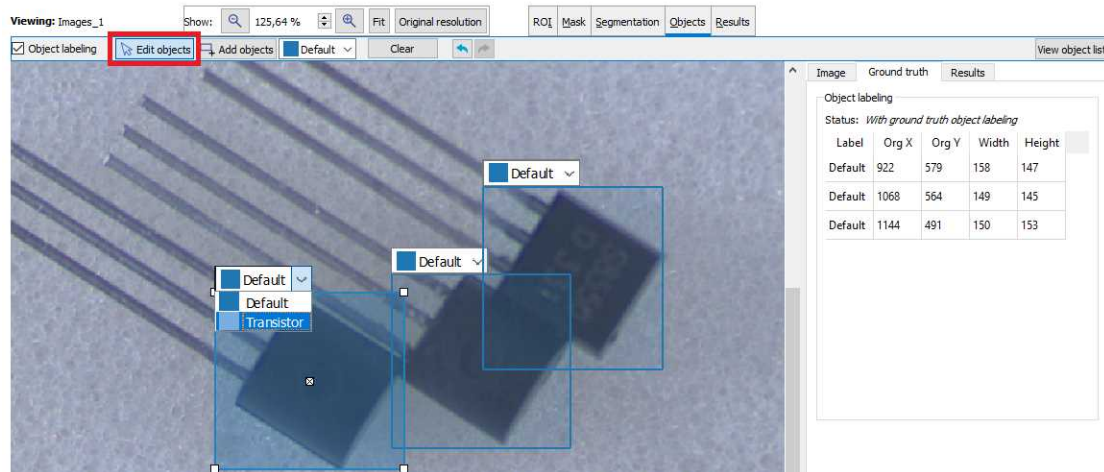
- c. To reset or unset the object labeling, uncheck the Object labeling checkbox (CTRL + L).



- d. Click on the Add objects button to add new objects with the label indicated next to the button.



- e. Click on the **Edit objects** button to modify the bounding box or the label of an object.



ROI and Mask

Setting a ROI

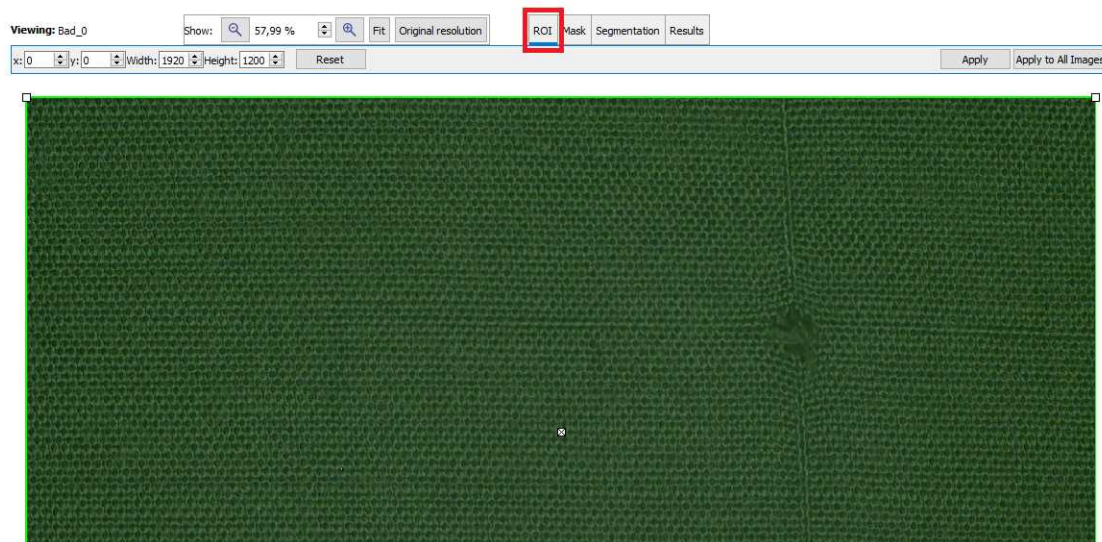
Use an ROI (region of interest) to crop an image or a whole dataset to a rectangular area aligned with the axis.

In the API:

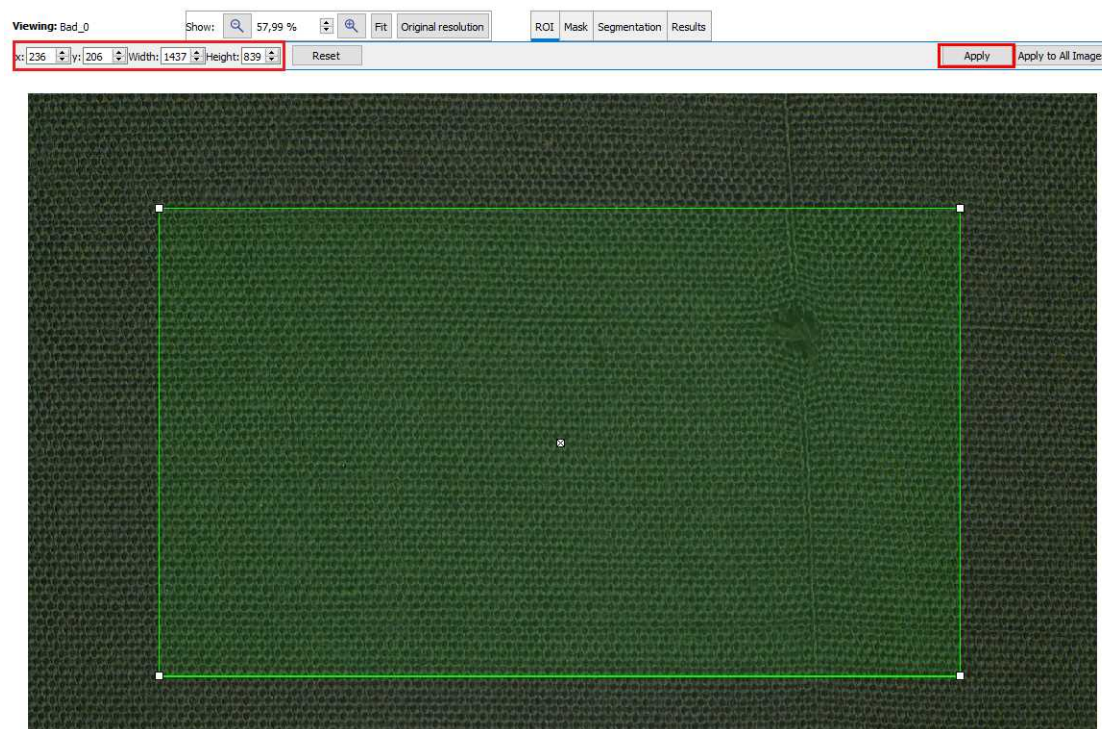
- To define an ROI for an image:
 - Specify the ROI when you add the image to the dataset.
 - Or use `EClassificationDataset::SetRegionOfInterest`.

In Deep Learning Studio:

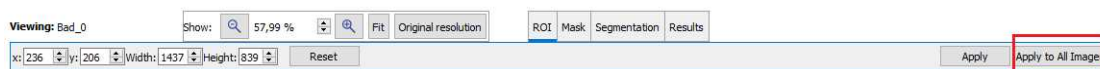
- To change the ROI:
 - a. Select an image from the dataset.
 - b. Click on the ROI button (ALT + I).



- c. Drag the ROI green box, or directly set the ROI origin (x and y), Width and Height.
- d. Click on the Apply button (CTRL + A).



- To set the same ROI for all the images of the dataset:
 - a. Set the ROI for one of the image.
 - b. Click on the Apply to All Images button (CTRL + SHIFT + A).



Setting a mask

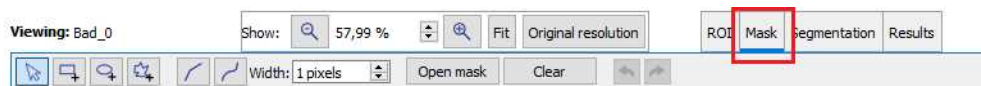
Set a mask on an image in a dataset to remove the pixels in the mask area from any computation. The mask works as a “don’t care area”.

In the API:

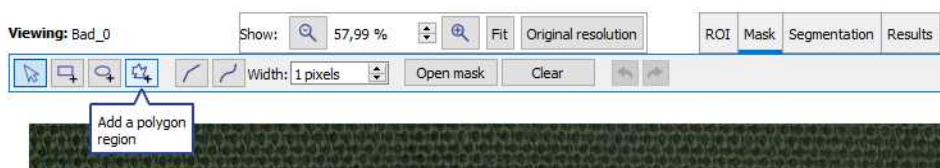
- To define a mask for an image:
 - Specify the mask when you add the image to the dataset.
 - Or use `EClassificationDataset::SetMask`.

In Deep Learning Studio:

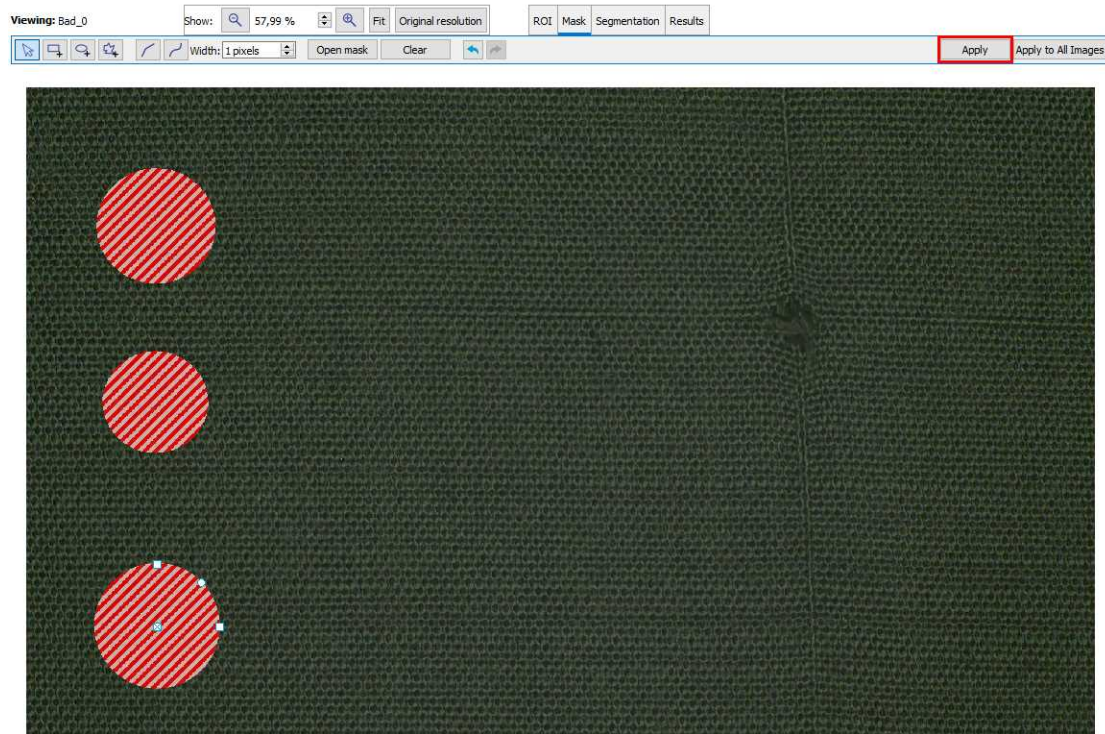
- To change the mask:
 - a. Select an image from the dataset.
 - b. Click on the Mask button (ALT + M).



- c. Select a kind of `ERegion` to draw the mask.



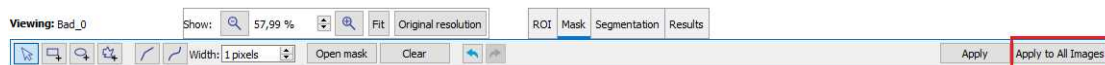
- d. Draw the mask.
- e. Click on the **Apply** button (CTRL + A).



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. Specify the mask for one of the images.
 - b. Click on the **Apply to All Images** button (CTRL + SHIFT + A).



1.5. Managing the Dataset

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 [Test and results](#).

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.



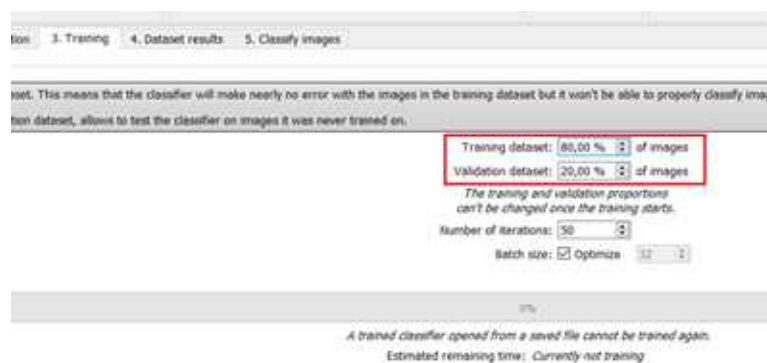
TIP

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.



- 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 methods:
 - For EasyClassify and EasySegment Unsupervised: `EClassificationDataset::SplitDataset(trainingDataset, validationDataset, trainingProportion)`
 - For EasySegment Supervised: `EClassificationDataset::SplitDatasetForSegmentation(trainingDataset, validationDataset, trainingProportion)`
 - For Easylocate: `EClassificationDataset::SplitDatasetForLocator(trainingDataset, validationDataset, trainingProportion)`

Using Data Augmentation

Data augmentation performs random transformations on images given to a deep learning tool (EClassifier, EUnsupervisedSegmenter or ESupervisedSegmenter object) during the training.

- Experiment different settings to choose the best parameters for your data augmentation.
- Configure data augmentation according to your problem. However, flips, shifts (20 - 40 px), brightness (5%), contrast (0.95 to 1.05) or salt and pepper noise (2%) can be useful on many datasets.
- 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.

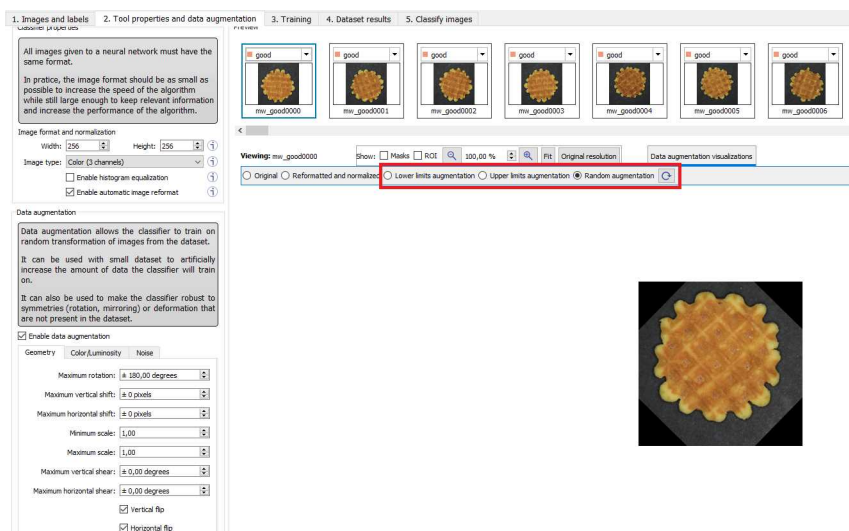


NOTE

With EasyLocate, we do not recommend to use rotation and shear data augmentation as it is not possible to compute the minimal bounding box surrounding the object after these geometric 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).

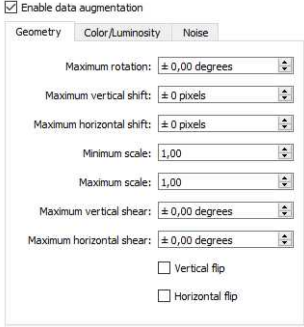


In the API:

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations.

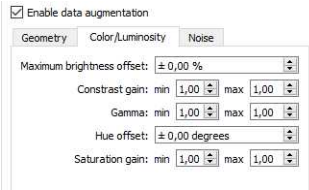
The possible transformations are:

Geometric transformations



- 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`)

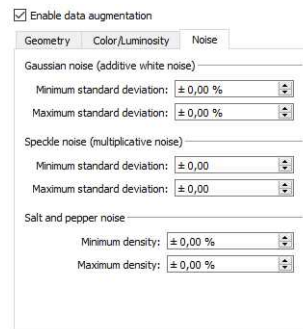
Color and luminosity transformations



- 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



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`).

1.6. Training a Deep Learning Tool

In the API, to train a deep learning tool, call the `EDeepLearningTool::Train(trainingDataset, validationDataset, numberOfIterations)` method.

- 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 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.
- *Multiple iterations:*
 - 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.
 - In successive calls to `EDeepLearningTool::Train`:
 - You can add images to the training and validation dataset to train the tool to recognize new instances of your problem.
 - We do not recommend that you remove images from the dataset as the tool might forget about these images during the new training phase.
- 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.

2. EasyClassify - Classifying Images

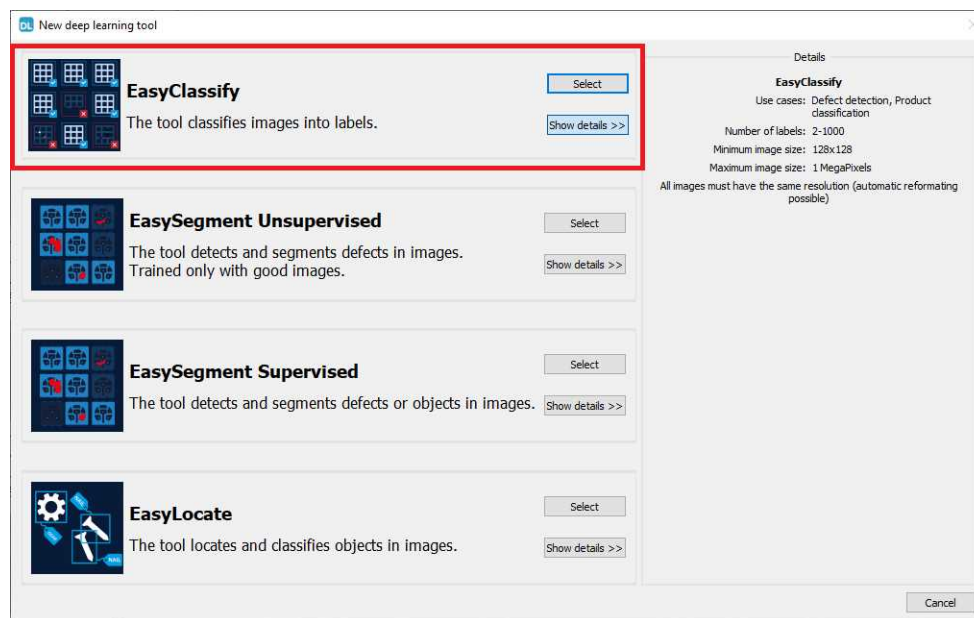
2.1. Tool and Images

EasyClassify is the deep learning classification library of Open eVision (`EClassifier` class).

Deep Learning Studio

To create a classification tool in Deep Learning Studio:

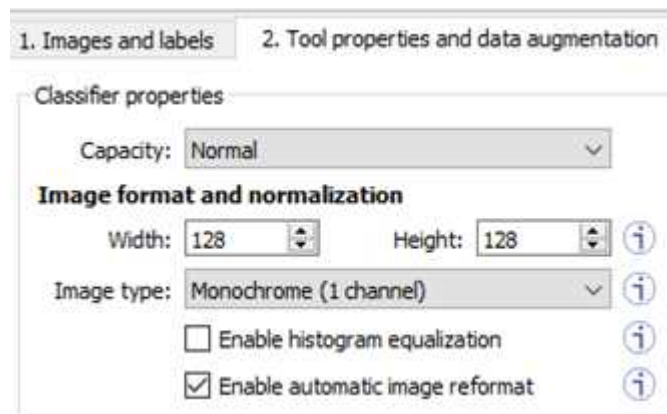
1. Start Deep Learning Studio.
2. Select EasyClassify in the New deep learning tool dialog.



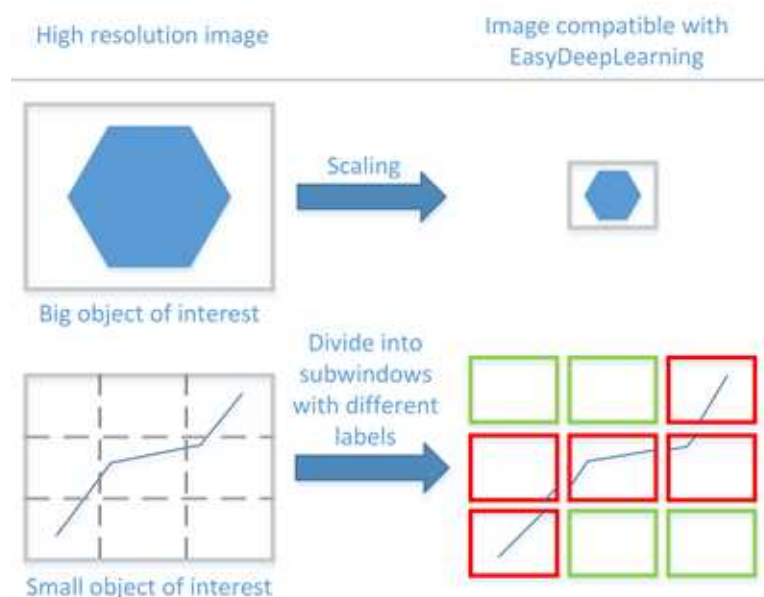
Input image format and normalization

- 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.

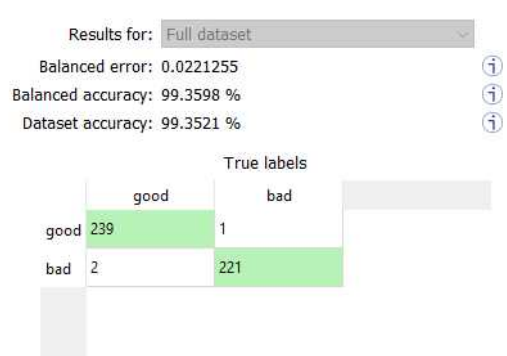
- In Deep Learning Studio, you can set the input image format in the Tool properties and data augmentation tab.



- 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 of at least 128 x 128 for the normal and the large capacity or 64 x 64 for the small capacity and at most 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 predictions. This presents the additional advantage of localizing the "object of interest" inside the original image.



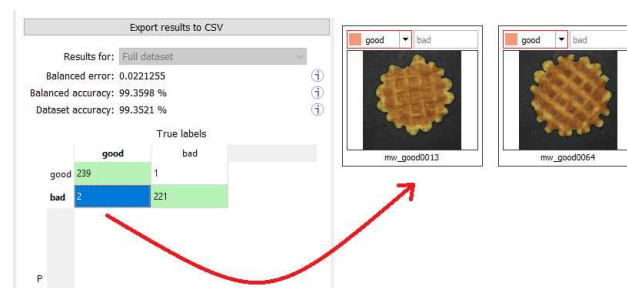
- In the **Dataset results** tab, there are 3 metrics displayed:
 - The **weighted error** and the **weighted accuracy** (normalized with respect to the label weights instead of being dependent of the number of images for each label).
 - The **dataset accuracy** (it does not use the label weights).



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.



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 `EDeepLearningTool::GetNumTrainedIterations()-1`.
 - Call `EDeepLearningTool::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.

2.3. Classifying New Images

Classify images

- In Deep Learning Studio, open the `Test` and `results` 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.
 - The heat map is colored, and the important parts are displayed in red.

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:

$$\text{ClassificationMemoryCPU} = 0.000232 \times W \times H \times B - 97 \text{ (MB)}$$

$$\text{ClassificationMemoryGPU} = 0.000226 \times W \times H \times B + 13 \text{ (MB)}$$

- 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 slower due to the memory allocations.

2.4. Benchmarks for EasyClassify

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Capacity small

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	19	2.7 ms	10.2 ms	-
	4	29	0.88 ms	-	54
	16	67	0.6 ms	-	134
	64	232	0.39 ms	-	463
256 × 256	1	30	3.3 ms	45.3 ms	-
	4	74	1.99 ms	-	144
	16	247	1.35 ms	-	497
	64	959	1.14 ms	-	1924
512 × 512	1	74	7.9 ms	199 ms	-
	4	260	4.56 ms	-	519
	16	998	5.89 ms	-	2001
	64	3908	4.74 ms	-	7936

Capacity normal

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	92	4.4 ms	13 ms	-
	4	102	1.29 ms	-	166
	16	141	0.66 ms	-	320
	64	277	0.41 ms	-	608
256 × 256	1	103	5.3 ms	50 ms	-
	4	145	2.33 ms	-	253
	16	315	1.3 ms	-	1219
	64	1042	1.1 ms	-	2144

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
512 × 512	1	155	17.8 ms	199 ms	-
	4	332	5.8 ms	-	1110
	16	1069	4.3 ms	-	2605
	64	4122	4.22 ms	-	8533

Capacity large

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	32	2.4 ms	17.1 ms	-
	4	50	1.2 ms	-	100
	16	131	0.9 ms	-	255
	64	421	0.6 ms	-	848
256 × 256	1	54	4.2 ms	69 ms	-
	4	137	2.7 ms	-	275
	16	473	1.8 ms	-	990
	64	1830	1.3 ms	-	3660
512 × 512	1	145	15.9 ms	324 ms	-
	4	502	8.6 ms	-	1004
	16	1932	7.2 ms	-	3961
	64	7690	6.2 ms	-	15380

3. EasySegment - Detecting and Segmenting Defects

3.1. Unsupervised vs Supervised Modes

EasySegment is the deep learning segmentation library of Open eVision.

It contains 2 different modes:

- The *unsupervised* mode:
 - The tool is trained only with good images ([EUnsupervisedSegmenter](#) class).
 - This mode does not require a ground truth segmentation and the creation of the dataset is thus much quicker than for the supervised mode.
 - This mode can detect unexpected defects while the supervised mode is only capable of detecting defects similar to those in the dataset.
- The *supervised* mode:
 - The tool is trained using the ground truth segmentation defined for the images ([ESupervisedSegmenter](#) class).
 - This mode can detect and segment more types of defects with better accuracy than the unsupervised mode. It directly builds a model for the defects while the unsupervised mode builds a model of the good images and tries to detect variations from this model.
 - You can also use this mode to segment other types of objects than defects.

3.2. EasySegment Unsupervised

Tool and Configuration

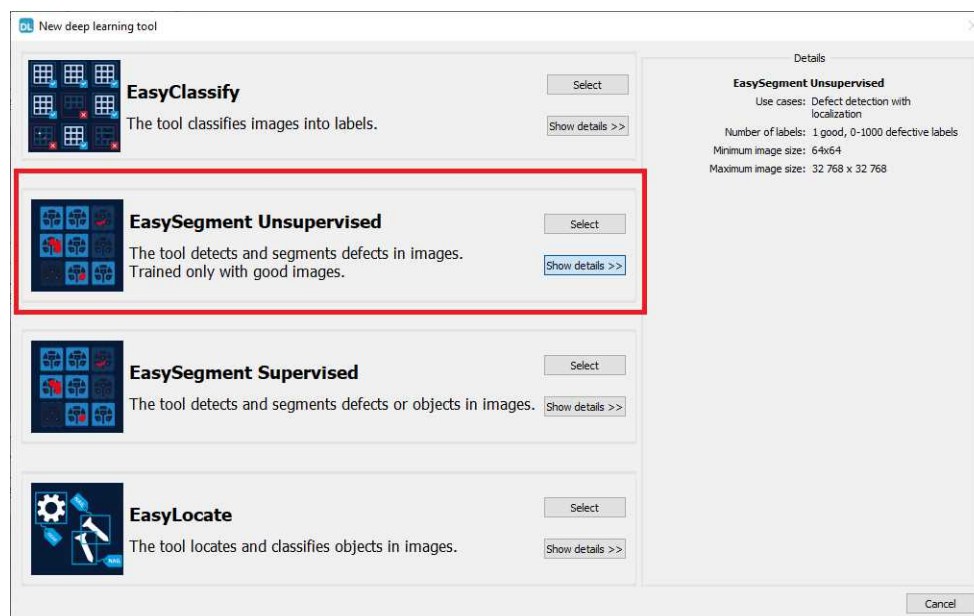
EasySegment Unsupervised is the deep learning tool part of the EasySegment segmentation library of Open eVision. It detects and segments defects in images.

This tool trains in an unsupervised way. This means that it is trained only with good images. So it does not require any ground truth segmentation of the defects.

Deep Learning Studio

To create an unsupervised segmentation tool in Deep Learning Studio:

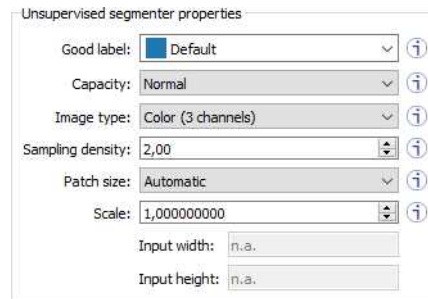
1. Start Deep Learning Studio.
2. Select EasySegment Unsupervised in the New deep learning tool dialog.



The following dialog is displayed at the start of Deep Learning Studio or when you create a new deep learning tool from the toolbar.



Configuration



The unsupervised segmenter tool has 6 parameters:

1. The **Good label** is the name of the class that contains the good images.
2. 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`.
- `EUnsupervisedSegmenter::Capacity` sets the capacity of the tool.

3. The **Image type** (default: **Monochrome** (↪ channel)):

In the API:

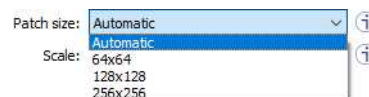
- To use monochrome (grayscale, 1 channel) images, set `EUnsupervisedSegmenter::ForceGrayscale` to **true**.
- To use color (3 channels) images, set `EUnsupervisedSegmenter::ForceGrayscale` to **false**.

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:
 $100 - 100 / \text{SamplingDensity} (\%)$
- In practice, the stride between 2 consecutive patches is:
 $\text{PatchSize} / \text{SampleDensity} (\text{pixels})$

5. The **Patch size** (`EUnsupervisedSegmenter::PatchSize`) is the size of the patches processed by the neural network.

- By default, the patch size is determined automatically from the images in the training dataset.
- You can also select the resolution of the patch size from the drop down list.



6. Use the `Scale` (`EUnsupervisedSegmenter::Scale`) to automatically resize your images to a lower resolution and accelerate the processing.

In Deep Learning Studio:

- If the dataset contains images with different resolutions, the `Input width` and the `Input height` indicate the range of the resolutions with the given scale.
- If all the images in the dataset have the same resolution, set either the `Input width` or the `Input height` to change the scale.

Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 213.

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 require 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).

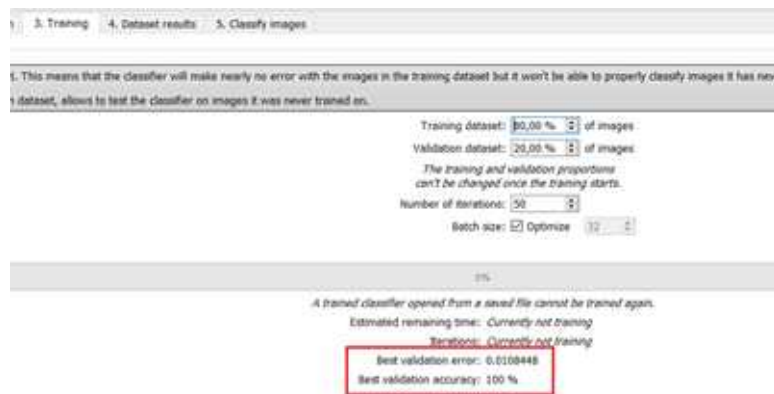
The unsupervised segmentation tool computes a score for each image (see `EUnsupervisedSegmenterResult::ClassificationScore`). The label of a result is obtained by thresholding this score with the segmenter classification threshold (`EUnsupervisedSegmenter::ClassificationThreshold`). So, the supervised 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 positive 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.



- 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 (`EDeepLearningDefectDetectionMetrics::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 (`EDeepLearningDefectDetectionMetrics::GetAccuracy`)
 - The Area under ROC curve (`EDeepLearningDefectDetectionMetrics::GetAreaUnderROCCurve`)
 - The ROC point corresponding to the classification threshold (`EDeepLearningDefectDetectionMetrics::GetROCPoint`)

Applying the Tool to New Images

In Deep Learning Studio:

- Open the **Test** and **results** 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).
- `EUnsupervisedSegmenterResult::Draw` draws the segmentation mask.

Benchmarks for EasySegment Unsupervised

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Image size

- The inference times are reported for 1024×1024 RGB images with all other settings at their default values.
- The inference times increase linearly with the width and height of the image. The inference times of a 512×512 image will be 25% of the time reported below.

Capacity small

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	2	641 ms	3082 ms	-
	4	4	299 ms	-	7
	16	13	202 ms	-	31
	64	56	199 ms	-	134

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	18	389 ms	3234 ms	-
	4	28	215 ms	-	46
	16	66	174 ms	-	132
	64	223	172 ms	-	472
256 × 256	1	71	227 ms	2519 ms	-
	4	109	170 ms	-	226
	16	256	159 ms	-	526
	64	875	166 ms	-	1777

Capacity normal

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	6	835 ms	6208 ms	-
	4	10	341 ms	-	18
	16	30	215 ms	-	67
	64	104	213 ms	-	234
128 × 128	1	65	577 ms	6245 ms	-
	4	82	261 ms	-	172
	16	152	192 ms	-	312
	64	447	189 ms	-	921
256 × 256	1	182	309 ms	5111 ms	-
	4	250	194 ms	-	620
	16	596	171 ms	-	1220
	64	1756	184 ms	-	3710

Capacity large

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	18	1188 ms	12880 ms	-
	4	25	439 ms	-	115
	16	61	245 ms	-	179
	64	228	251 ms	-	473
128 × 128	1	170	911 ms	13344 ms	-
	4	203	359 ms	-	527
	16	409	240 ms	-	846
	64	1008	219 ms	-	2214
256 × 256	1	464	853 ms	11642 ms	-
	4	594	293 ms	-	1103
	16	1613	227 ms	-	3226
	64	3968	215 ms	-	8044

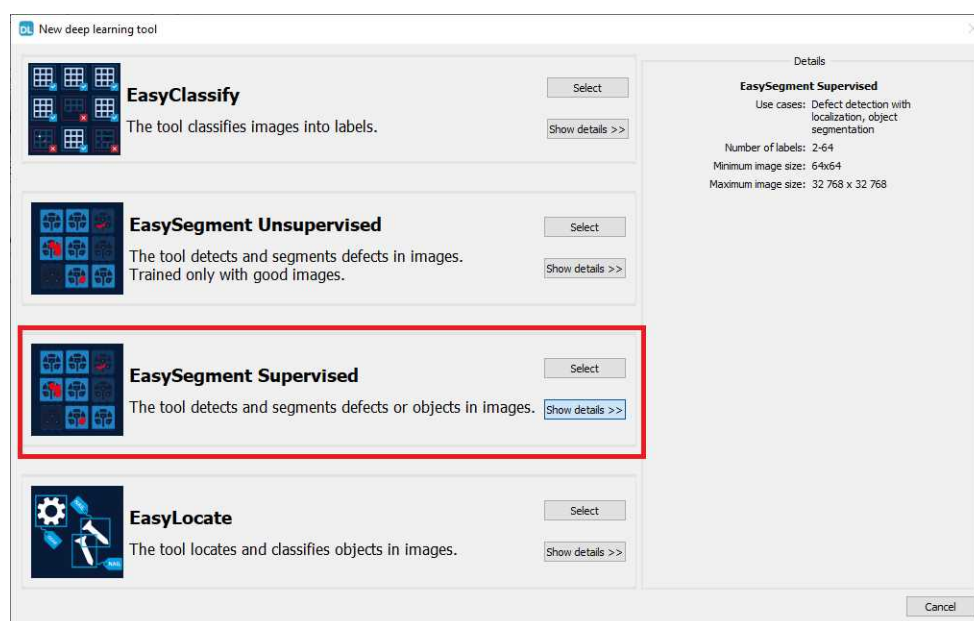
3.3. EasySegment Supervised

Tool and Configuration

Deep Learning Studio

To create an supervised segmentation tool in Deep Learning Studio:

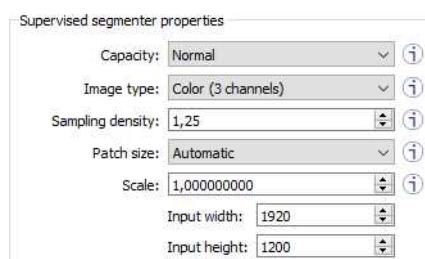
1. Start Deep Learning Studio.
2. Select EasySegment Supervised in the New deep learning tool dialog.



The following dialog is displayed at the start of Deep Learning Studio or when you create a new deep learning tool from the toolbar.



Configuration



The supervised 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 `ESupervisedSegmenterCapacity`.
- `ESupervisedSegmenter::Capacity` sets the capacity of the tool.

2. The **Image type** (default: **Monochrome** (↳ channel)):

In the API:

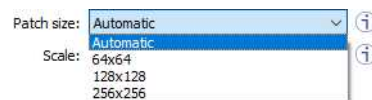
- To use monochrome (grayscale, 1 channel) images, set `ESupervisedSegmenter::ForceGrayscale` to **true**.
- To use color (3 channels) images, set `EUnsupervisedSegmenter::ForceGrayscale` to **false**.

3. The **Sampling density** (`ESupervisedSegmenter::SamplingDensity`) is the parameter of the sliding window algorithm used to process whole images using patches of size (`ESupervisedSegmenter::PatchSize`).

- It indicates how much overlap there is between the image patches:
 $100 - 100 / \text{SamplingDensity} (\%)$
- In practice, the stride between 2 consecutive patches is:
 $\text{PatchSize} / \text{SampleDensity} (\text{pixels})$

4. The **Patch size** (`ESupervisedSegmenter::PatchSize`) is the size of the patches processed by the neural network.

- By default, the patch size is determined automatically from the images in the training dataset.
- You can also select the resolution of the patch size from the drop down list.



5. Use the **Scale** (`ESupervisedSegmenter::Scale`) to automatically resize your images to a lower resolution and accelerate the processing.

In Deep Learning Studio:

- If the dataset contains images with different resolutions, the **Input width** and the **Input height** indicate the range of the resolutions with the given scale.
- If all the images in the dataset have the same resolution, set either the **Input width** or the **Input height** to change the scale.

Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 213.

Using the Supervised Segmenter

To get the result, a supervised segmenter follows these steps:

1. For each pixel, the supervised segmenter tool computes the probabilities that it belongs to each of the segmentation labels.
2. From these probabilities, it extracts a set of potential foreground blobs (groups of contiguous pixels for which the highest probability corresponds to the same foreground segmentation label).
3. For each one of these potential foreground blobs:
 - It computes a score.
 - It removes, from the predicted segmentation map, the blobs with a score that is below or equal to the threshold of the supervised segmenter tool.
4. The score of an image is the maximum among the scores of the potential foreground blobs.

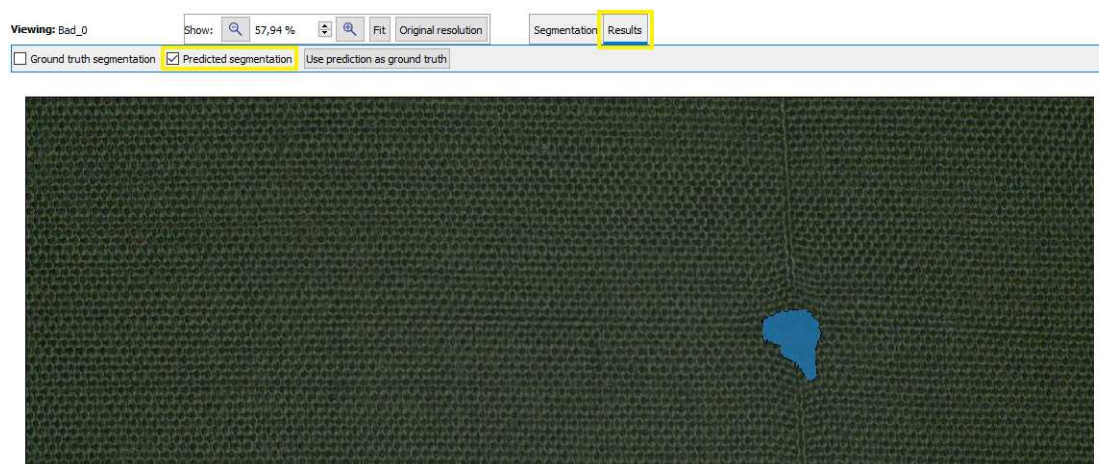


NOTE

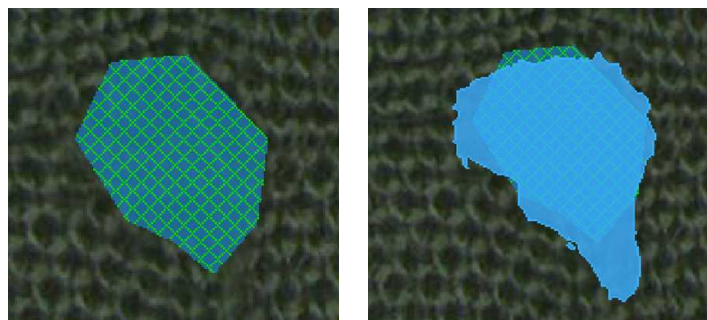
In the context of defect detection, an image is considered to be without defect when its score is below or equal to the threshold of the supervised segmenter tool.

In Deep Learning Studio:

- To apply the segmenter to new images, add these images to the dataset or use the **Test** and **results** tab.
- To visualize the segmentation of an image, check the **Predicted segmentation** option (CTRL + P) in the **Result** menu (ALT + R) of the image viewer.



- If the image has a ground truth, check the **Ground truth segmentation** option (CTRL + G) to display it. It appears with a green pattern drawn over it.

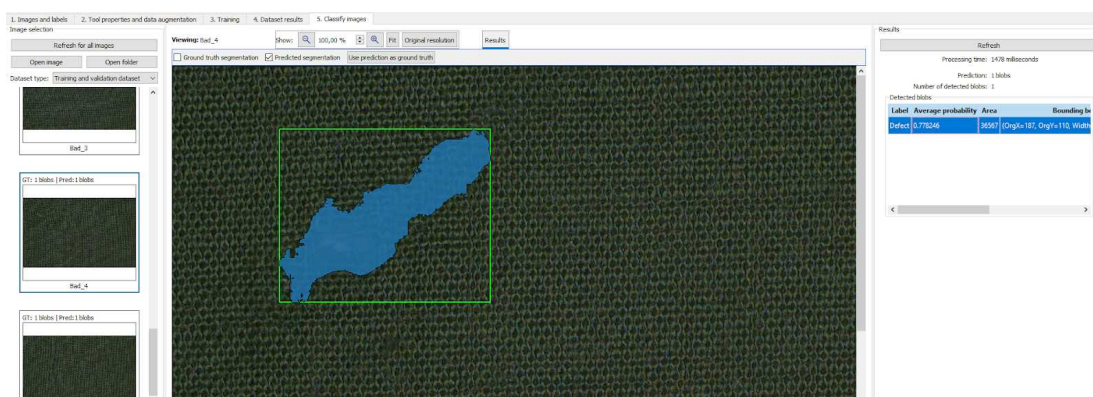


Ground truth (left) and prediction on top of ground truth (right)

- To accept the whole predicted segmentation as ground truth, click on the **Use prediction as ground truth** button (CTRL + U).
- To accept a single predicted blob as ground truth, right click on the blob and select **Accept into ground truth** in the menu.



- A list of blobs with various characteristics is available in the **Classify** tab.



In the API:

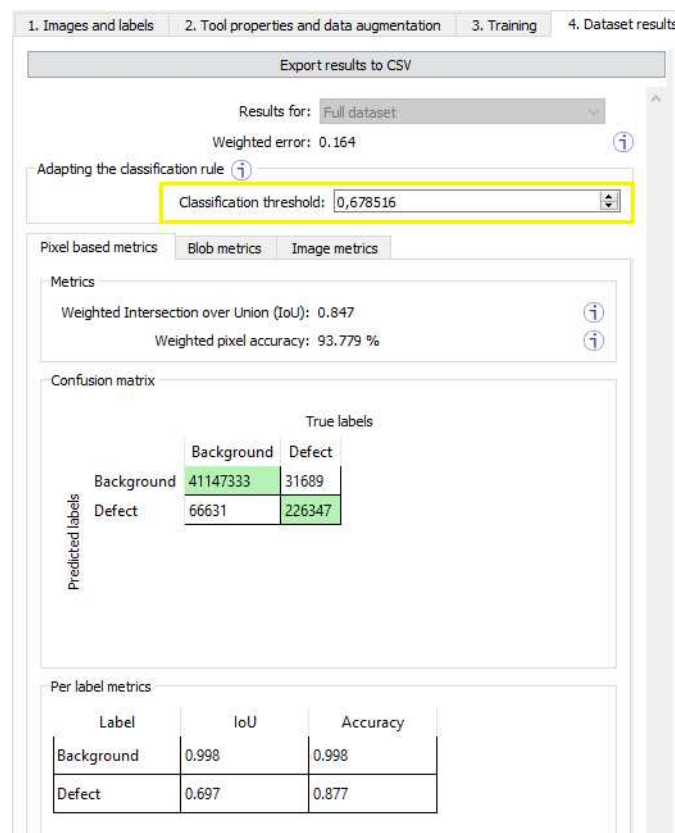
- To apply the supervised segmenter to an image use `ESupervisedSegmenter::Apply`. This method returns a `ESupervisedSegmenterResult` object.
 - Use `ESupervisedSegmenterResult::GetProbabilityMap` to retrieve the probability map for a given label. The probability map pixels contain the index of the predicted label.
 - Use `ESupervisedSegmenterResult::Draw` to draw the segmentation with the segmentation label colors of the dataset used for training.
 - Use `ESupervisedSegmenterResult::GetBlobs` to retrieve the filtered list of blobs.
 - Use `ESupervisedSegmenterResult::Score` to retrieve the score of an image.
 - Use `ESupervisedSegmenterResult::GetRegionForLabel` to obtain an `ERegion` object containing the pixels of the specified label.

Evaluating the Results

There are 3 types of metrics for the supervised segmentation tool:

- The *pixel-based metrics* that quantify the performance of the tool at the pixel level.
- The *blob-based metrics* that quantify the performance of the tool at the blob level. A blob is a contiguous region of pixels that have the same foreground segmentation label. By definition, there is no background blob.
- The *image-based metrics* that quantify the performance of the tool at the image level. These metrics are related to the capacity of the tool to correctly detect background images (images with no blobs) and foreground images (images with blobs).

- In Deep Learning Studio:
 - The metrics are available in the **Dataset results** tab.
 - Most metrics depends on the value of the **Classification threshold**.



1. Images and labels | 2. Tool properties and data augmentation | 3. Training | 4. Dataset results

Export results to CSV

Results for: Full dataset

Weighted error: 0.164

Adapting the classification rule

Classification threshold: 0,678516

Pixel based metrics | Blob metrics | Image metrics

Metrics

Weighted Intersection over Union (IoU): 0.847

Weighted pixel accuracy: 93.779 %

Confusion matrix

		True labels	
		Background	Defect
Predicted labels	Background	41147333	31689
	Defect	66631	226347

Per label metrics

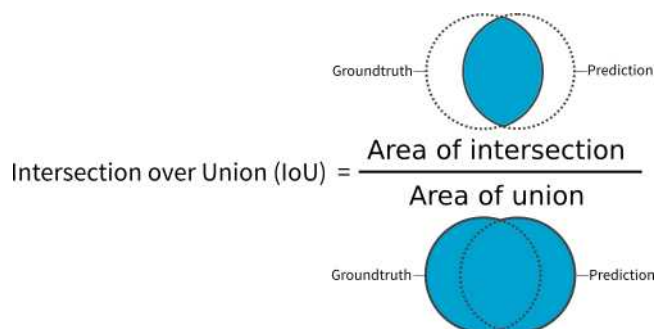
Label	IoU	Accuracy
Background	0.998	0.998
Defect	0.697	0.877

- In the API:
 - The metrics are represented by an `ESupervisedSegmenterMetrics` object.

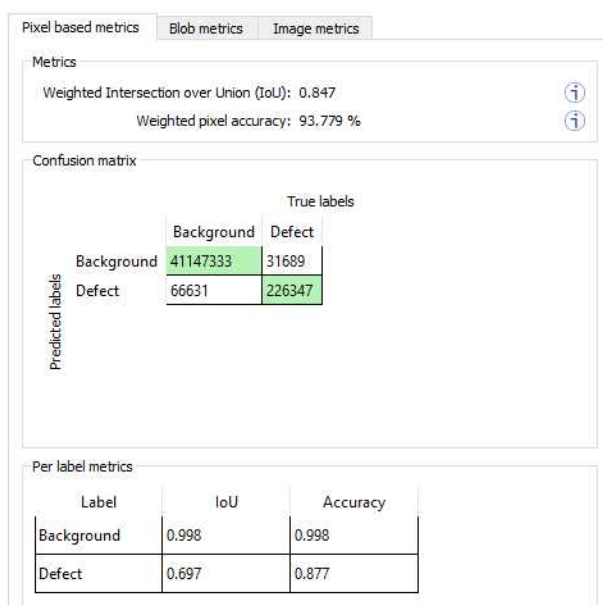
The pixel-based metrics

- The pixel-based metrics are:
 - The weighted *Intersection over Union* (IoU) that is the weighted average of the IoU over all the labels (see per-label metrics).
 - The weighted *pixel accuracy* that is the weighted average of the accuracy over all the labels (see per-label metrics).
 - The *pixel confusion matrix* that shows the number of pixels from a given label that are predicted to belong to another label.

- The per label metrics are:
 - The *Intersection over Union* (IoU) that is the ratio of the intersection between the ground truth and the prediction for the label to the union of the ground truth and prediction for the label.



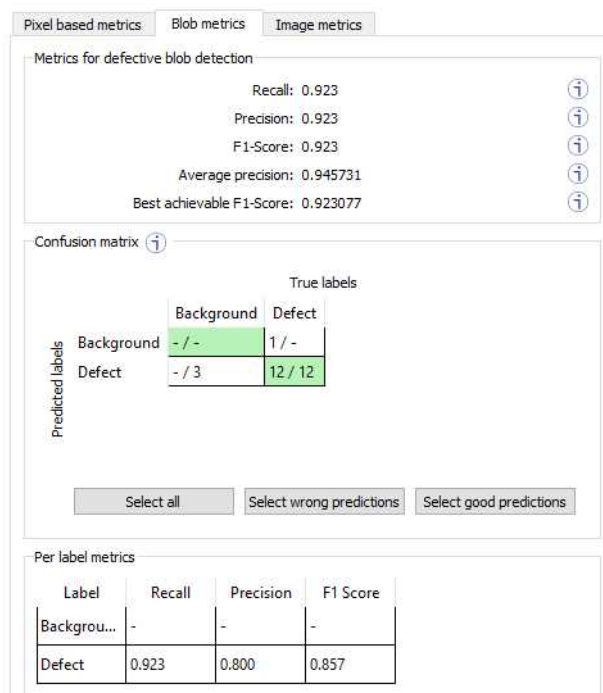
- The *accuracy* that is the proportion of the pixels of the label that are correctly predicted.



The blob-based metrics

- The metrics related to the correct prediction of blobs are:
 - The *recall* that is the ratio of the correctly predicted blobs to the total number of ground truth blobs.
 - The *precision* that is the ratio of correctly predicted blobs to the total number of predicted blobs.
 - The *F1-Score* that is the harmonic mean of the *recall* and the *precision*.
 - The *average precision* that is the average of the *precision* for different threshold weighted by the *recall* values.
 - The *best achievable F1-Score* that is the maximum *F1-Score* achievable by selecting an appropriate threshold.

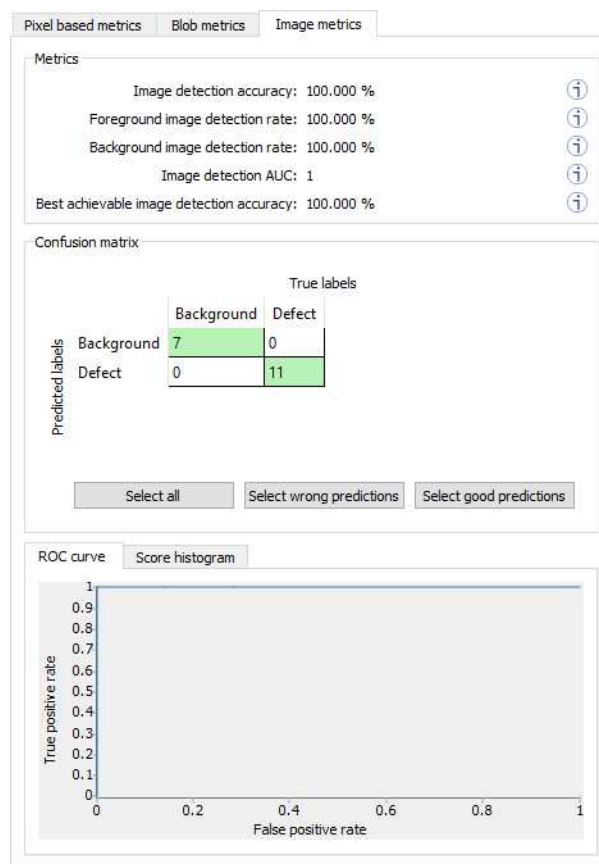
- The confusion matrix:
 - It shows the number of blobs of a given true label that are predicted to be of the corresponding predicted label.
 - The image list of the [Dataset results](#) tab only shows the images containing the blobs corresponding to the selected cells of the matrix.
 - Each matrix element shows the number of ground truth blob and the number of corresponding predicted blobs separated by a "/" as a ground truth blob can correspond to one or more blobs in the prediction and inversely.
A dash (-) indicates that blobs are not defined for this category.
 - For example, in the screenshot below, there are:
 - 3 predicted blobs of the label [Defect](#) that correspond to the [Background](#).
 - 1 ground truth blob of the label [Defect](#) that does not correspond to any predicted blob.
 - 12 predicted blobs of the label [Defect](#) that correspond to 12 ground truth blob of the same label.
- The metrics for each individual foreground label are:
 - *Recall*
 - *Precision*
 - *F1-Score*



The image-based metrics

- The metrics related to the correct detection of the class of the images (background / foreground or good / defective in the context of defect detection):
 - The *image detection accuracy* that is the proportion of image correctly predicted to have foreground blobs or not.
 - The *foreground image detection rate* that is the proportion of correctly predicted images with foreground blobs.
 - The *background image detection rate* that is the proportion of correctly predicted images with no foreground blobs.

- The *image detection AUC* (Area under the ROC Curve, see ROC Curve below).
 - The *best achievable image detection accuracy* that is the maximum *image detection accuracy* obtained by changing the threshold.
- The confusion matrix:
 - It shows the number of images of a given true label that are predicted to be of the corresponding predicted label.
 - The image list of the [Dataset results](#) tab only shows the images corresponding to the selected cells of the matrix.
- The 2 available graphics are:
 - The *ROC curve* that plots the true positive rate (*foreground image detection rate*) versus the false positive rate (1 minus the *background image detection rate*) for various threshold values.
Click on a point on the plot to set the threshold at the corresponding value.



- The *score histogram* that plots:
 - In green: the cumulative histogram of the scores of the background images.
 - In orange: the cumulative histogram of the scores of the foreground images.
 - The blue line corresponds to the current value of the threshold.



Benchmarks for EasySegment Supervised

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Image size

- The inference times are reported for 1024 × 1024 RGB images with all other settings at their default values.
- The inference times increase linearly with the width and height of the image. The inference times of a 512 × 512 image will be 25% of the time reported below.

Capacity small

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	16	545 ms	10090 ms	-
	4	29	187 ms	-	151
	16	87	123 ms	-	263
	64	335	122 ms	-	573

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	27	261 ms	14085 ms	-
	4	83	133 ms	-	306
	16	366	102 ms	-	757
	64	1441	114 ms	-	3139
256 × 256	1	148	544 ms	17606 ms	-
	4	369	151 ms	-	1185
	16	1497	111 ms	-	3091
	64	7050	148 ms	-	12564

Capacity normal

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	18	789 ms	24458 ms	-
	4	45	271 ms	-	229
	16	213	160 ms	-	450
	64	828	171 ms	-	1753
128 × 128	1	110	1952 ms	34032 ms	-
	4	216	346 ms	-	878
	16	884	188 ms	-	1864
	64	3446	177 ms	-	6124
256 × 256	1	301	327 ms	42868 ms	-
	4	864	280 ms	-	1822
	16	3027	156 ms	-	6632
	64	14048	-	-	25796

Capacity large

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	91	7443 ms	58658 ms	-
	4	140	1076 ms	-	727
	16	440	437 ms	-	1201
	64	1666	346 ms	-	3332
128 × 128	1	226	1043 ms	83118 ms	-
	4	562	820 ms	-	996
	16	1822	289 ms	-	4222
	64	6925	-	-	16156
256 × 256	1	877	1153 ms	106446 ms	-
	4	2249	567 ms	-	3887
	16	7417	-	-	15603
	64	28225	-	-	50306

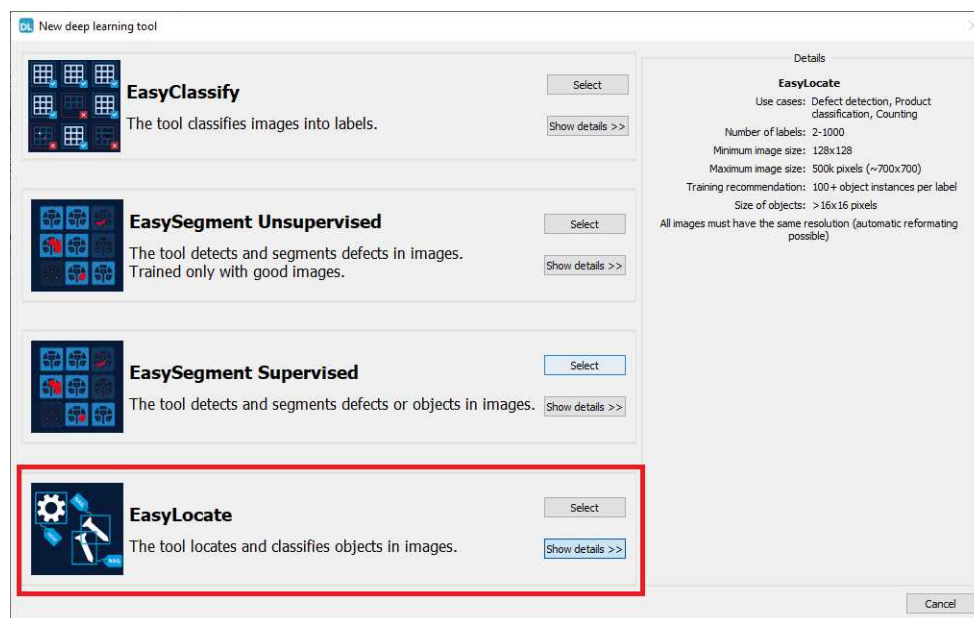
4. EasyLocate - Locating Objects and Defects

4.1. Tool and Configuration

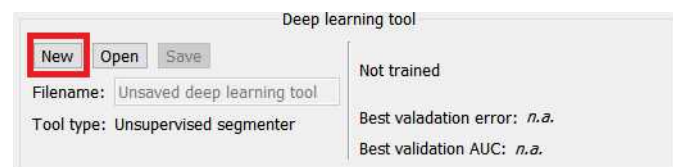
Deep Learning Studio

To create an EasyLocate tool in Deep Learning Studio:

1. Start Deep Learning Studio.
2. Select EasyLocate in the New deep learning tool dialog.



The following dialog is displayed at the start of Deep Learning Studio or when you create a new deep learning tool from the toolbar.



Configuration (main parameters)

The EasyLocate tool has 3 main 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 `ELocatorCapacity`.
- `ELocator::Capacity` sets the capacity of the tool.

2. The **Image type** (default: *Monochrome* (1 channel) if the dataset contains only grayscale images, otherwise *color* (3 channels)):

In the API:

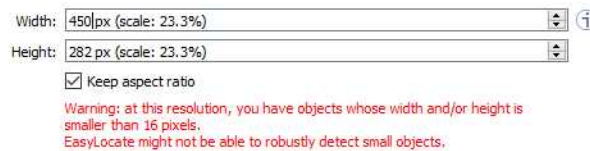
- To use monochrome (grayscale, 1 channel) images, set `ELocator::Channels` to 1.
- To use color (3 channels) images, set `ELocator::Channels` to 3.

3. The size of the images (**Width** and **Height**). You must configure EasyLocate for a specific image size.

- The image must contain less than 500 000 pixels (about 707×707 pixels for a square image).
- The width and the height must be at least 128 pixels.
- The images are automatically resized to the specified size before EasyLocate processes them.
- The lower the image size, the faster EasyLocate is.
- EasyLocate works best with objects equal to or bigger than 16×16 pixels.

In Deep Learning Studio:

- Use the **Width** and **Height** controls to change the size of the images.
- Uncheck **Keep aspect ratio** if you want to control the width and height independently of each other.
- By default, the width and height are set to the size of the images in the dataset. or, when they are bigger than 500 000 pixels, to the maximum possible size so that the aspect ratio of the image is kept and it contains at most 500 000 pixels.
- A warning is displayed when the selected size makes the ground truth objects smaller than 16×16 pixels.



Width: 450px (scale: 23.3%) ⓘ

Height: 282px (scale: 23.3%) ⓘ

Keep aspect ratio

Warning: at this resolution, you have objects whose width and/or height is smaller than 16 pixels.
EasyLocate might not be able to robustly detect small objects.

In the API:

- Use `ELocator::Width` and `ELocator::Height` to specify the image size.

Configuration (advanced parameters)

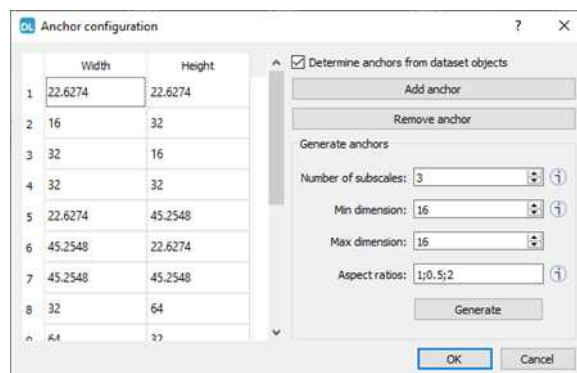
The EasyLocate tool has 4 advanced parameters linked to EasyLocate neural network design.

- The EasyLocate neural network works as an image pyramid where the size of the input image is halved at each level. EasyLocate attempts to detect objects using one or more pyramid levels depending on the size of the objects to detect.
- To do so, EasyLocate uses a set of typical object size, called *anchors*, that are assigned to pyramid levels according to their surface. Then, for each pixel and anchor of a pyramid level, EasyLocate predicts whether there is an object or not located around that pixel and whose size approximately matches the anchor.
- EasyLocate then performs a post-processing on the prediction of the neural network. Indeed, the neural network can predict the same object several times using different levels of the pyramid, different anchors or neighboring positions in the image. EasyLocate keeps only the prediction with the highest probability and removes duplicates based on the overlap between objects.

The advanced parameters are:

1. The **anchors**. By default, the anchors are determined automatically from the objects in your dataset. The set of anchors must reflect the variety of object sizes that must be detected.

To check or manually edit the anchors, click on **See/edit anchors** to open the following dialog:



The dialog lists the current anchors and enables the following operations:

- To edit an existing anchor, double-click on its width or on its height in the list.
- To add or remove an anchor, click on the corresponding button.
- To generate a new set of anchors, specify the number of subscales, the minimum and the maximum dimensions of the anchors and one or more aspect ratios.

The dimension of an anchor is the square root of its surface and determines the pyramid level assigned to the anchor. The number of subscales represent the number of dimensions to generate for each pyramid level. For each of those dimensions, the anchors with the specified aspect ratios are generated.

In the API:

- Use `ELocator::SetPredictionAnchors` and `ELocator::GenerateAnchors`.

2. The **Maximum number of objects in an image**. By default the value is 100. A lower value can speed up the post-processing of the results.

In the API:

- Use `ELocator::MaxNumberOfObjects`.

3. The **Same label maximum overlap** is the maximum overlap between objects with the same label. By default the value is 0.5.

In the API:

- Use `ELocator::SameLabelMaxOverlap`.

4. The **Absolute maximum overlap** is the maximum overlap between objects, regardless of their label. By default the value is 1 and it means that the tool can predict two objects with different labels but with the exact same bounding box.

In the API:

- Use `ELocator::SameLabelMaxOverlap`.

The overlap between two objects is their intersection over union (IoU), defined as the ratio between the surface of the intersection of their bounding boxes and the surface of the union of their bounding boxes.

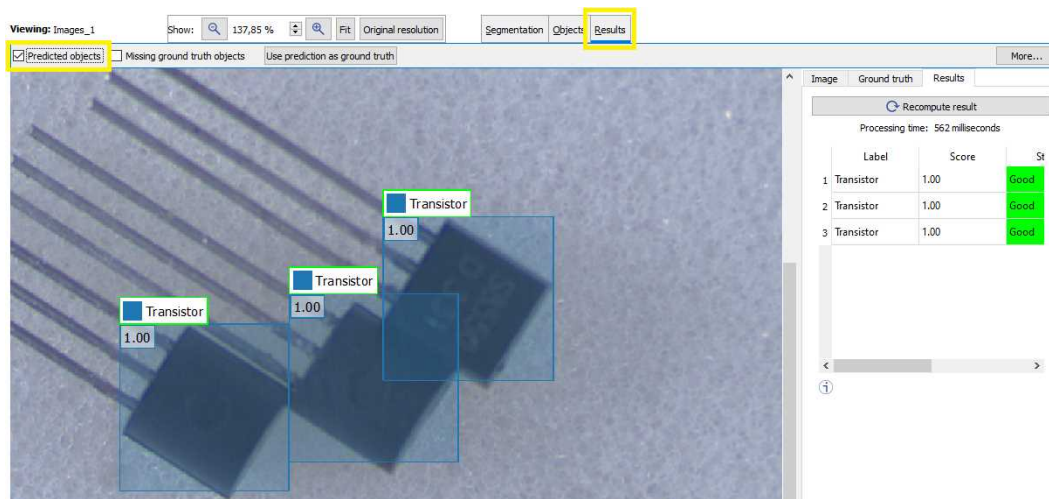
Training

To train your tool, see ["Training a Deep Learning Tool" on page 213](#).

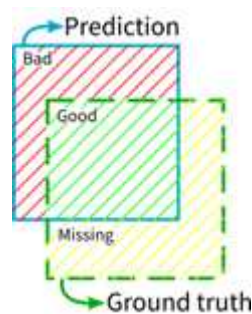
4.2. Locating Objects

In Deep Learning Studio:

- To apply EasyLocate to new images:
 - Add these images to the dataset.
 - Or use the Tests and results tab.
- To visualize the predicted objects of an image:
 - a. Open the Results menu (ALT + R) of the image viewer.
 - b. Check the Predicted objects option (CTRL + P).



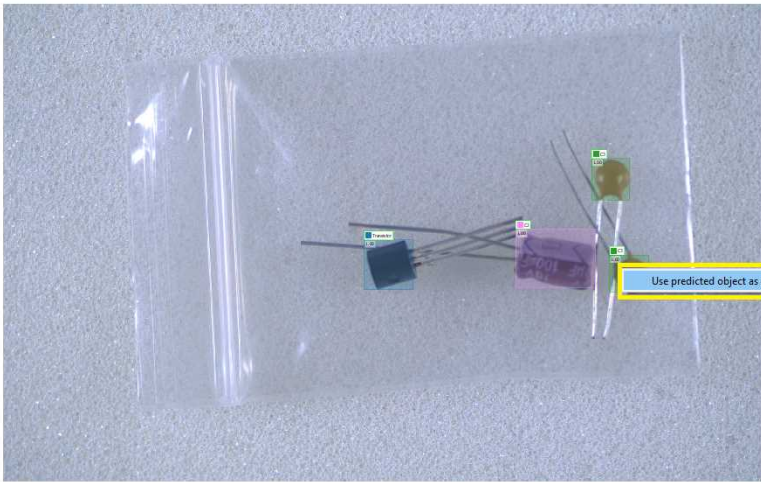
- If the image has a ground truth, check the **Missing ground truth object** option (CTRL + G) to display missing objects. They appear with a yellow pattern drawn over it.
- In the **Results** tab on the right side:
 - The list of detected objects shows their label, their score, whether they are matched to a ground truth object and their predicted bounding box.
 - To see how close a predicted object is to a ground truth object, select the object in the list on the right side of the image. The ground truth object is displayed on top of the predicted object with the following color code:



- To accept all the predicted objects as ground truth:
 - Click on the **Use prediction as ground truth** button (CTRL + U).
 - Note that it removes any previous ground truth object already present in the image.
- To accept a single predicted object as ground truth:
 - a. Right click on the object.
 - b. Select **Accept into ground truth** in the menu.

Viewing: Images_2 Show: 45,17 % Fit Original resolution Objects Results

Predicted objects Missing ground truth objects **Use prediction as ground truth** More...



	Label	Score	St
1	C3	1,00	Good
2	C2	1,00	Good
3	C3	1,00	Good
4	Transistor	1,00	Good

Processing time: 531 milliseconds

Use predicted object as ground truth

In the API

- To apply EasyLocate to an image, use `ELocator::Apply`.

This method returns an object `ELocatorResult`.

- Use `ELocatorResult::GetDetectedObjects` to retrieve the predicted objects as an array of `ELocatorPredictedObject`.
- Use `ELocatorResult::Draw` to draw all the predicted objects.
- For each predicted object, use:
 - `ELocatorObject::Width` to get its width.
 - `ELocatorObject::Height` to get its height.
 - `ELocatorObject::Label` to get its label.
 - `ELocatorPredictedObject::Probability` to get its predicted probability.

4.3. Validating the Results

The EasyLocate tool exposes 2 types of metrics. These object-based metrics quantify the performance of the tool :

- At the object level.
- At the image level. These metrics are related to the tool ability to correctly detect images without object and images with objects.

In Deep Learning Studio:

- The metrics are available in the **Dataset results** tab.
- Most metrics depends on the value of the **Detection threshold**.

Export results to CSV

Results for: Full dataset

Adapting the detection rule ⓘ

Detection threshold: 0,930796

Objects Images

Metrics ⓘ

Average precision: 0,98787 ⓘ

	Label weighted	ⓘ
Recall: 95.3051%	95.1773%	ⓘ
Precision: 96.8254%	97.2048%	ⓘ
F-Score: 0.959943	0.962456	ⓘ

Per label metrics

Label	# Detected	# Correct	# Bad	# Not detected	Recall
Transi...	177	163	14	2	98.79%
Diode	128	126	2	21	85.71%
C1	126	126	0	2	98.44%
C2	139	135	4	8	94.41%
C3	123	121	2	1	99.18%

In the API:

- The metrics are represented by an object `ESupervisedSegmenterMetrics`.

The object-based metrics

The object-based metrics are computed by matching actual, ground truth objects to detected objects. A ground truth object and a detected object are matched together if:

- They have the same label.
- Their overlap ("Intersection over Union") is higher or equal to the `Same label maximum overlap` parameter of the tool.
- There is no other ground truth that has a higher overlap with the detected object and there is no other detected object that has a higher overlap with the ground truth object.

The object-based metrics are:

- The `Average precision (AP)` is the average of the precision (proportion of detected objects that are matched to a ground truth objects) for different values of recall (true positive rate, proportion of ground truth objects that are matched to detected objects) obtained by varying the `Detection threshold`.
 - Its value is between 0 (bad detector) and 1 (good detector).
 - It is a standard metric for evaluating object detector.
- The `recall`, also called the "true positive rate", is the proportion of ground truth objects matched with a predicted object.
- The `weighted recall` is the weighted average of the `recall` for each label.
- The `precision`, also called the "positive predicted value", is the proportion of predicted objects matched with a ground truth object.
- The `weighted precision` is the weighted average of the `precision` for each label.
- The `F-Score` is the harmonic mean of the recall and the `precision`.
- The `weighted F-Score` is the weighted average of the `F-Score` for each label.
- The `Per label metrics` table shows various metrics of the objects of each label. The columns starting with a "#" indicate the number of objects in the corresponding category.
 - Selecting one or more cells from these columns filters the image list to show only the images that have objects falling in the corresponding categories.
 - If there is no selection, all the images are listed.
 - Use CTRL + Left Click to add cells to the current selection.

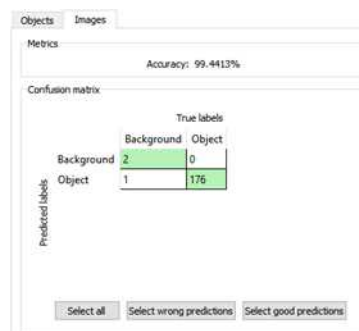
For example, the following selection shows the images containing either:

- A badly detected “transistor” object
- A detected (correctly or badly) “diode” object
- A correctly detected “C2” object
- An undetected ground truth “C3” object

Per label metrics

Label	# Detected	# Correct	# Bad	# Not detected	Recall
Transi...	177	163	14	2	98.79%
Diode	128	126	2	21	85.71%
C1	126	126	0	2	98.44%
C2	139	135	4	8	94.41%
C3	123	121	2	1	99.18%

The image-based metrics



The metrics related to the correct detection of the class of the images (background / with object or good / defective in the context of defect detection) are:

- The **image detection accuracy** is the proportion of images correctly predicted to have objects or not.
- A **Confusion matrix** listing the number of images in each category.
 - Selecting one or more cells of the confusion matrix filters the image list to show only the images in the corresponding categories.

4.4. Benchmarks for EasyLocate

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Capacity small

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	58	6.7 ms	78.91 ms	-
	4	123	2.39 ms	-	265
	16	453	1.05 ms	-	988
	64	1711	1 ms	-	3506
256 × 256	1	123	7.52 ms	375.43 ms	-
	4	383	4.49 ms	-	893
	16	1711	3.22 ms	-	3720
	64	6745	3.09 ms	-	14651
512 × 512	1	383	23 ms	2508 ms	-
	4	1429	13.69 ms	-	3365
	16	6745	11.12 ms	-	14651
	64	26878	-	-	53758

Capacity normal

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	63	7.2 ms	81.55 ms	-
	4	132	2.55 ms	-	279
	16	475	1.13 ms	-	1032
	64	1795	1.04 ms	-	3674
256 × 256	1	132	6.93 ms	393 ms	-
	4	408	4.30 ms	-	937
	16	1795	3.20 ms	-	3888
	64	7074	3.28 ms	-	15310

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
512 × 512	1	405	24.18 ms	2583 ms	-
	4	1513	12.5 ms	-	3532
	16	7074	11.24 ms	-	15310
	64	28191	-	-	56386

Capacity large

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	196	12.04 ms	141 ms	-
	4	327	3.65 ms	-	726
	16	893	1.85 ms	-	1938
	64	3201	1.33 ms	-	7065
256 × 256	1	334	38 ms	684 ms	-
	4	857	9.17 ms	-	2159
	16	3201	5.46 ms	-	6699
	64	12434	-	-	26030
512 × 512	1	851	34.6 ms	4266 ms	-
	4	2952	22.47 ms	-	6174
	16	12434	-	-	26030
	64	49367	-	-	98737

PART VI
3D PROCESSING TOOLS

1. Easy3D - Using 3D Toolset

1.1. Basic Concepts

Easy3D

Easy3D is a set of tools for solving computer vision problems 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.

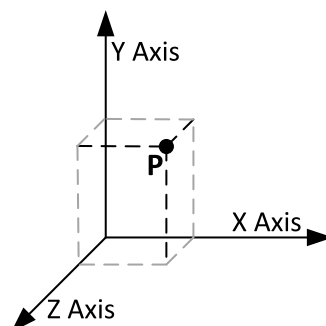
Easy3D provides a calibration tool to generate corrected, metric point clouds and meshes from depth maps. Most 3D operators work on point clouds or meshes. The included export functions to the standard PCD file format allows integration with other 3D tools.

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.

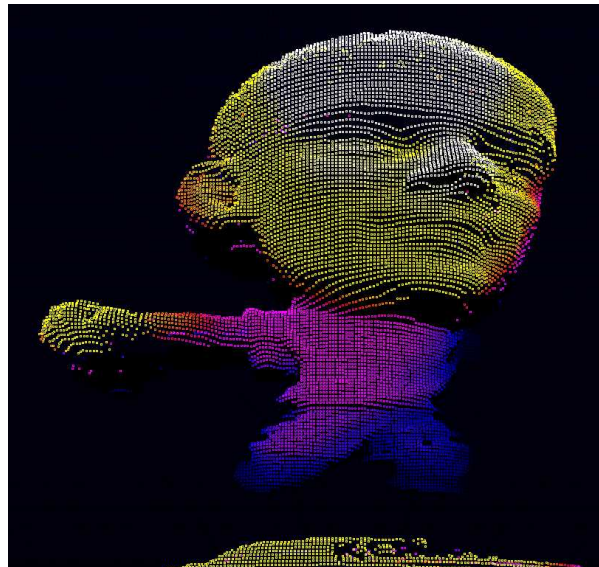
A depth map contains a gray scale image coded on 8, 16 or 32 bits per pixel.

- 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 (-3.402823×10^{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

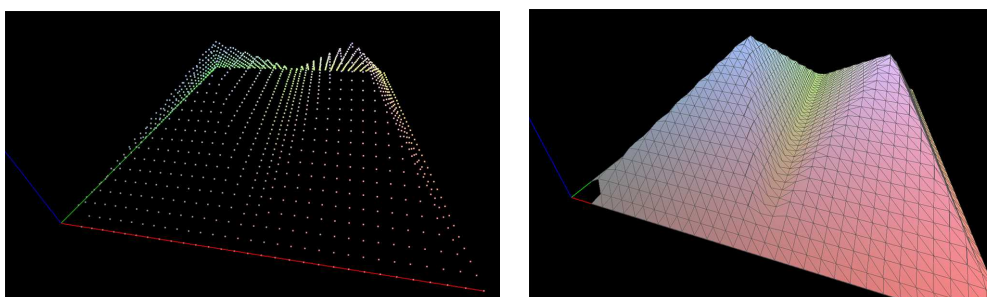
A Mesh is a geometric representation of a 3D surface, a set of connected 3D points.

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.

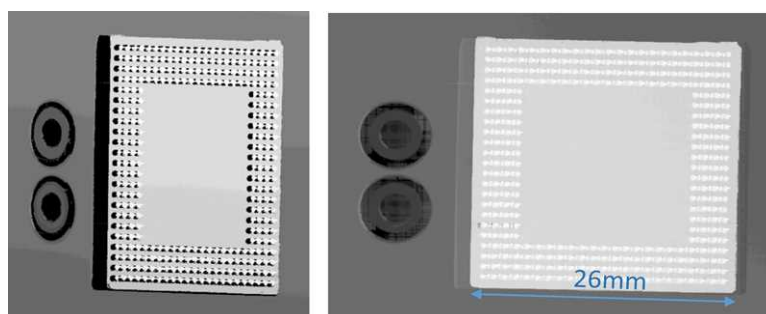
EMesh uses a metric space representation that can be generated from a depth map and that can be used to produce a ZMap.

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 orthogonal projection of a point cloud or a mesh onto an arbitrary 3D reference 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()`.

1.2. Static Methods

EFilters class

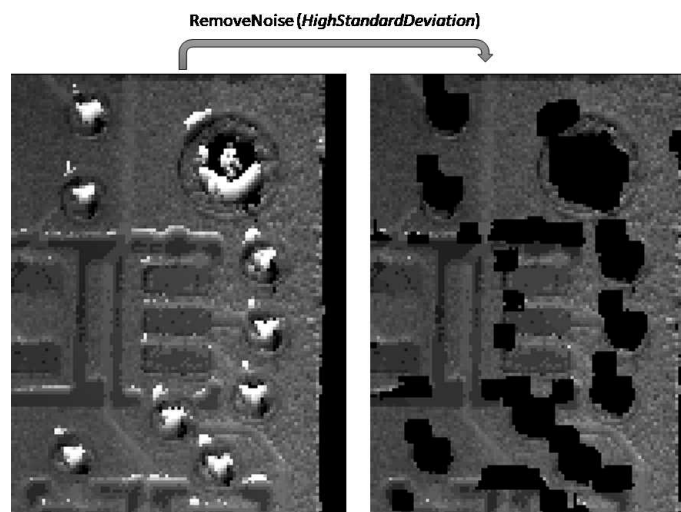
The `EFilters` class contains static methods used to apply filters to ZMaps or depth maps.

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 $(2 \times \text{halfKernelSize} + 1)$ 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
EZMap-6 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
EZMap-6 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 $(2 \times \text{halfKernelSize} + 1)$ where `halfKernelSize` is a parameter of the method.

ComputeStandardDeviationMap

The `ComputeStandardDeviationMap()` method computes a map of the local standard deviation.

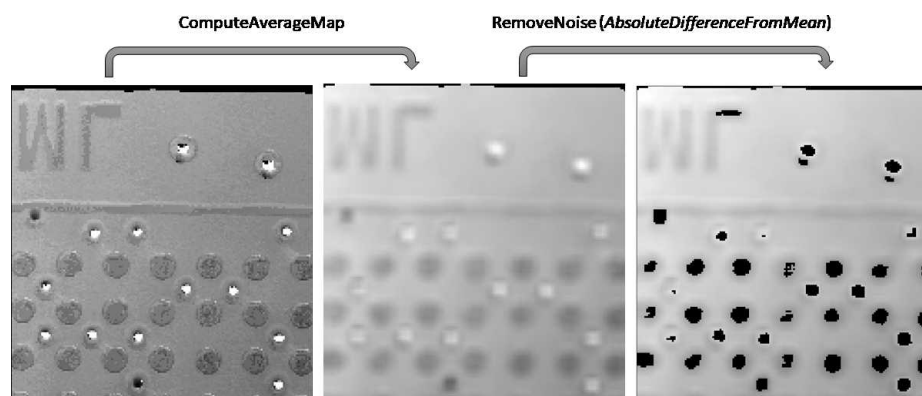
- You can use this method to visually determine the threshold value to use with the `RemoveNoise()` method when using the `ENoiseRemovalMethod_HighStandardDeviation` setting.



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 a 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
EZMap 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
EZMap 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 3x3, the threshold is 20.0
EZMap filteredZMap;
filteredZMap.SetSize(zmap);

EFilters::RemoveNoise(averagedZMap, filteredZMap, ENoiseRemovalMethod_AbsoluteDifferenceFromMean, 5, 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 to the Z resolution.

Use an `ERegion` object to specify the region of the ZMap or depth map used to compute the statistics.

1.3. Point Cloud

Mapping Attributes

- In addition to the (x, y, z) coordinates, you can store other components in an `EPointCloud` such as normals, colors, intensities, textures, indexes, confidences and other custom attributes.
- When you use additional components, a mapping exists between each 3D point and an attribute and the different operations performed on the point cloud conserve this mapping.
- To add components to an `EPointCloud`, use the functions `FillAttributeBuffer` or `AddCustomAttributeBuffer`.
 - You can also save and load these attributes from point cloud files.
 - Several file formats are supported, but we recommend the formats `PCD` and `PLY` for handling additional components.

Coordinates Transformations

Geometric transformations

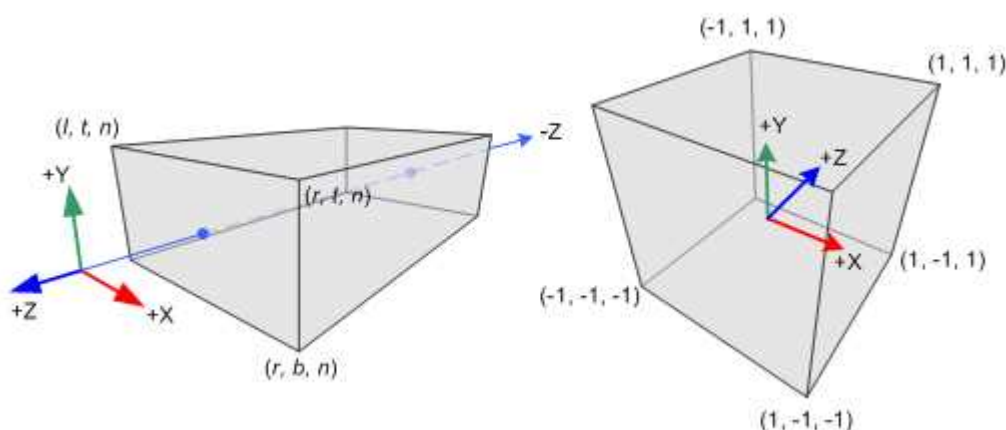
Transforms allow you to reposition the point cloud inside the 3D space.

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 allows you to simulate the perspective effect given by an eye or a camera.

- Use the class `E3DTransformMatrix` to create the geometric transformations. The class `EAffineTransformer` applies a transformation defined by a `E3DTransformMatrix` to a point cloud.

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 produce a new point cloud with the selected points.

Decimation

- Open eVision offers 2 capabilities to decimate an `EPointCloud`:
 - `ERandomDecimator` decimates the cloud by copying a specified number of points, randomly selected, to a new point cloud.
 - `EGridDecimator` splits the space using a grid of given size; the new cloud is created by averaging the points of every grid together, resulting in a regularly sampled cloud.
- To use `ERandomDecimator`, specify the number of points to keep as a parameter of the constructor.

```
EPointCloud pc;
pc.LoadPCD("c:\\images\\data.pcd");

// Explicitly decimate the point cloud

ERandomDecimator decimator(5000);
EPointCloud pcDecimated;

decimator.Decimate(pc, pcDecimated);
pcDecimated.SavePCD("c:\\images\\decimatedData.pcd");
```

- To use `EGridDecimator`, specify the cell size, either as a `E3DPoint` or as a float if cell is cubic.

```
EPointCloud pc;
pc.LoadPCD("c:\\images\\data.pcd");

// Explicitly decimate the point cloud

EGridDecimator decimator(=0.f);
EPointCloud pcDecimated;

decimator.Decimate(pc, pcDecimated);
pcDecimated.SavePCD("c:\\images\\decimatedData.pcd");
```

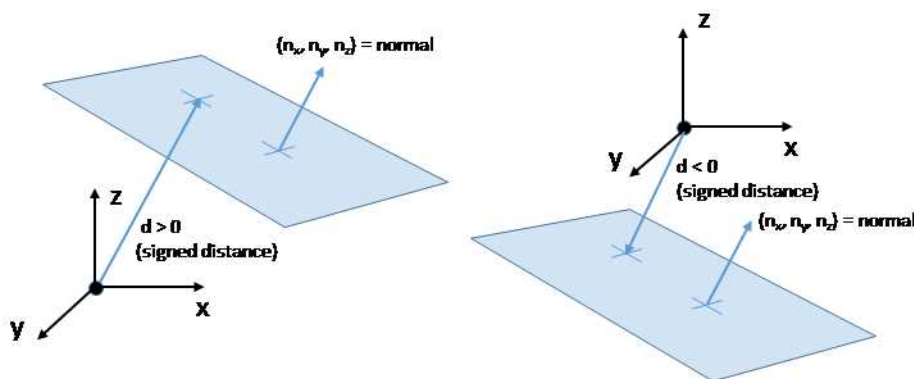
Managing Planes

E3DPlane

A plane can be represented as an [E3DPlane](#) object.

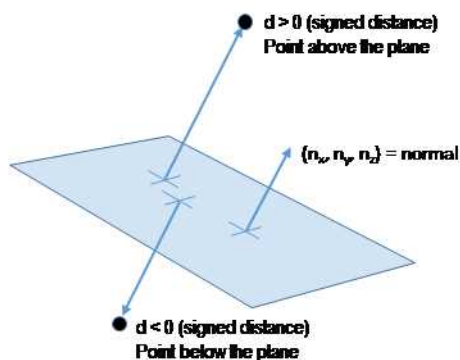
This plane is characterized by:

- 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.



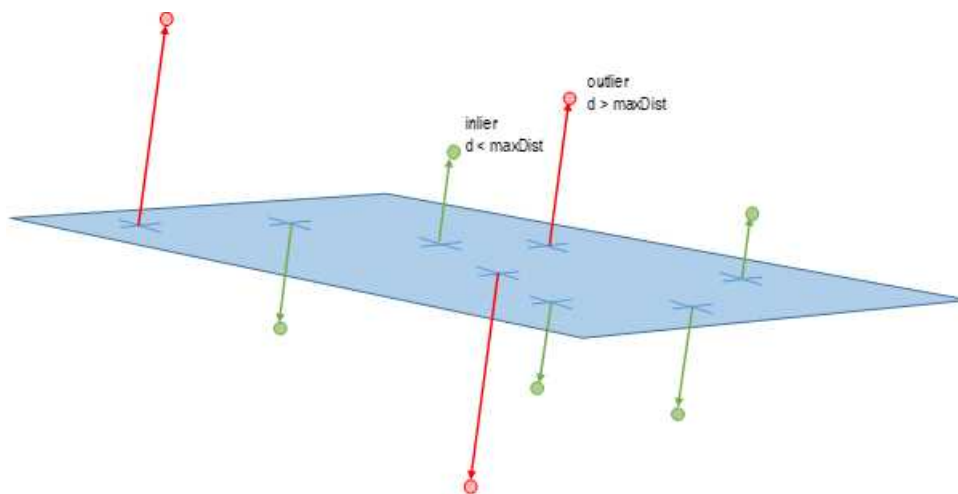
EPlaneFinder

You can search for a plane in a point cloud using the object [EPlaneFinder](#) object.

The main parameters of this object are:

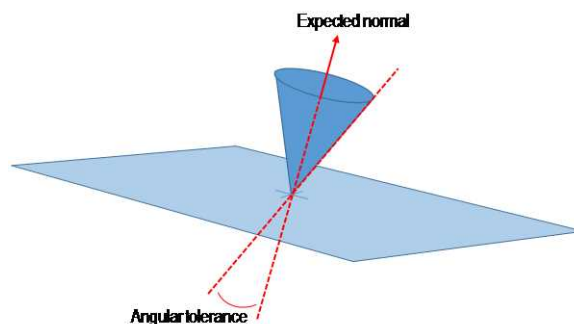
- The maximum distance between the searched plane and a point that belongs to this plane.
- The expected ratio between the numbers of inliers and the total number of points in the point cloud.
 - An **inlier** is a point that belongs to a plane (closer than this maximum distance).
 - An **outlier** is 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.
 - It should also take the 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. This means that we estimate that about 30% of the points belong to the plane.
 - This parameter is not as critical as the maximum distance.
 - It affects the maximum time that the algorithm spends to search a plane and its robustness.
- For more fine-grained control, you can specify the ratio of inliers as a range.
 - The min of the range is the minimum ratio of inliers for a plane to be considered as valid.
 - The algorithm stops searching for a plane when it finds one with the max of the range inliers.
 - The bigger the min of the range and the smaller the max, the faster the algorithm is.
 - Specifying the range as a single value x is equivalent to setting a range of $[x/2, x]$.

- You can specify the expected normal vector to the searched plane.
 - Specify also an angular tolerance with respect to this direction.
 - The algorithm only searches for a plane that satisfies the condition.
 - Set this condition may speed up the plane search.



- You can specify 1 or 2 points contained in the searched plane.
 - These points are not specified as inliers (points closer to the plane than its maximum distance) but as points exactly contained in the main plane.
 - They may not be exactly in the final plane after the final fitting step detailed below.
 - The algorithm only searches for a plane that satisfies the condition.
 - Set this condition may speed up the plane search.

Once the main plane is found, a fit is done on all the inliers points and the result is returned (see [EPlaneFitter](#) below).

Decimation

By default, the [EPlaneFinder](#) decimates the input point cloud to accelerate the search.

- The default decimator reduces the input point cloud to 10,000 points.
- You can disable this decimation.
- You can change the number of points the cloud is reduced to.
- You can decimate a point cloud explicitly.
 - Use an [ERandomDecimator](#) object.
 - Use the decimated point cloud as input for the [EPlaneFinder](#).
 - Disable the default decimator.

EPlaneFitter

The [EPlaneFitter](#) operator computes a fit on all the points of a point cloud and returns a [E3DPlane](#) object. The “average” plane, that minimizes the orthogonal distance of the points to the plane, is returned.

Aligning

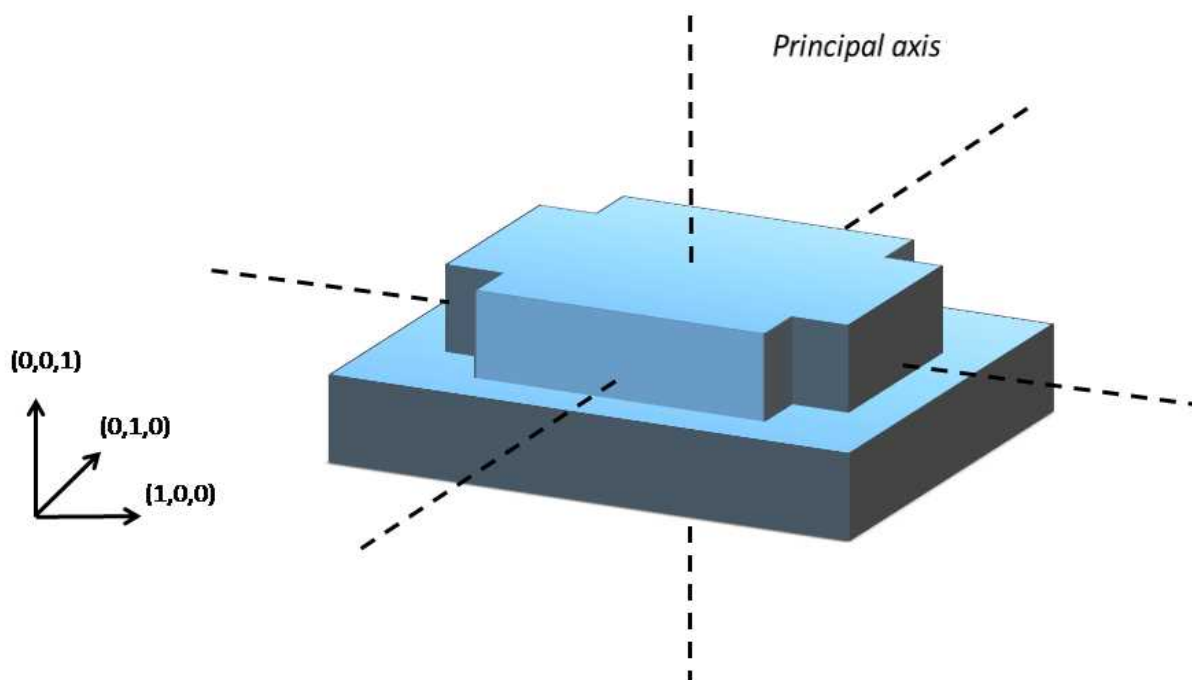
EPrincipalAxisExtractor

The `EPrincipalAxisExtractor` computes the “principal axis” of an object from a point cloud (`EPointCloud`) and returns a `E3DTransformMatrix` 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 axes are oriented along the “principal axis” of the object. This is the result of the “PCA” calculation (principal axis analysis).
- The directions of the axes 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 axes of an object.



The principal axes 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 axes.

You can use the returned `E3DTransformMatrix` 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.

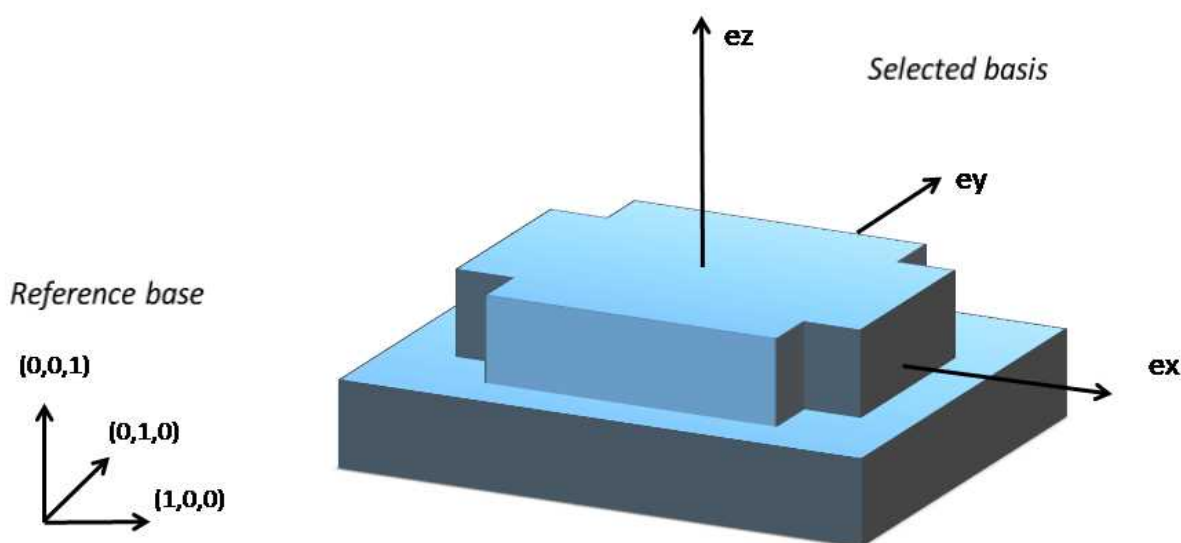
Specification of a reference transformation

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 axes. 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 axes **ex**, **ey** and **ez**.



EFeaturesAligner

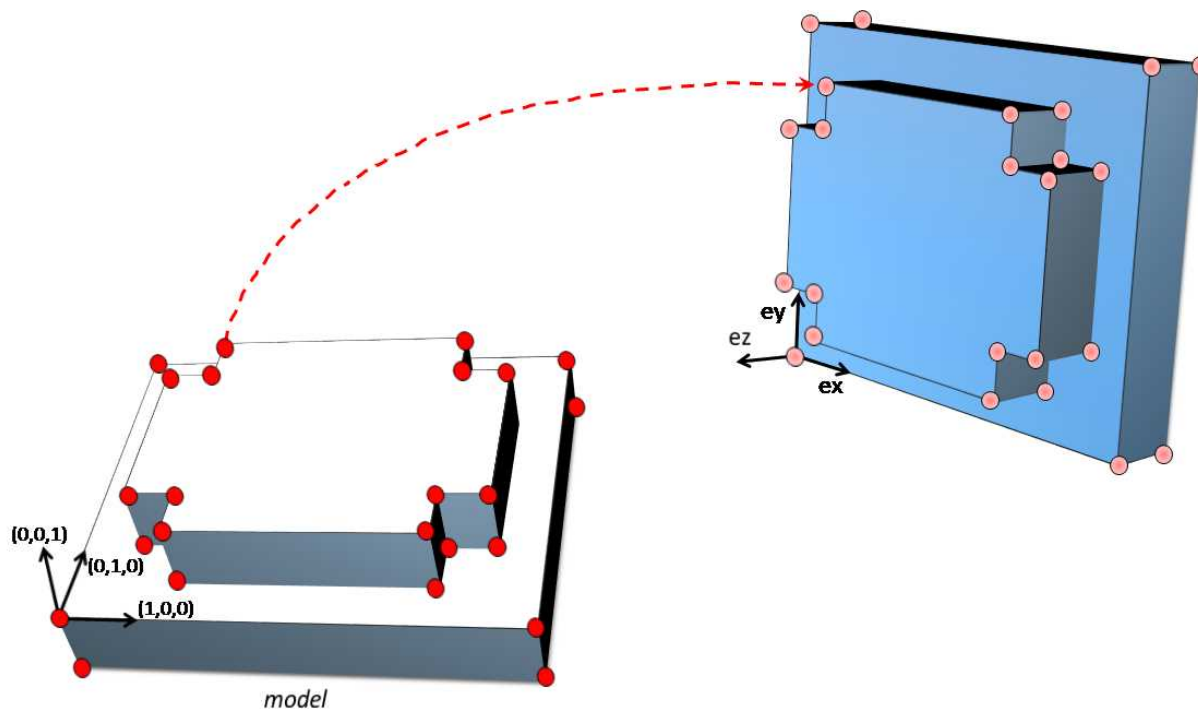
A [EFeaturesAligner](#) object finds the best transformation that maps a list of points to another list of points.

- 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 (\mathbf{ex} , \mathbf{ey} , \mathbf{ez}) and the center point \mathbf{t} that defines the new basis.

You can also apply the computed transformation on any 3D point as illustrated in the code below.

```
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 P<sub>w</sub>, results in point P<sub>b</sub>
E3DPoint P<sub>w</sub> = E3DPoint(...);
E3DPoint P<sub>b</sub> = alignBase*P<sub>w</sub>;
```

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.

1.4. 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_\(file_format\)](https://en.wikipedia.org/wiki/STL_(file_format))) using the `SaveSTL()` and `LoadSTL()` methods which respectively write to and read from ASCII or binary STL files.

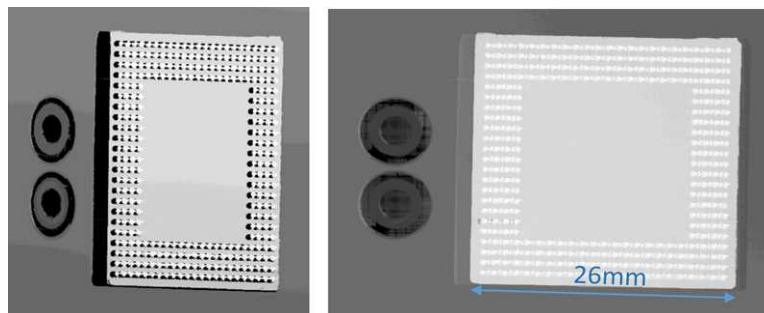
You can use an `EMesh` to produce a ZMap (see "[Generating a ZMap](#)" on page 268). Because an `EMesh` represents a surface, the so generated ZMap can show better continuity and less undefined pixels.

1.5. 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 grayscale 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.

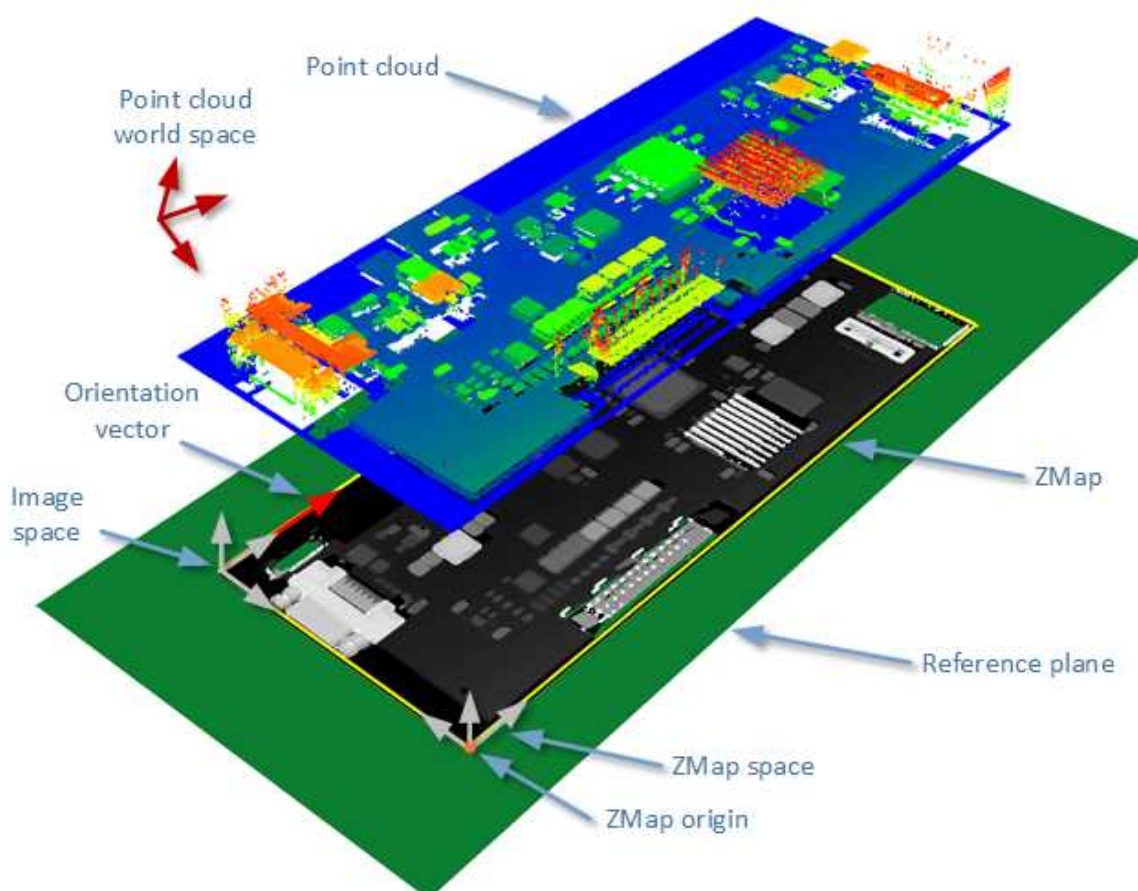
The orientation vector allows you to “rotate” the object around the normal of the reference plane.

- [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 plane 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 set up 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.

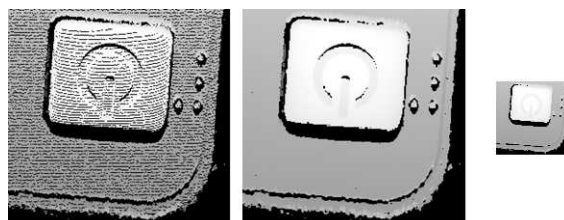


The projection of a point cloud on a ZMap, showing 3 coordinate systems: the world space, the ZMap space and the pixel space.

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

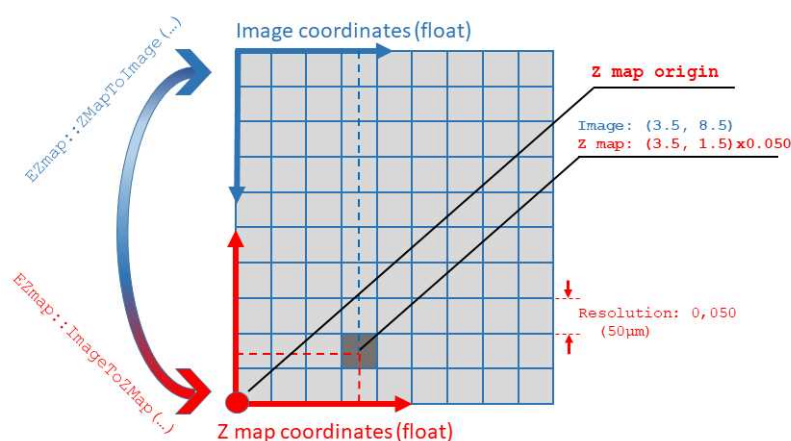
Coordinate systems on a ZMap

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 of 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 access 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`.

1.6. 3D Viewer

The class `E3DViewer` is an interactive 3D viewer for point clouds, ZMaps and meshes.

- It features multiple sources display, color ramps, 3D point picking and text label display.
- It is compatible with Windows and Linux and can be integrated to Win32, MFC and QT frameworks.

Creating a 3D viewer

- The general constructor of a 3D viewer is:
`E3DViewer(EUIAPI uiApi, int orgX, int orgY, int width, int height, int parent)`
- The way to use it depends on the Operating System and the User Interface API.

Windows only

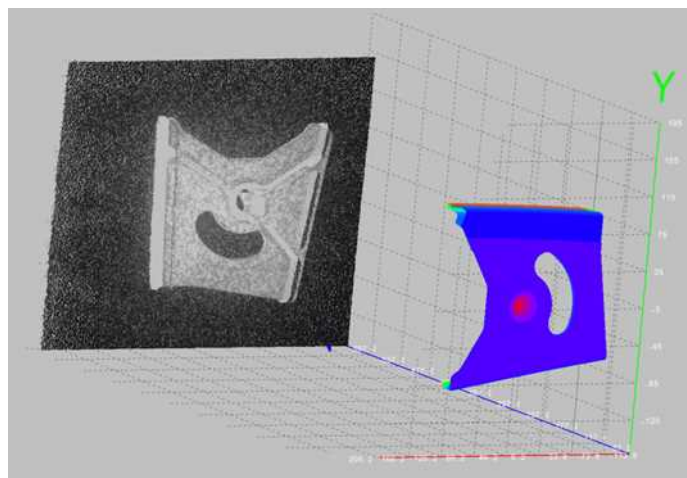
- To create a 3D viewer stand-alone window:
 - Use `E3DViewer(EUIAP::EUIAPI_Win32, orgX, orgY, width, height)`
- To create a 3D viewer as a part of another window in an MFC application:
 - Use `E3DViewer(EUIAP::EUIAPI_Win32, orgX, orgY, width, height, handle to the parent window)`
 - See the MsVc 3DViewer sample.

Windows and Linux

- To create a 3D viewer in a Qt application:
 - Use `E3DViewer(EUIAP::EUIAPI_Qt, orgX, orgY, width, height)`
 - You must instance the class inside a `QOpenGLWidget` object.
 - See the 3DViewer Qt sample.

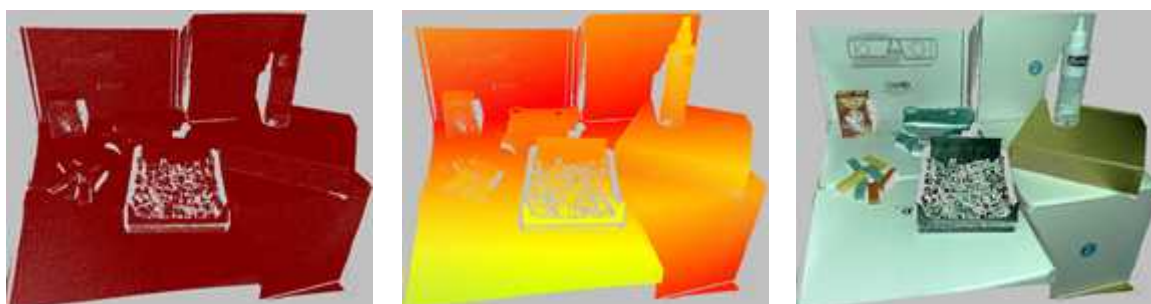
Managing the render sources

- A render source is a displayed entity. It can be an [EPointCloud](#), an [EZMap](#) or an [EMesh](#). You can display one or several render sources simultaneously in the 3D viewer.

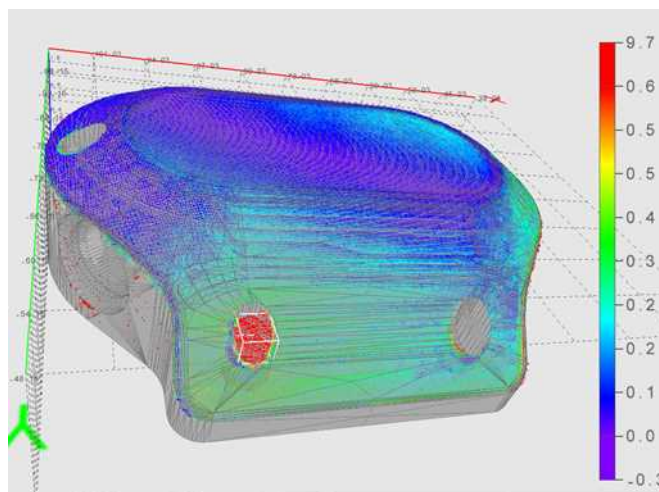


A point cloud displayed in gray scale and a mesh in false colors in the 3D viewer

- To manage the list of render sources, use the methods:
 - [AddRenderSource](#) to add another render source to the current list. The render source has a name for further reference.
 - [SetRenderSource](#) to change the content of the render source.
 - [RemoveRenderSource](#) to remove a render source from the current list.
- The render sources API exposes several display attributes:
 - Visibility (controlled by [ShowRenderSource](#)/[HideRenderSource](#))
 - Color mode ([SetRenderSourceColorMode](#)): choose between constant color, color ramp or point cloud color attributes.
 - Opacity ([SetRenderSourceOpacity](#))
 - Point size ([SetRenderSourcePointSize](#)): applies to point clouds and ZMaps only.
 - Wire frame ([SetRenderSourceWireFrame](#)): applies to meshes only.



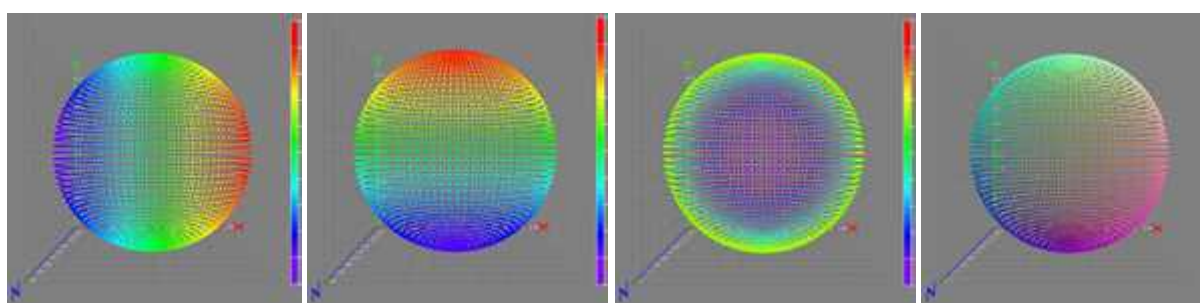
A point cloud displayed with constant color, color ramp or color attributes (data courtesy of Zivid)



A mesh with wireframe and transparency, combined with a point cloud with color ramp

Using a color ramp

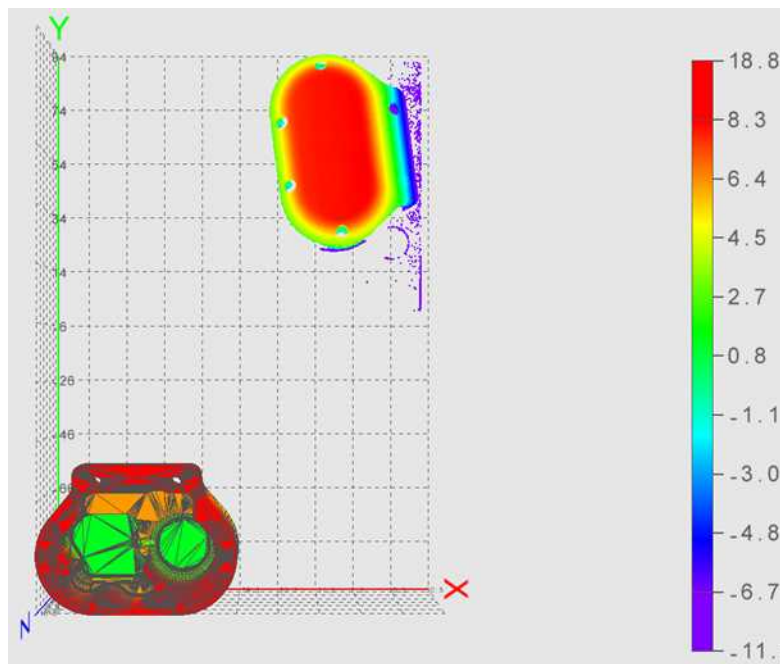
- When the color mode of a render source is `ESourceColorMode_Ramp`, the color of each point is calculated from the position or the attribute of the point.
- Use `SetColorRampMode` to choose the color ramp:
 - `EColorRampMode_HueFromX/Y/Z` computes the colors from respectively X/Y/Z point coordinates (`EColorRampMode_HueFromZ` is the default color ramp mode).
 - `EColorRampMode_RGBCube` computes the colors by mixing X,Y,Z point coordinates.
 - `EColorRampMode_HueFromIntensity` computes the colors from the intensity attribute of the point.
 - `EColorRampMode_HueFromConfidence` computes the colors from the confidence attribute of the point.
 - `EColorRampMode_HueFromDistance` computes the colors from the distance attribute of the point.



Color ramp modes Hue from X/Y/Z and RGB cube

- When a color ramp is defined, you can display a legend at the right side of the window (default position). To control the color ramp legend aspect, use the methods `Show/HideColorRampLegend`, `SetColorRampGraduationColor` and `SetColorRampLocation`.

- When the “Smart color ramp” is enabled with the method `SetEnableSmartColorRamp`, an outlier filtering processing is applied to remove the noise and spread the colors on the main part of the object. The outliers are then displayed with constant red or blue colors.



- A color ramp `EColorRampMode_HueFromZ` with outlier removal process:
- The extreme points with Z coordinate between 8.3 and 18.8 are drawn in red
 - The Z coordinate of 98% of the points are between -6.7 and 8.3

Interactive controls

On Windows, the interactive controls are built in the class `E3DViewer`.

- The following interactions are possible:

Intercation	Control
Rotate the view	left-click + mouse move
Translate the view	right-click + mouse move
Change the view distance	mouse wheel
Reset the view	r
View along the positive / negative X axis	x / shift+x
View along the positive / negative Y axis	y / shift+y
View along the positive / negative Z axis	z / shift+z
Show / hide the axis	a
Enable / disable the wireframe mode	w
Increase / decrease the point size	plus sign (+) / minus sign (-)

- Use the following methods to implement custom view controls:
 - [LockRotationInitialPosition](#), [UpdateRotationPosition](#) and [LockRotationFinalPosition](#) correspond to the sequence click-drag-release to change the viewpoint by rotation.
 - [LockTranslationInitialPosition](#), [UpdateTranslationPosition](#) and [LockTranslationFinalPosition](#) correspond to the sequence click-drag-release to change the viewpoint by translation.
 - [UpdateViewDistance](#) changes the view distance, usually controlled by the mouse wheel.
 - [ResetView](#) restores the default viewpoint.

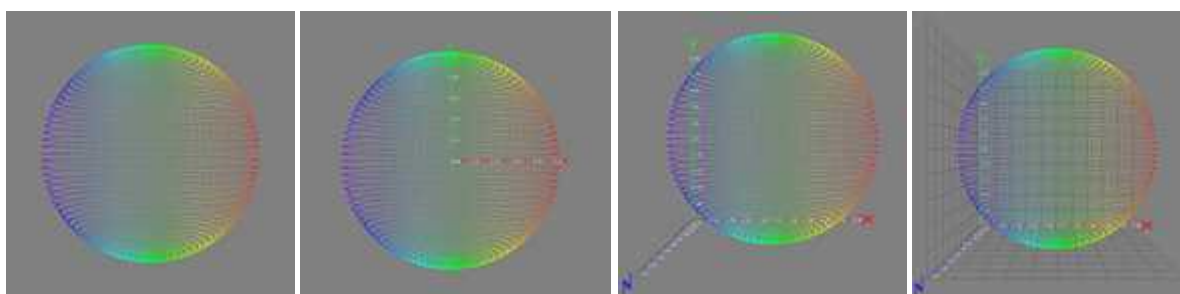
See the [Qt 3DViewer](#) sample for a use case of this view control API.

- You can also directly configure the view position with the methods:
 - [SetViewTarget](#) (by default, it is the center of the object).
 - [SetViewAngle](#) to choose the orientation of the view.
 - [SetViewDistance](#) to choose the distance to the view target.

View parameters

You can customize the 3D view and:

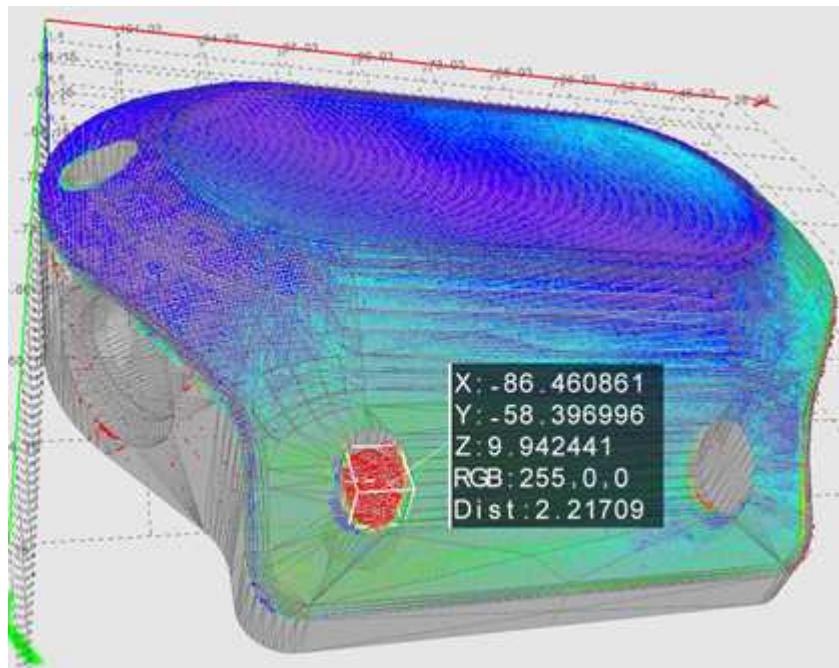
- Change the field of view with [SetFieldOfView](#).
- Switch between the perspective and the orthographic view with [SetProjectionType](#).
- Enable or disable the display of the X, Y and Z axis with [SetRenderAxis](#).
- Switch the axis origin between the world origin and the object center with [SetAxisOrigin](#).
- Enable or disable the display of a grid with [SetRenderGrid](#).
- Activate an auto rotate animation with [SetAutoRotate](#).
- Use a decimation level (remove some points to speed up the rendering) with [SetRenderDecimationLevel](#).



No axis / world centered axis / bounding box axis / axis with grid

Picking a 3D point

- Picking a point means detecting the point closest to the given coordinates in a [E3DViewer](#) window. You can then display the detected 3D point, with attributes, as a text label.

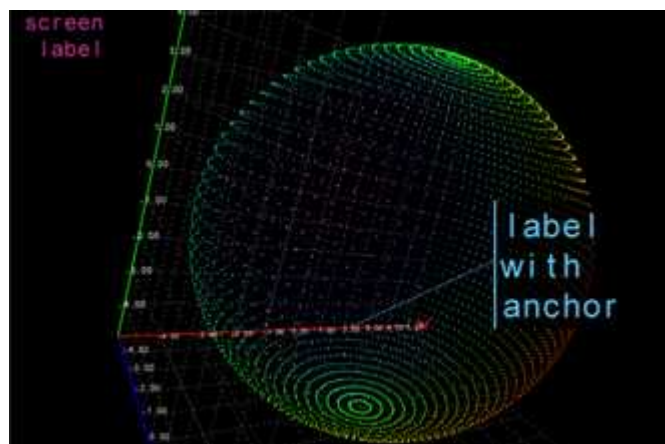


Displayed coordinates and attributes of a picked point on the 3D view

- The distance threshold used to select a picked point is defined by [SetPickingDistanceThreshold](#). There is no picked point if the point cloud distance to the picked position is greater than this threshold.
- On the Win32 interface framework, the built-in control for the picking is **ctrl + left-click**. You can also control the picking with the methods: [Pick3DPoint](#), [GetLastPickedPoint](#) and [ResetPicking](#).
- To configure the display of the picking label, use [SetPickingDisplay](#), [SetPickingLabelSize](#), [SetPickingLabelColor](#) and [SetPickingLabelFixed](#).

Text labels and 3D objects

- You can add custom text labels and 3D objects to the current view of the 3D viewer.

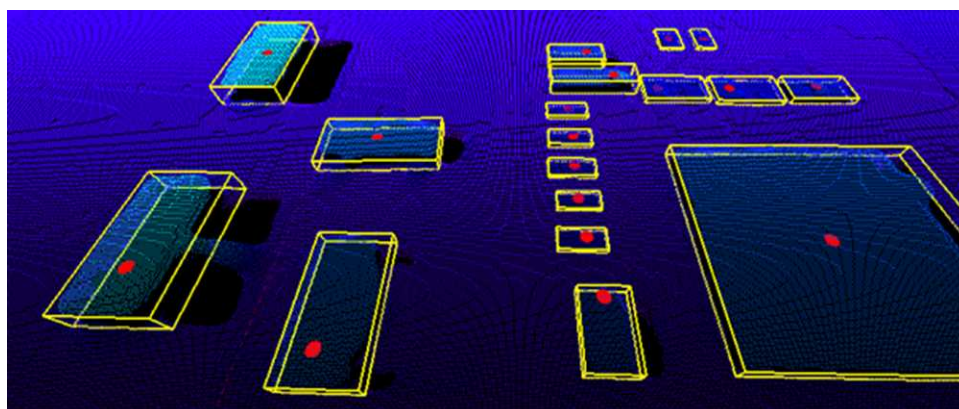


A screen label in the top left corner and a text label with 3D anchor

- To control the text label display, use:
 - [AddTextLabel](#) to add a text label with or without a 3D anchor. [AddTextLabel](#) returns an ID used for further reference.
 - [EditTextLabel](#) to change the position, color, size or text of a label.
 - [GetTextLabel](#) to get the attributes of a label.
 - [RemoveTextLabel](#) to remove a label.
 - [ClearTextLabels](#) to remove all labels.
- The class [E3DViewer](#) can also display [E3DObject](#) over a point cloud, a ZMap or a mesh. Use the tools [Easy3DObject](#) and [Easy3DMatch](#) to create the [E3DObjects](#).
- Use the methods [Register3DObjects](#) and [RemoveCurrent3DObjects](#) to manage the list of [E3DObjects](#) that you want to display.

An [E3DObject](#) contains several features (center point, bounding box, base plane...). Use the methods [Show / HideFeatureFor3DObject](#) and [Show / HideFeatureForAll3DObjects](#) to select the displayed features.

See the [3DObjectExtraction](#) MSCV sample as an example for the display of [E3DObjects](#).



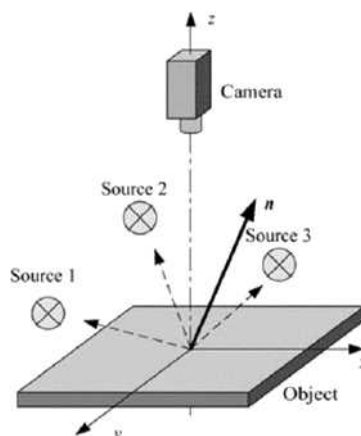
3D objects drawn with a point cloud displaying the bounding boxes and the top positions

1.7. Photometric Stereo

Photometric Stereo and Process

Introduction

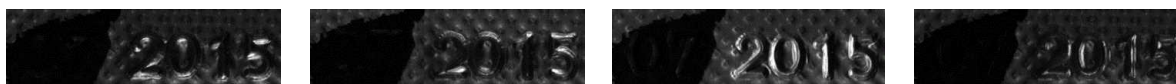
The Photometric Stereo is a technique used to estimate the normals at the surface of an object.



Photometric stereo setup

Source: https://www.researchgate.net/figure/Principle-of-photometric-stereo_fig7_222422584

- It takes at least 3 images of the same object taken under different known light directions.



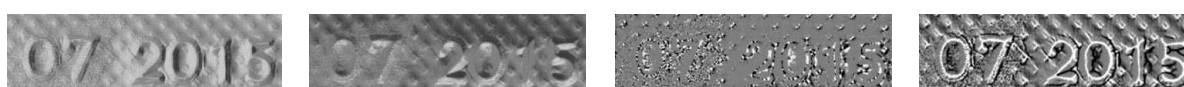
Inputs: images with different light directions

- It produces an image containing the fraction of light reflected (called *albedo*) and the normal of the surface at each pixel.



Outputs: albedos - normals

- The normals are processed to compute gradients and curvatures, allowing to easily see bumps and holes.

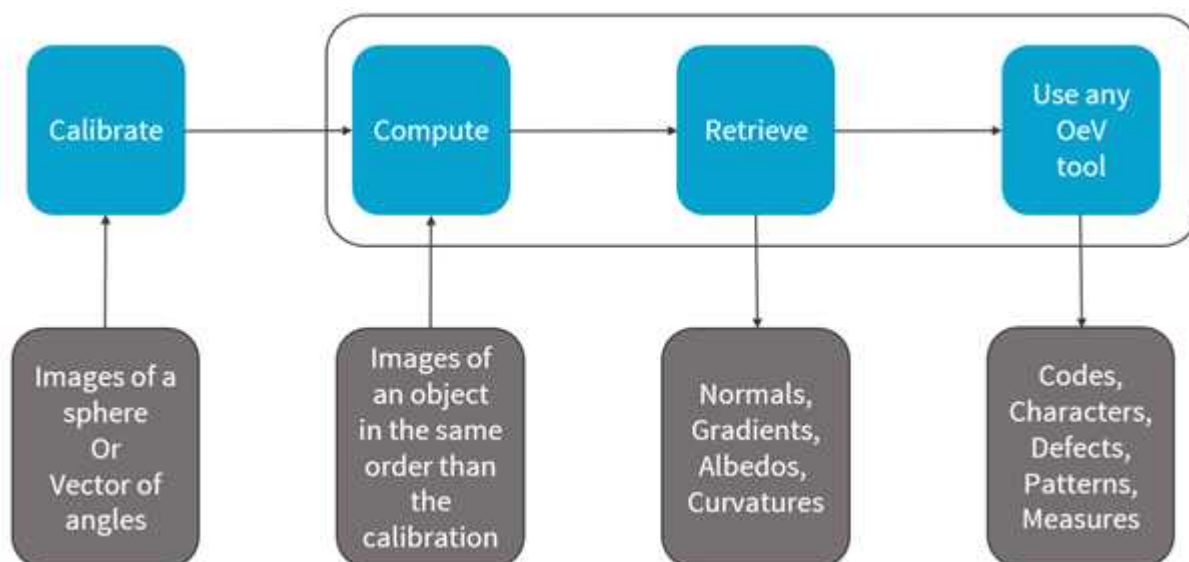


Outputs: gradients X - gradients Y - gaussian curvatures - mean curvatures

Process

You can use the object [Easy3D::PhotometricStereoImager](#) in a 4-step process:

1. Calibrate the setup from a sphere or from predefined angles (once per setup).
2. Perform the photometric stereo computation on the object images.
3. Retrieve the results.
4. Use and apply the Open eVision tools on the results.



Photometric stereo process

Resources

- The example described here demonstrates how to perform photometric stereo with Open eVision 3D libraries and tools.
- A sample application is also distributed with the source code. You can find it in `... \Sample Programs\MsVc samples\PhotometricStereo`.
- This example and the sample application are based on the following resources:
 - Open eVision 2.15
 - Microsoft Visual Studio 2017



NOTE

The license for Easy3D is necessary to use the photometric stereo tools.

Calibration

- You can perform the calibration either:
 - By setting the calibration angles.
 - By computing the calibration angles from images of a (hemi)sphere.

Azimuth and elevation

- To define a light direction, two angles are necessary, the **azimuth** and the **elevation**.
- When facing the image, the X-axis points right, Y points top and Z points towards the camera.
- The **azimuth** angles are oriented trigonometrically around the Z-axis.
 - A light source on the right of the image has an azimuth of 0°.
 - A light source on the top of the image has an azimuth of 90°.
- The **elevation** is the angle formed by the base plane and the light source.
 - A light source on the horizon has an elevation of 0°.
 - A light source on the camera has an elevation of 90°.

Code snippet

- The following code snippet shows how to perform the calibration from a (hemi)sphere.

```
EPhotometricStereoImager photometricStereo;

std::vector<EImageBW8> calibrationImages;
// Load calibration images (Todo)

std::vector<EROIBW8> calibrationROIs;
// Set the calibration ROIs (Todo)// Calibrate
float score = photometricStereo.CalibrateFromSphere(calibrationROIs);
```

- If the sphere is not detected, the calibration fails and generates an **EException** (**EError** **EError_SphereDetectionFailed**).

In that case, you can pass the position of the circle to the method:

```
ECircle circle;

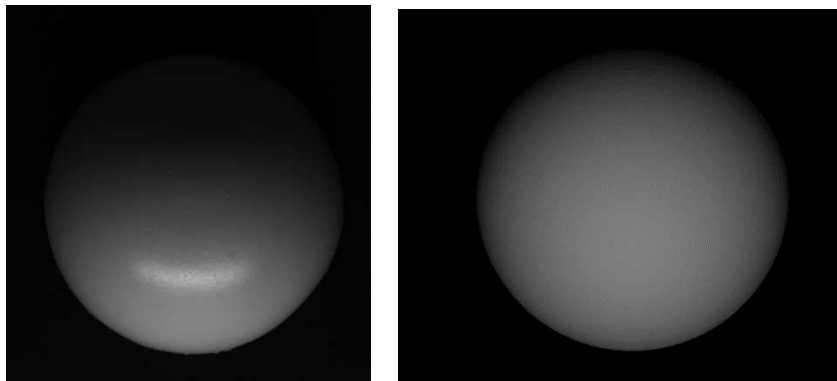
// Define circle (Todo)

photometricStereo.CalibrateFromSphere(calibrationROIs, circle)
```

- The method returns a score that indicates the reliability of the calibration.
 - The higher the value, the better the calibration.
 - The score range is [0, 1].
 - The scores above 0.75 are considered as good.
 - The scores below 0.50 are considered as bad.
 - The scores in between are considered as acceptable.
 - The method never fails. A bad score does not mean that you will not get good results on your images. It means that, if you do not, it is probably due to the calibration.

- The score is composed of 2 factors:
 - The lambertian (matte) of your sphere (the more lambertian the better).
 - The plausibility of the detected light directions.

The following figure shows the scores for 2 examples.



2 images from different spheres:
 left: a high reflection and a score of 0.70 - right: perfectly lambertian and a score of 0.96

- You can also directly set the calibration angles and retrieve them.
 Use `Easy::SetAngleUnit` to define the angle unit.

```
Easy::SetAngleUnit(EAngleUnit_Degrees);
std::vector<float> azimuths, elevations;
// Define the values of the angles (Todo)
photometricStereo.SetCalibrationAngles(azimuths, elevations);
```

Computation and Results

Computation

- Once the calibration is done, you can perform the photometric stereo computation on the object images.

```
std::vector<EImageBW8> objectImages;
// Load object images in the same order than the calibration images/angles (Todo)
std::vector<EROIBW8> objectROIs;
// Set the object ROIs (Todo)
// Compute
photometricStereo.Compute(objectROIs);
```

- You can also use an [ERegion](#).

```

ECircleRegion circle;

// Define the ERegion (Todo)

// Compute
photometricStereo.Compute(objectROIs, circle);
    
```

- The computation time is proportional to the number of pixels in the image.

To reduce this time, you can:

- Set a smaller ROI.
- Use an [ERegion](#).
- Use several threads.

The following table shows the computation time in different configurations (when computing albedos, mean and gaussian curvatures with high contrast, see below).

Number of lights	Image size	Number of threads	Computation time (s)
4	4096 × 3072	1	1.473
		4	0.596
4	1024 × 768	1	0.092
		4	0.039

↙ /2.5

↙ /16

↙ /2.35

Retrieving the results

- Use the method `Get...` or `Compute...` of [PhotometricStereoImager](#) to retrieve your results.

```

// Retrieve the results
EImageC24 normals = photometricStereo.GetNormals();

EImageBW8 albedos = photometricStereo.GetAlbedos(Easy3D::EPhotometricStereoContrast_HighContrast);

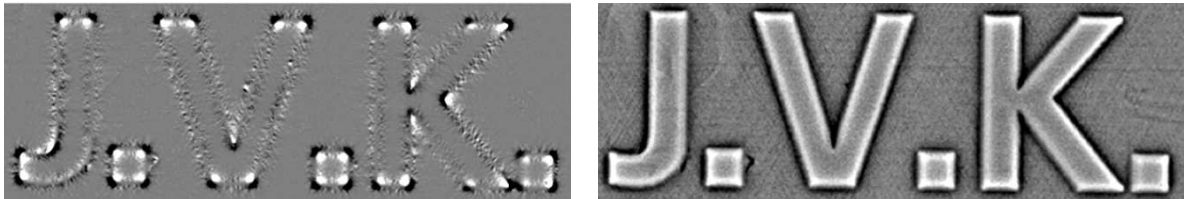
EImageBW8 gradientsX = photometricStereo.GetGradientsX();
EImageBW8 gradientsY = photometricStereo.GetGradientsY();

EImageBW8 gaussianCurvatures = photometricStereo.ComputeGaussianCurvatures(Easy3D::EPhotometricStereoContrast_HighContrast);

EImageBW8 meanCurvatures = photometricStereo.ComputeMeanCurvatures(Easy3D::EPhotometricStereoContrast_HighContrast);
    
```

- The `normals` [EImageC24](#) represents the x,y,z normals of the surface at each pixel.
 - A RGB pixel intensity of (0, 128, 255) corresponds to a x,y,z normal of (-1, 0, 1)
- The `albedos` [EImageBW8](#) represents the fraction of light reflected at each pixel.
 - Compared to the input image, the albedo is independent of the lighting direction and intensity.
 - The albedos are normalized to the full image range.
- The `gradientsX` and the `gradientsY` [EImageBW8](#) represent the gradients of the surface along the X- and Y-axis.
 - The gradients are clipped to +/- 3.715 before being mapped to the full image range.

- The `gaussianCurvatures` and the `meanCurvatures` `EImageBW8` represent the local curvature of the surface at each pixel.
 - The gaussian curvatures are important when the curvature of the object is big in 2 orthogonal directions.
 - The mean curvatures only need an important change in 1 direction.
 - You can think of the gaussian and the mean curvatures as a corner detector and an edge detector.



The gaussian curvature highlights the corners and the mean curvature highlights the edges

`GetAlbedos`, `ComputeMeanCurvatures` and `ComputeGaussianCurvatures`

Most of the computations are done in the method `Compute`, however:

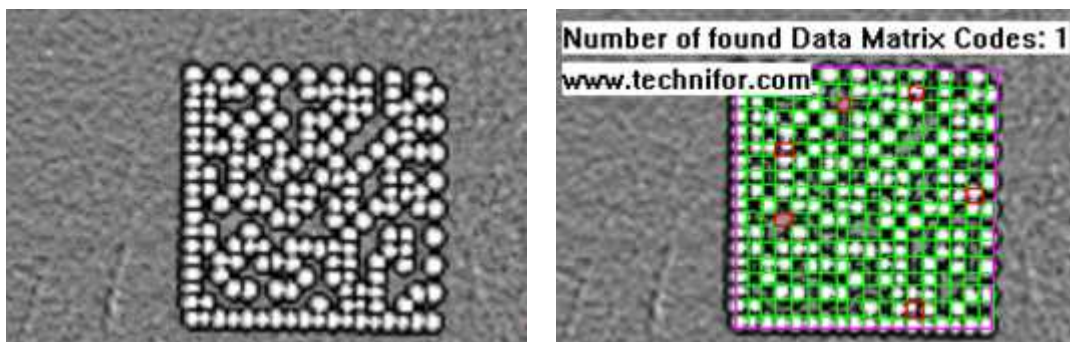
1. The last computations required for mean and gaussian curvatures are only performed when retrieving these maps to avoid computing them unnecessarily.
 - This is why these methods are named `ComputeXXX` instead of `GetXXX`.
 - The results are cached to avoid computing them several times.
2. Albedos, mean and gaussian curvatures are intrinsically floating point images. To convert them to `EImageBW8`, you can choose between several arguments:
 - `EPhotometricStereoContrast_HighContrast` to ignore outliers and produce images with a high contrast.
 - `EPhotometricStereoContrast_FullRange` to produce images where no data is ignored.
 - `EPhotometricStereoContrast_FixedRange` to produce images where the range is specified using an additional argument. This is especially useful when processing several different objects with a fixed threshold.

Processing the Results with Open eVision Tools

As the computation results are [EImageBW8](#), you can process them with the various tools of Open eVision.

Here are some examples:

- Reading an embossed code with EasyBarCode2, EasyMatrixCode and EasyQRCode.



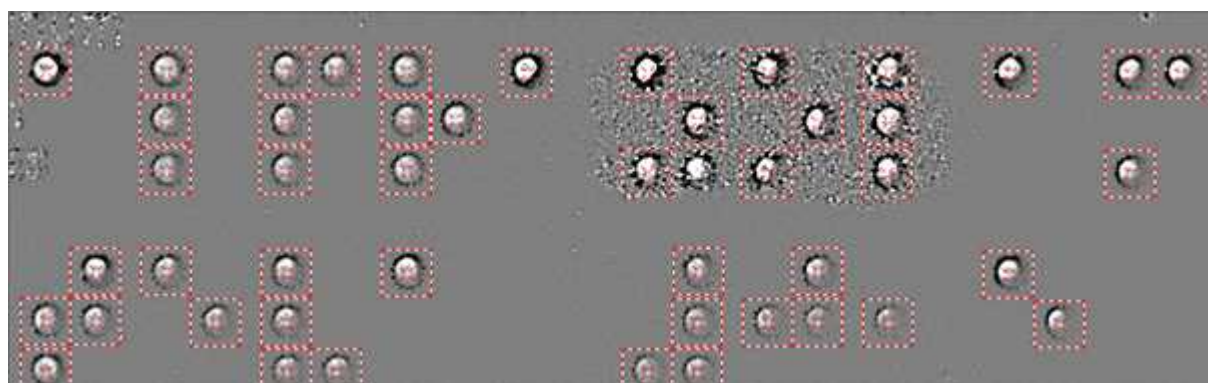
Mean curvatures of an engraved matrix code and EasyMatrixCode2 reading

- Reading an engraved text with EasyOCR2.



EasyOCR2 reading on the mean curvatures

- Finding patterns on embossed surfaces with EasyFind, EasyMatch and EasyObject.



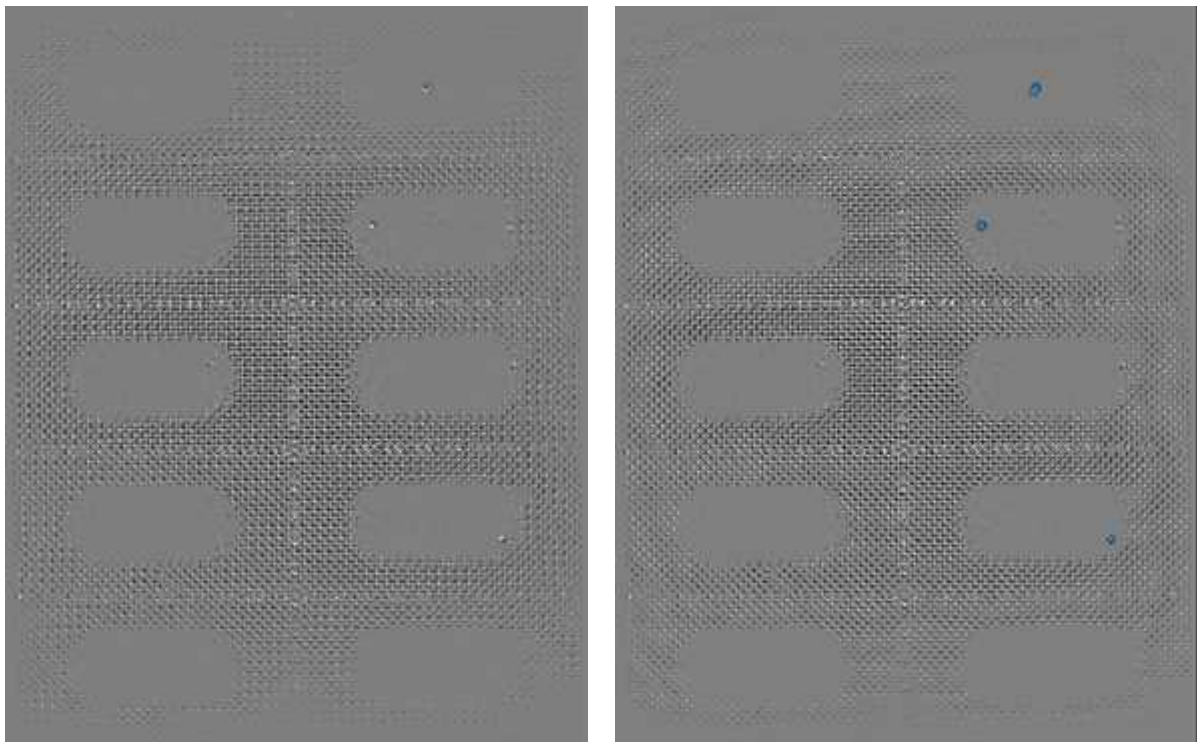
EasyFind results on the gaussian curvatures to detect braille characters

- Measuring shapes with EasyGauge.



Rectangle measurement with EasyGauge on the mean curvatures

- Finding defects in objects with EasySegment.



Gaussian curvatures of a blister pack with 3 holes and EasySegment supervised potential results

Improving the Results

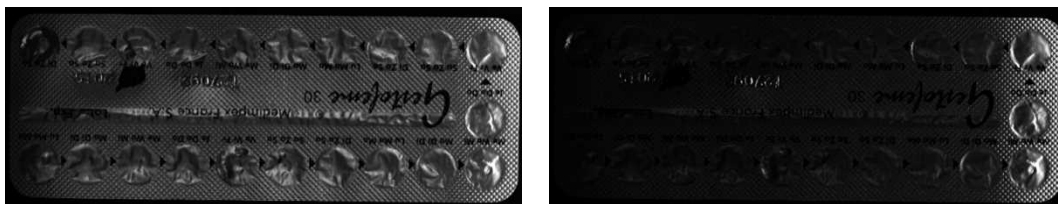
Using a dark image to account for ambient lighting

The photometric stereo assumes that each image is lit from a single light source.

- This assumption is not valid if the setup is exposed to (non-negligible) ambient lighting.
- To handle this issue, the `EPhotometricStereoImager` provides an `EImageBW8` dark image to the methods `Calibration` and `Compute`.
 - This dark image is an image of the object under ambient light only (all setup lights are off).



The dark image



The object image: raw (left) and after correction with the dark image (right)

Using flat images to correct non-uniform lighting

Photometric stereo assumes that each image is lit from an intensity uniform light source.

- This means that each pixel is lit by the same quantity of light.
- This assumption is not valid in physical setups using leds, where the part of the image closest to the leds receives more light.
- To handle this issue, the `EPhotometricStereoImager` provides a method to register a flat image used by the method `Compute`.
 - This flat image is an image of a uniform background taken in the same lighting configuration.

```
// calibrate imager or sets its angles (Todo)
std::vector<EImageBW8> flatImages;

// Load flat images in the same order than the calibration images/angles (Todo)
std::vector<EROIBW8> flatROIs;
```

```
// Set the flat images ROIs (Todo)

// Configure flat images, this could optionally be done with a dark image as well
photometricStereo.ConfigureNonUniformLightingCorrection(flatROIs);

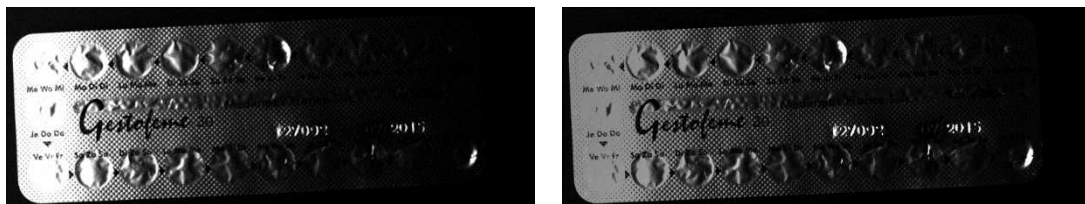
// Perform one or more computations, each will use the flat images (Todo)
photometricStereo.Compute(objectROIs);

// Optional: non uniform lighting correction could be disabled or (re-)enabled
// using SetEnableNonUniformLightingCorrection
```

- The following example illustrates the effect of a non-uniform lighting correction on the object images.
 - The proximity of the light source generates a lighting effect on the left of the image that is visible on both the flat and the raw images.
 - This effect is corrected on the last image, where the brightest pixels are those oriented towards the surface.



The flat image



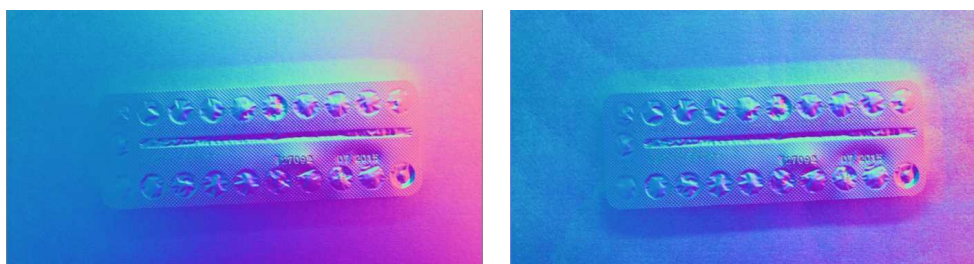
The object image: raw (left) and after correction with the flat image (right)

- The following examples illustrate the effects of a non-uniform lighting correction on 2 albedos images.
 - The corrected albedos show less burning on the extremities of the images.



The albedos images: raw (left) and after correction with the flat images (right)

- The following examples illustrate the effects of a non-uniform lighting correction on a normals images.
 - The normals fields is more uniform.



The normals image: raw (left) and after correction with the flat image (right)

Effect of the distance between lights and object

There is a tradeoff in the distance between the light and the object (that is the elevation angle).

- When the elevation angle is high, the lighting is more uniform. This means that:
 - The “burning” effects visible on some images is less important.
 - The shadows are also less of a problem.
- When the lighting source is close, the lighting directions are more diverse. This means that:
 - The quantity of information used to build the photometric stereo is higher.



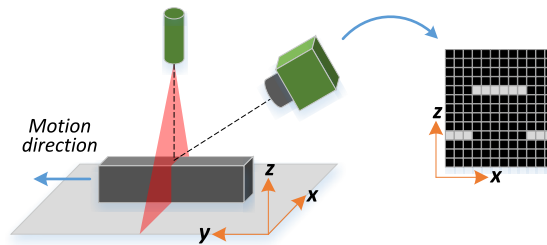
TIP

We recommend using elevation angles between 30 and 70°.
We achieve our best results around 40°.

2. Easy3DLaserLine - Laser Line Extraction and Calibration

2.1. 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.

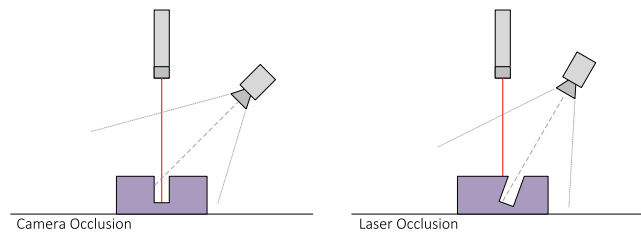
From the scanning you can reconstruct its 3D shape.



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.



TIP

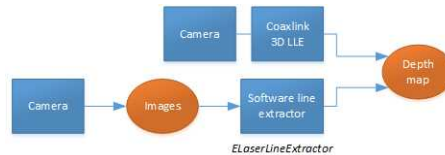
You can limit or avoid occlusions by using advanced scanning methods, for example by using two cameras or two lasers.

2.2. 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.

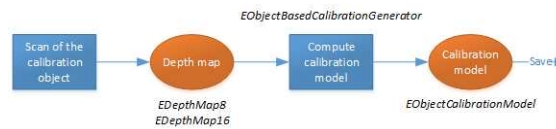
The source material for 3D processing is the depth map, coming from a Coaxlink Quad 3D-LLE or generated from a list of images.

3 types of depth map are available, one for each different pixel coding scheme (8, 16 or 32 bits).



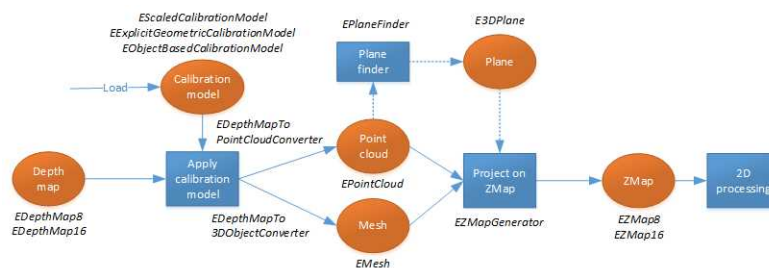
The generation of a depth map, from a hardware or a software source

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”.



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.

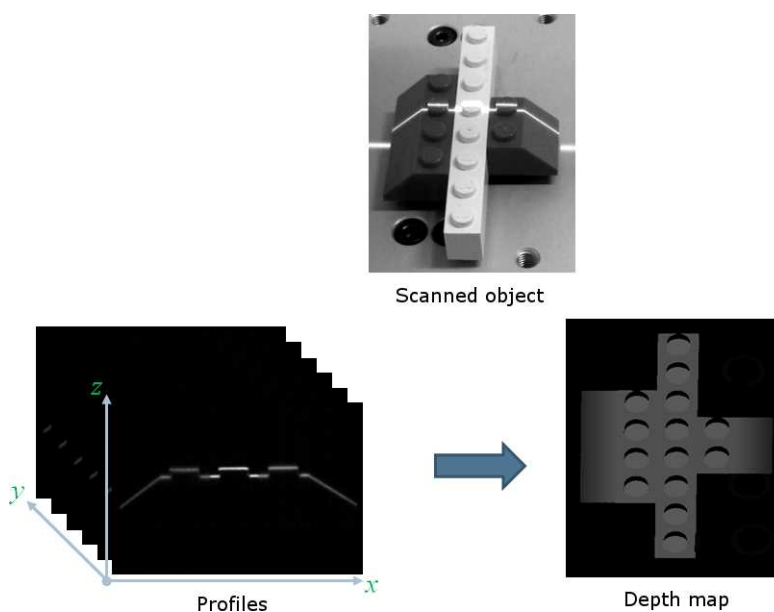
2.3. Laser Line Extraction

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.



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):

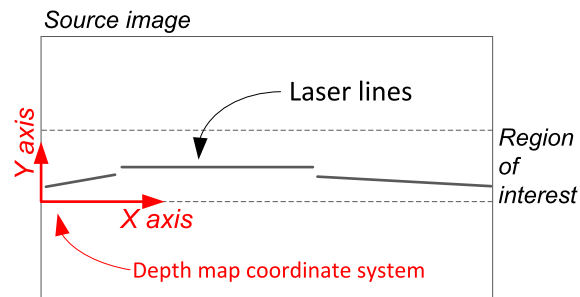
- **Maximum detection** returns the position of the pixel of maximum intensity. It's the fastest method but it doesn't support sub-pixel precision.
- **Peak detection** approach detects local maxima. If several maxima are detected, the one with the highest intensity is returned. The position is returned 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.

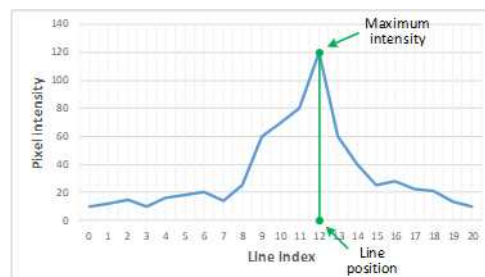
The line position returned by the laser line extraction algorithms is relative to the bottom of the region of interest. So, values in the depth map range from 0 (bottom of the ROI) to the height of the ROI.



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:

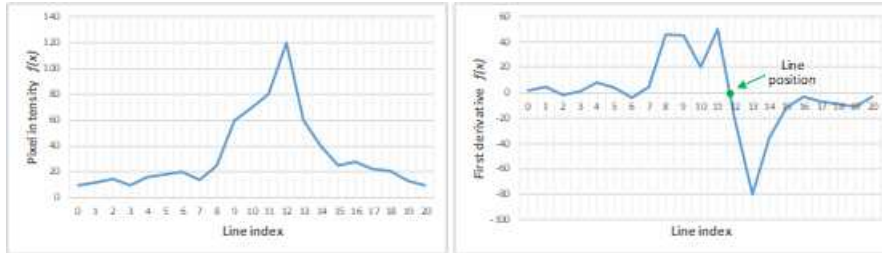
- A low-pass filter to reduce the high frequency variations in the image.
- A threshold to eliminate the background noise from the sensor.

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 $y = 0$:

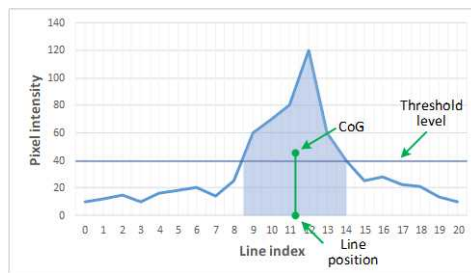
$$x = \frac{x_1 y_2 - x_2 y_1}{y_2 - y_1}$$

Center of gravity

The center of gravity (CoG) method uses an algorithm that calculates the center of mass of an image object. Also known as "centroid of plane figures", the CoG is obtained by the following equations:

$$\bar{X} = \frac{\sum ax}{\sum a} \quad \bar{Y} = \frac{\sum ay}{\sum a}$$

where \bar{X} and \bar{Y} are the coordinates of the CoG and a is the pixel intensity along the x and y axes.

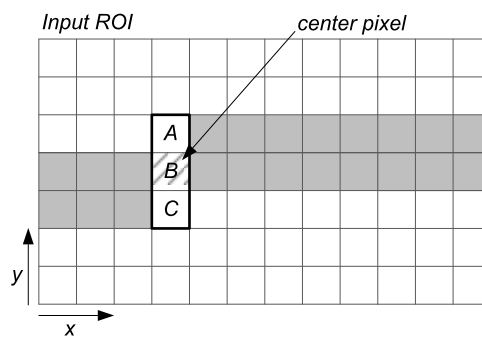


Center of gravity on a ROI profile

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)`.

2.4. 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 set up this conversion:

- Apply a simple scale on the pixel coordinates of the depth map ([EScaleCalibrationModel](#) class)
- Use the explicit geometric model ([EEExplicitGeometricCalibrationModel](#) 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 $x = \text{width} - 0.5$ and $y = \text{height} - 0.5$ 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

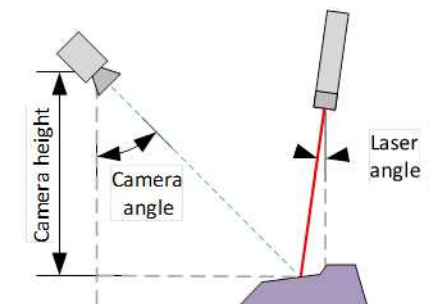
The explicit geometric model ([EExplicitGeometricCalibrationModel](#)) defines a simple and ideal laser triangulation setup. The explicit calibration makes some strong assumptions on the setup geometry and can only be used when a minimum set of parameters are known:

- 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).



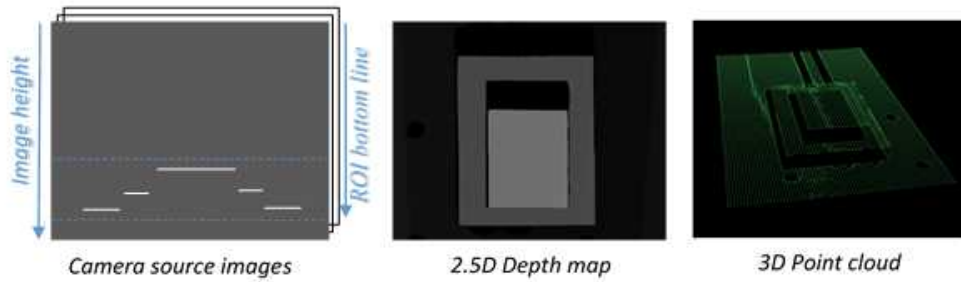
TIP

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.



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](#)" on page 299 section.

2.5. 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

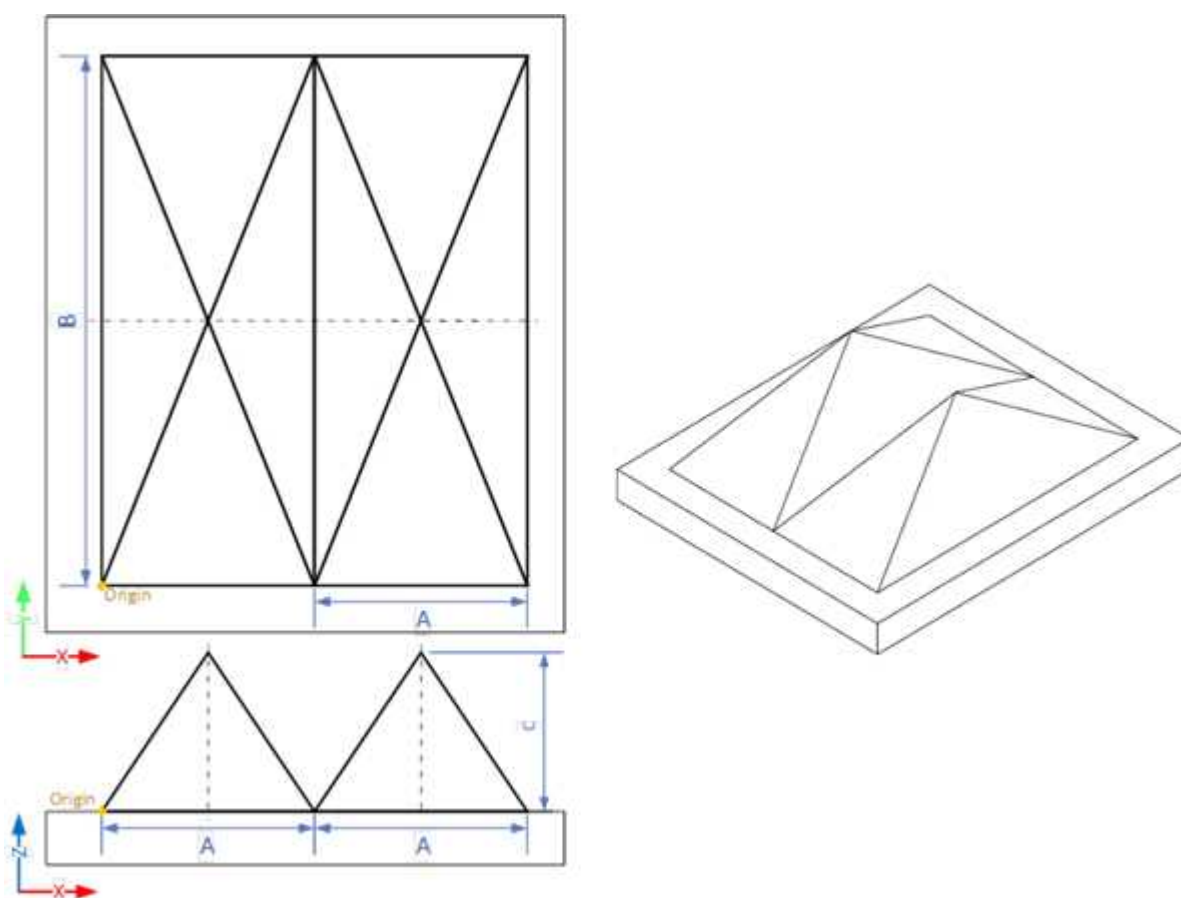
The general principle of Easy3D calibration is to match a scan of a known calibration object to its true geometric dimensions.

The double pyramid

**TIP**

In Open eVision 2.7 the “double truncated pyramid” calibration object is recommended over the “double pyramid” model.

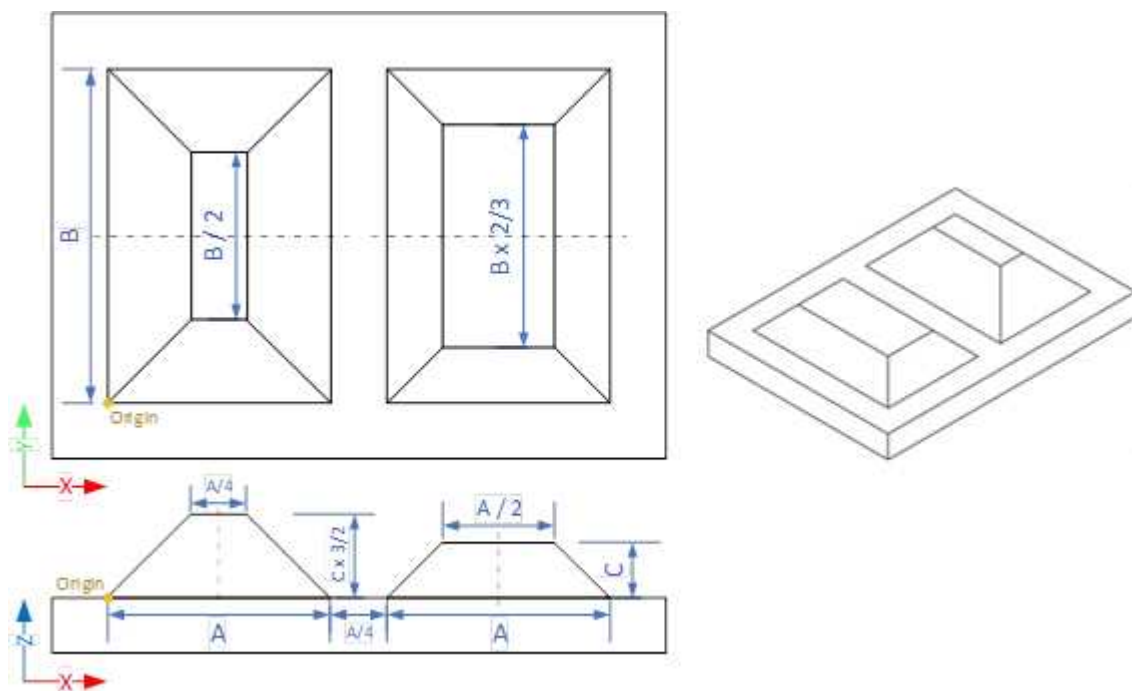
The dimensions of the “double pyramid” calibration object along the X-, Y- and Z-axes are named A, B and C respectively.



The "double pyramid" calibration model

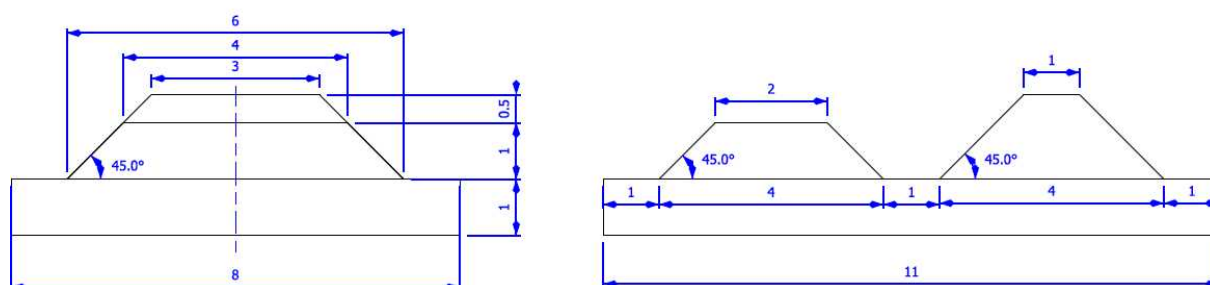
The truncated double pyramid

- 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

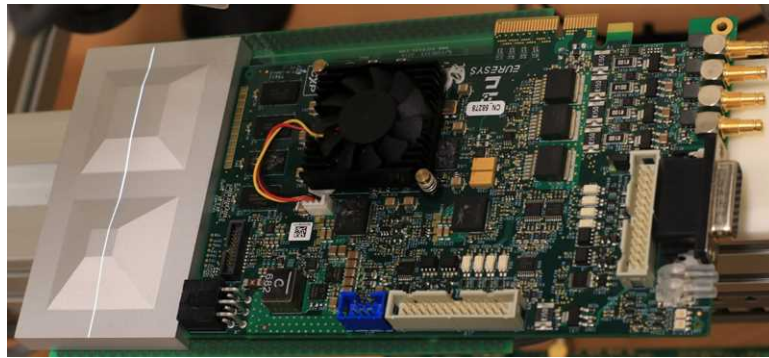
Overall dimensions

- 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.



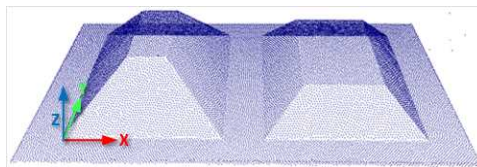
A PCB scanning setup with the associated calibration object
The calibration object dimensions (A, B and C) match the width and the height of the PCB



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.

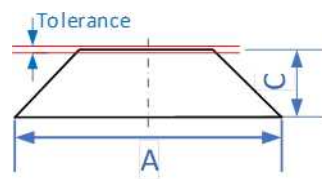


A calibrated point cloud with the origin and the axis of the coordinates system
The 3D origin is located at the external corner of the higher pyramid

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.

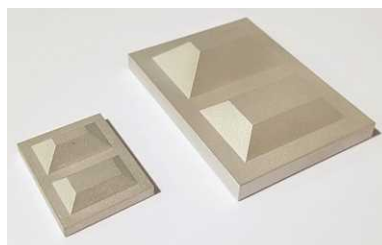
The build material and the surface finishing are also important and must have:

- 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.

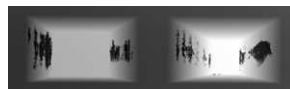
Download these files from the Open eVision download area in the **Additional Resources** section (www.euresys.com/Support).

OPEN EVISION			
	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)		Windows
	Open eVision Documentation EN 2.7.0.12133.exe	72 MB	Windows
	Open eVision Documentation EN CN 2.7.0.12133.exe	0.1 GB	Windows
	Open eVision Documentation EN JP 2.7.0.12133.exe	0.1 GB	Windows
Setup Files	Open eVision 2.7.0.12133 Installer.msi	0.2 GB	Windows
Additional Resources	Easy2D Calibration Models 2.7.0.12133.zip	0.8 MB	
	Easy2D Scanning Windows/Free Licenses v1.0.12133.zip	34 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

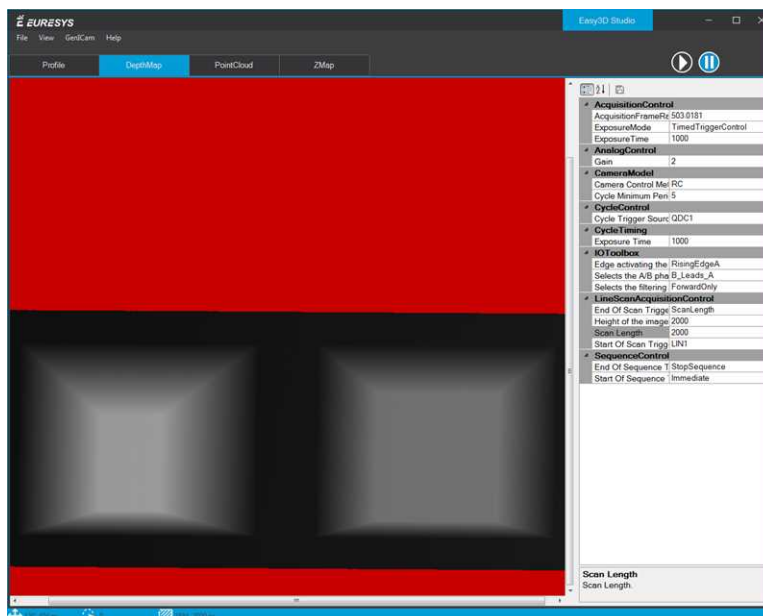
Calibration with Easy3D Studio

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:

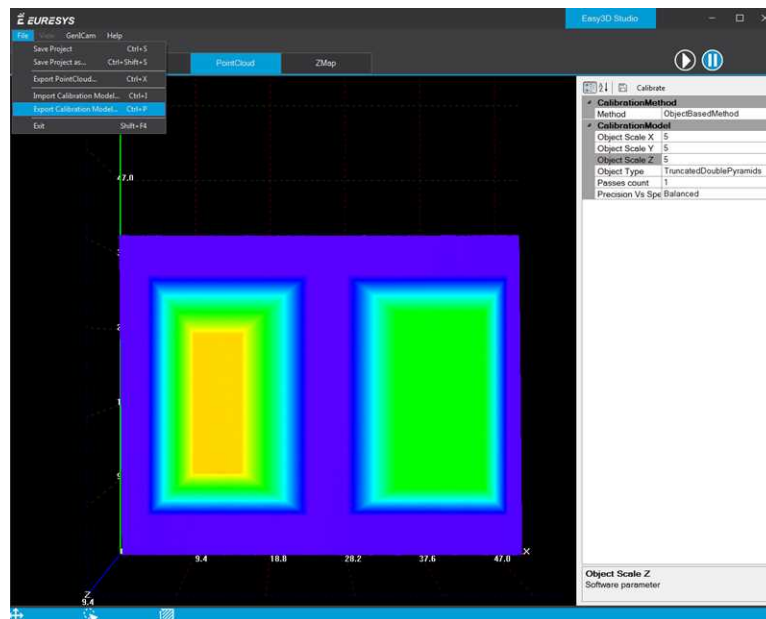
- The scanned image.
- The acquisition parameters on the right side.



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.



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
EDepthMap_6 depth_map;
depth_map.LoadImage("ctx-calibration object.png"); // from Easy3D sample images
depth_map.SetZResolution(1.f / (1 << 5)); // 1.5 fixed point pixel format
// Initialize the calibration generator
EObjectBasedCalibrationGenerator calib_generator;
calib_generator.SetCalibrationObjectType(EObjectBasedCalibrationType_TruncatedDoublePyramid, 40.f, 60.f, 10.f);
// Type and 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) EDepthMap_6 depth_map;
depth_map.LoadImage("ctx-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");

```

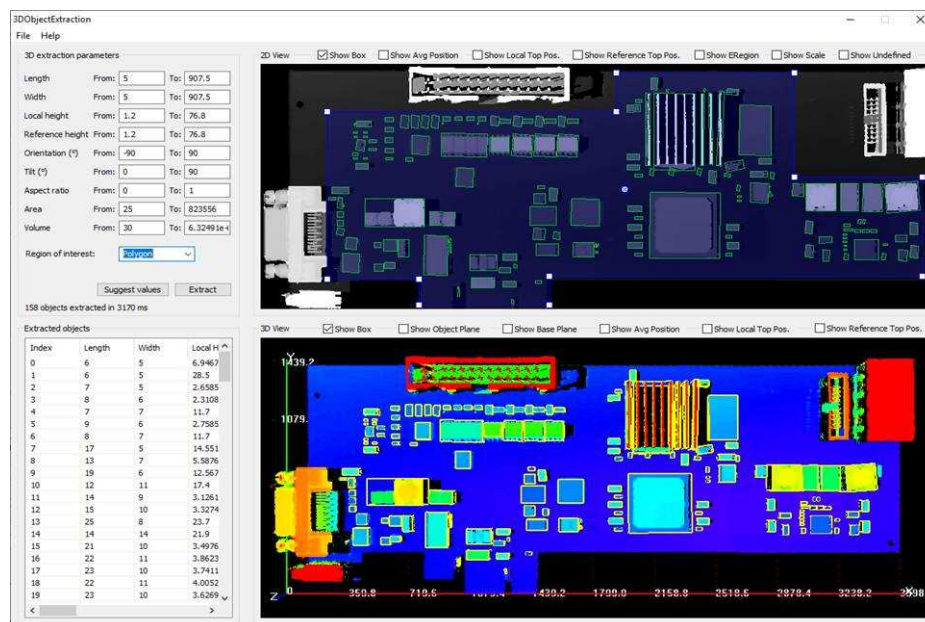
To experiment and learn about the Easy3D calibration, a C++ sample called `3DCalibration` is provided with the source code in the Open eVision distribution.

3. Easy3DObject - Extracting 3D Objects

3.1. 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.



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.

3.2. 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.

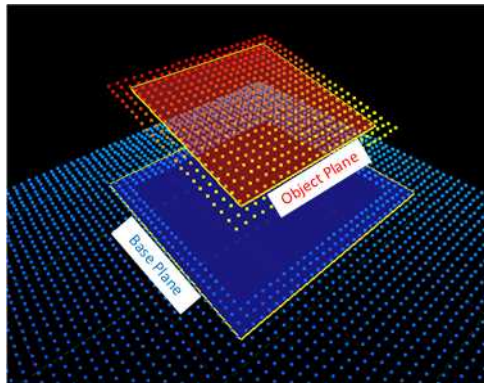
Object plane and base plane

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`

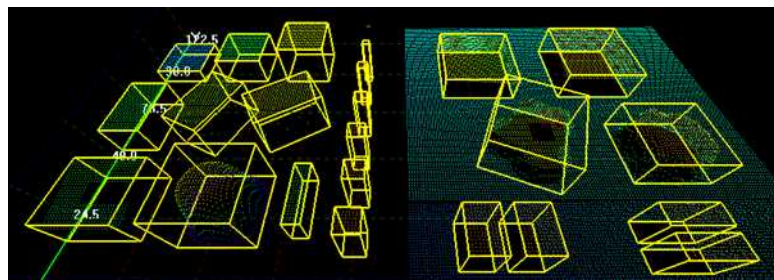
- This plane is called the *base plane*.
- It is an estimation of the local background around the object.
- If there are too many undefined pixels in this area, the base plane is equal to the reference plane of the input `EZMap`.



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.

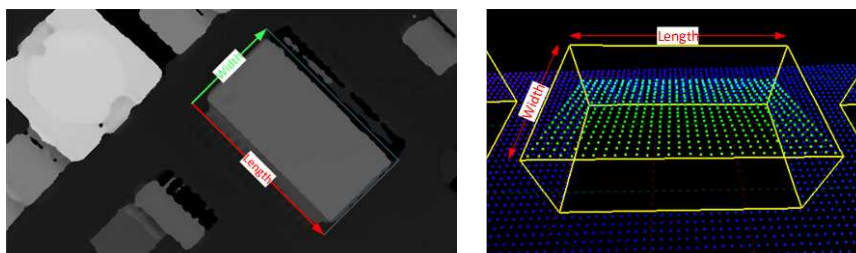


Length and width

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.



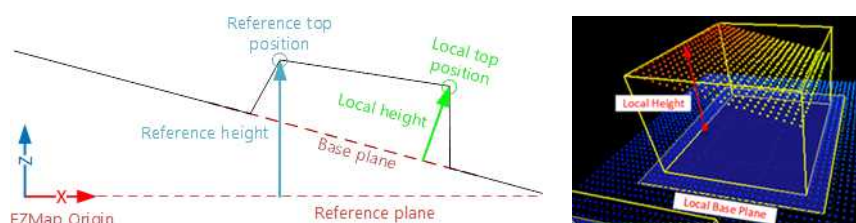
Local and reference top positions and heights

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.

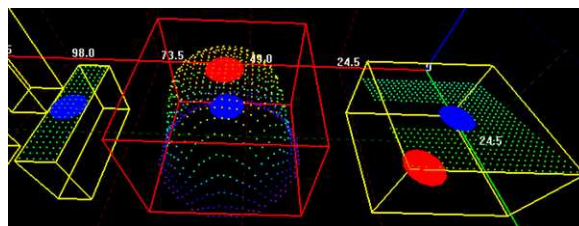
Use the `E3DObjectExtractor.LocalHeightRange` and the `E3DObjectExtractor.ReferenceHeightRange` accessors to set the ranges of allowed values for the local and the reference height.

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.

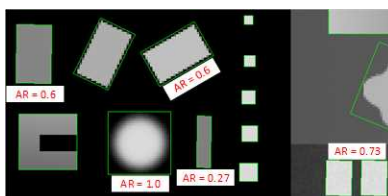


Aspect ratio

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.

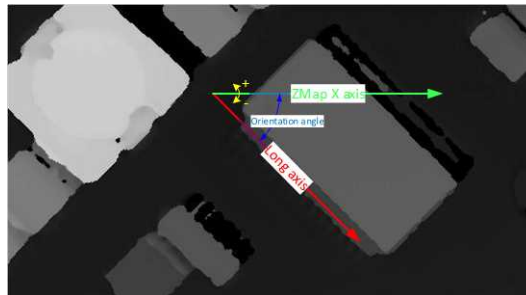


Orientation angle

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 $+90^\circ$.

Use the `E3DObjectExtractor.OrientationRange` accessor to set the range of allowed values for the orientation angle.



Local and reference tilt angles

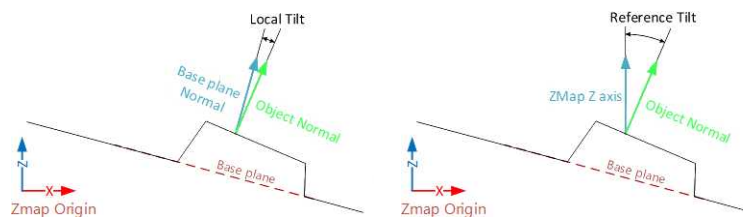
The *local tilt angle* is the angle between the base plane and the object plane.

- A value of 0 means that the object top surface is parallel to its base.
- The value must lie between 0° and $+90^\circ$.

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 $+90^\circ$.

Use the `E3DObjectExtractor.LocalTiltRange` and the `E3DObjectExtractor.ReferenceTiltRange` accessors to set the range of allowed values for the tilt angles.

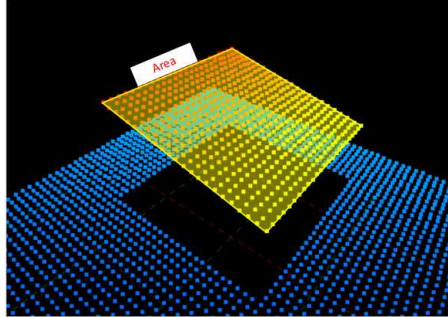


Area

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](#)].

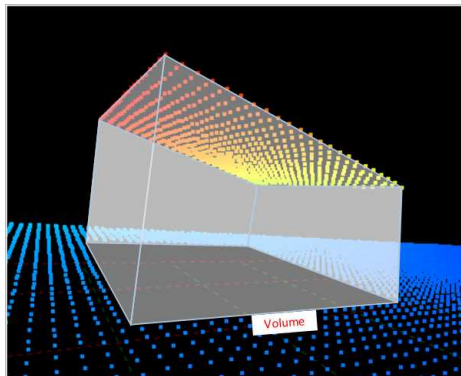
Use the [E3DObjectExtractor.AreaRange](#) accessor to set the range of allowed values for the area.



Volume

The object *volume* is the volume that lies between the top surface and the base plane of the object.

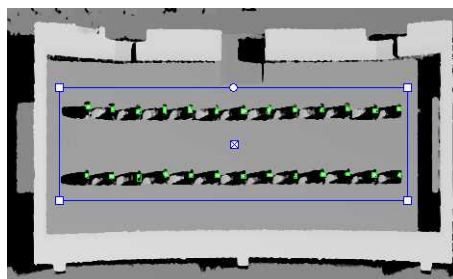
Use the [E3DObjectExtractor.VolumeRange](#) accessor to set the range of allowed values for the volume.



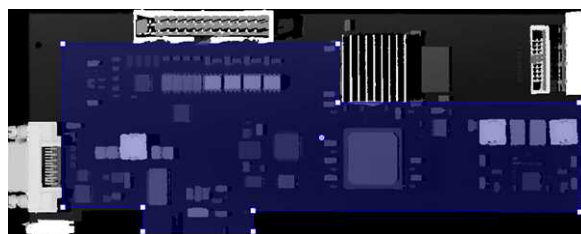
3.3. Extracting and Using Objects

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

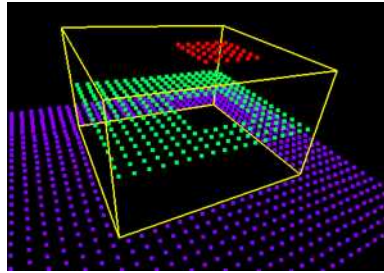
Adjust the extraction ranges to reduce the number of objects and speed up the extraction process.

Overlapped objects

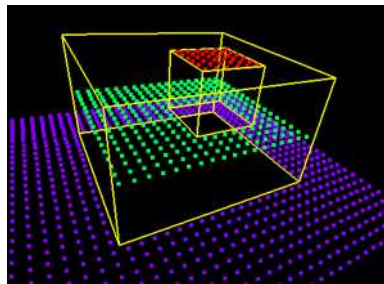
By default, the extraction does not produce objects that overlap on the ZMap. You must enable the `SetOverlappedObject` option to extract “stacked” objects.

The *area ratio* and the *height difference* parameters control how overlapped objects are extracted:

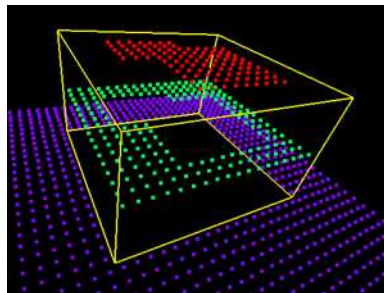
- The *area ratio* is configured by `SetOverlappedAreaRatio`. The area of the bottom object divided by the area of the top object must be larger or equal than that ratio.



Overlapped extraction is disabled.

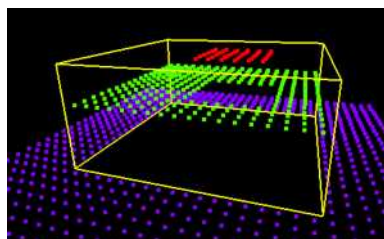


Overlapped extraction is enabled, with `OverlappedAreaRatio = 4`.

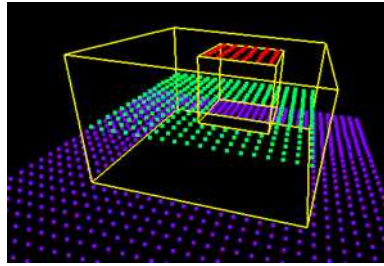


Overlapped extraction is enabled, with `OverlappedAreaRatio = 4`.
The top object is too large to be extracted, the ratio of the areas is lower than 4.

- The *height difference* is configured by `SetOverlappedHeightDifference`. This represent the minimum height difference between the top and the bottom object



Overlapped extraction is enabled, with `OverlappedHeightDifference = 2`.
The height of the top object (red) from the bottom object (green) is too small, the object is not extracted.

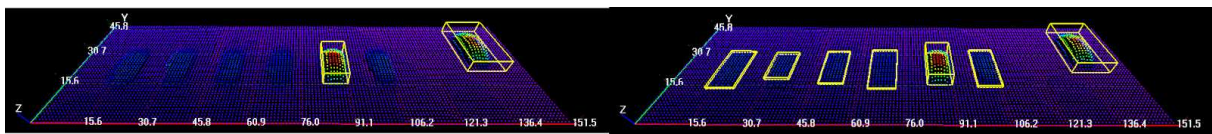


Overlapped extraction is enabled, with `OverlappedHeightDifference = 2`. The height of the top object from the bottom object is larger than 2, the object is extracted.

Controlling the object detection

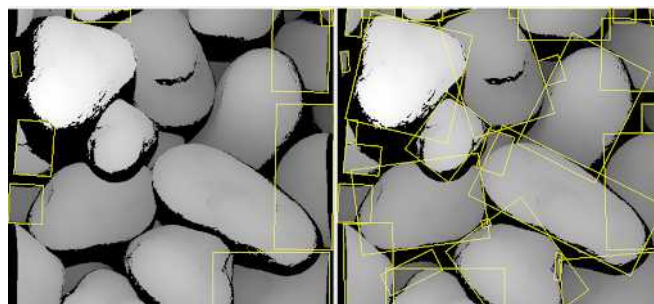
Two optional parameters affect the detection of the object:

- `SetExtractionSensitivity` controls the sensitivity of the extraction.
 - A higher value increases the ability to detect objects that are mixed with their surrounding, because their grey values are close to the background or because the transition (the gradient) between the object and the background is smooth.
 - This parameter value ranges from 0 to 1 (the default value is 0.6).



Extraction sensitivity: 0.5 (left) and 0.8 (right)

- `SetContourReinforce` affects the extraction of the objects.
 - As the extraction can fail when objects are close or touch each other, this parameter enables a filter to enhance the frontiers between objects and enable the extraction of such objects.
 - The filter can affect the measurements.



Contour reinforcement: OFF (left) and ON (right)

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.

```
// 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 objPlane = 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`.

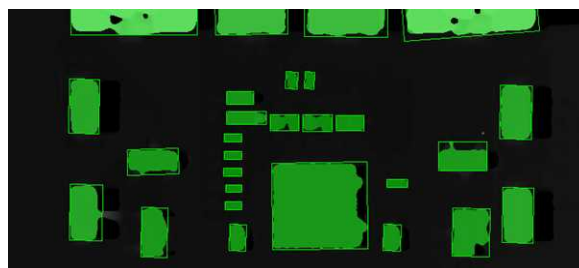


TIP

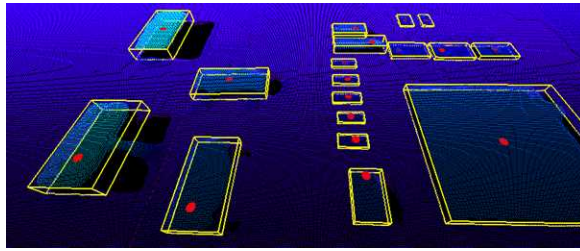
In an `E3DViewer`, use the `ERenderStyle` structure to choose your rendering style.

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](#)



3.4. Use Case - Inspecting a PCB

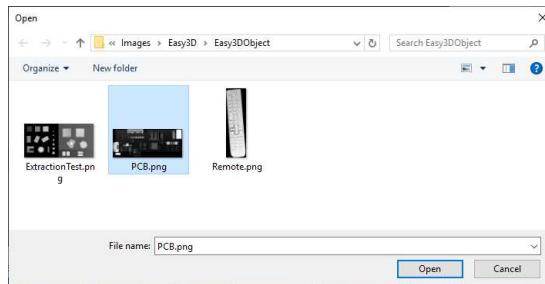
The purpose of this use case is to test if all the components are present and correctly placed on the PCB.



TIP

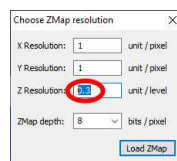
This example uses the sample image `Sample Images\Easy3D\Easy3DObject\PCB.png` and the illustrations are based on the Easy3DObject demo application.

1. Load the PCB image.



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.



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.

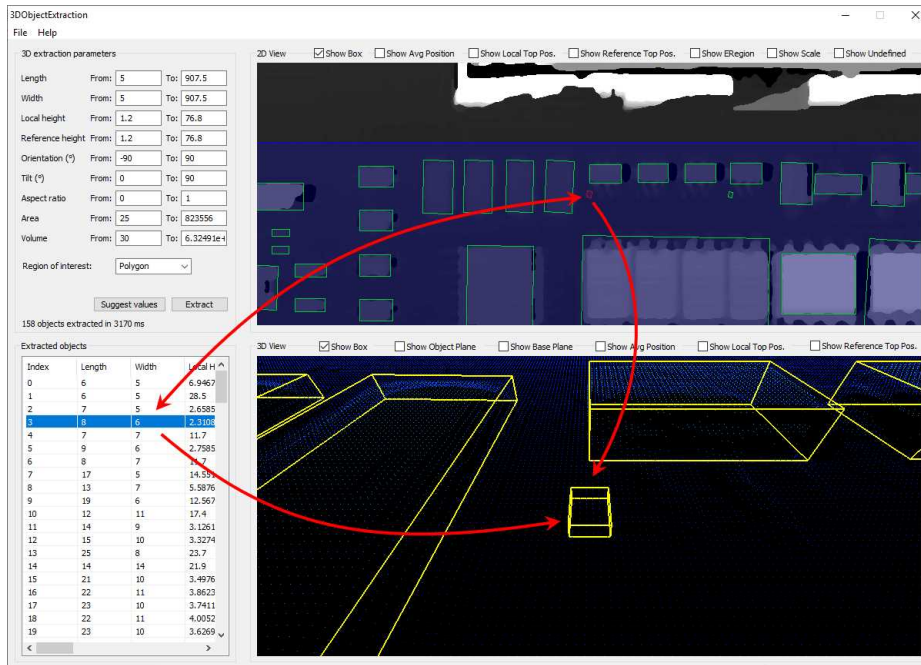


5. Use a polygon region of interest to restrict the searched area.

- 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.



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.



8. Use the size ranges to discard the smaller components.

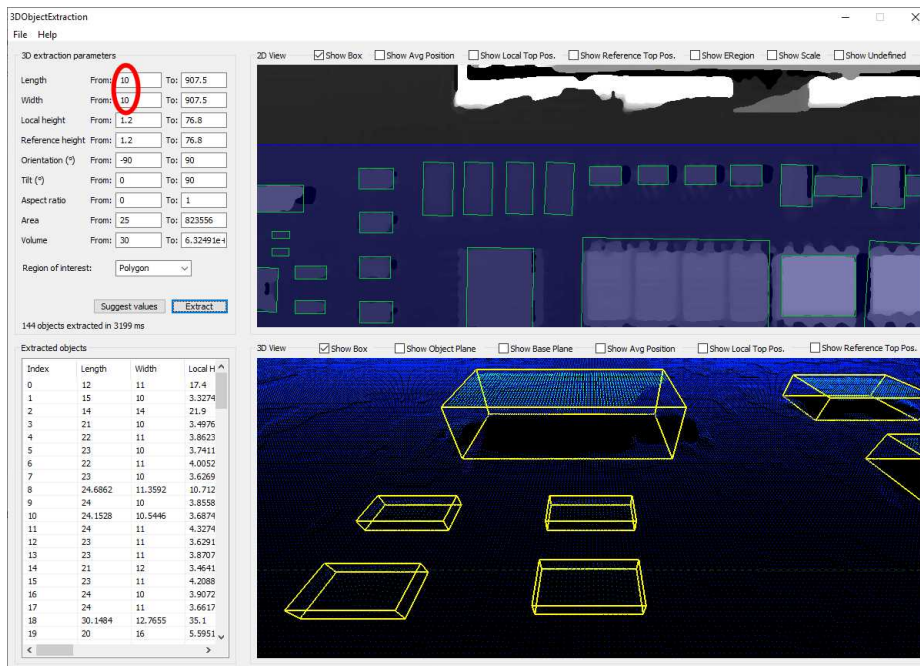
To add or remove objects:

- Change the extraction parameters, like the length and width ranges.
- In the illustration below, objects smaller than 10x10 metric unit are not extracted.

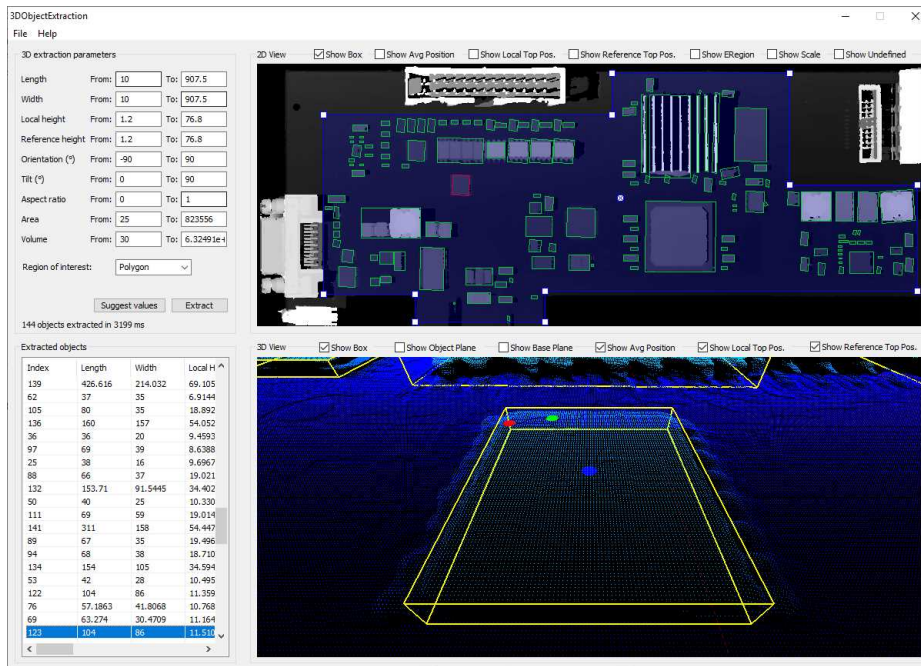


NOTE

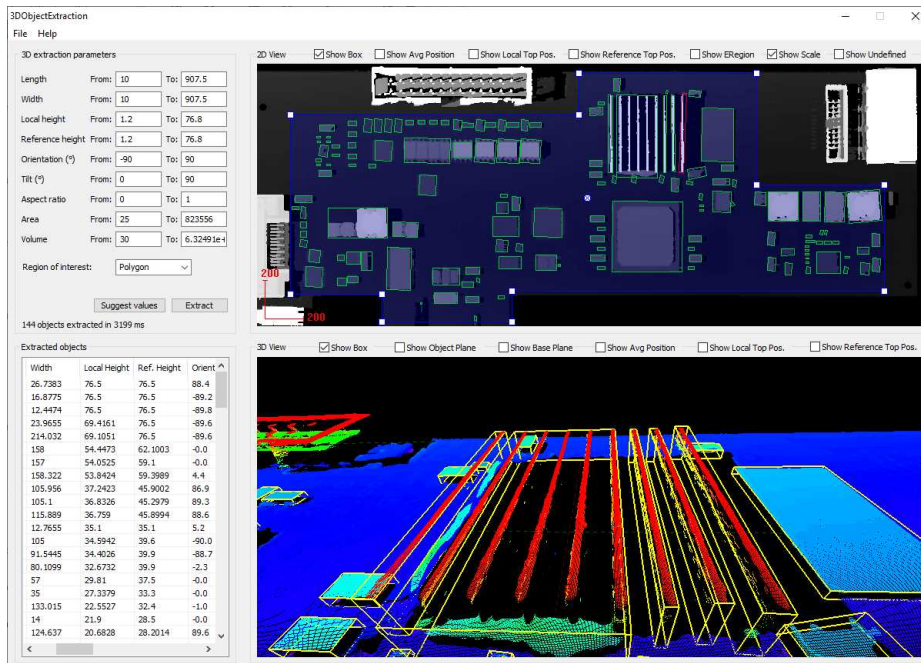
After changing a parameter, press the **Extract** button to perform a new extraction.



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.



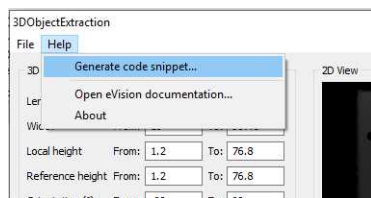
10. Adjust the extraction parameters to accept or reject objects based on the results.



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.



```
Code snippet
// 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(34, 254);
points[1] = EPoint(1844, 254);
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
```

4. Easy3DMatch - 3D Alignment and Comparison

4.1. Purpose and Workflow

Purpose

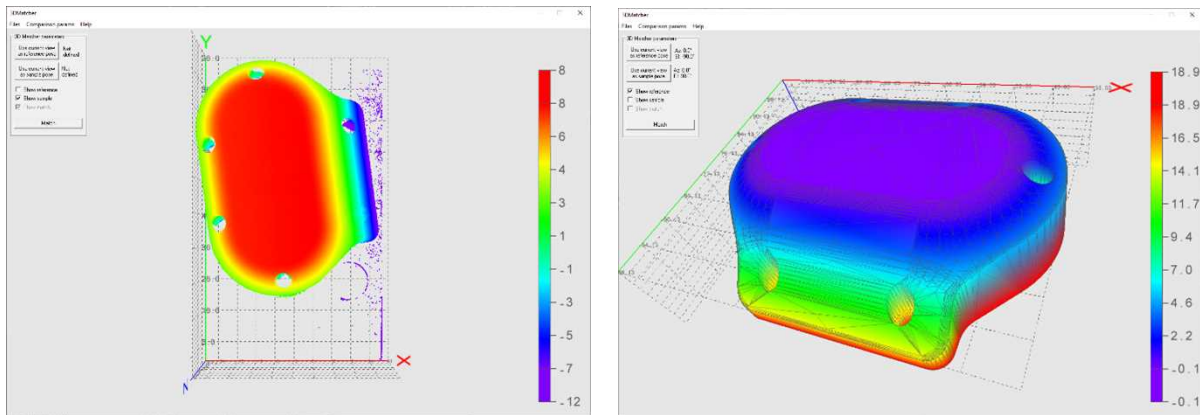
Easy3DMatch allows you to:

- Align a scanned object with another scan or with a reference mesh.
 - The Easy3DMatch tool features alignment functions to find the exact pose (position and orientation) of acquired 3D objects using a reference model.
 - You can specify this model as a reference point cloud or as a 3D mesh from CAD software.

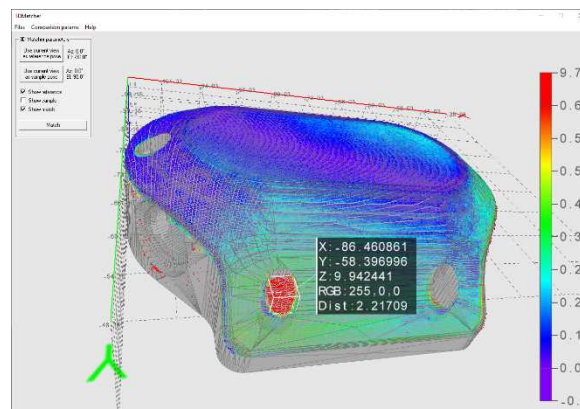


CAD model — sample point cloud — model and sample aligned
(3D models courtesy of Direct Dimensions)

- Compare an aligned scan with a reference model or mesh:
 - a. Compute the local distances between 3D scans and a golden sample or a reference mesh.
 - b. Detect anomalies such as misplaced features, geometric distortions, gaps and bumps.



Reference — sample
(3D CAD models courtesy of Direct Dimensions)



Result of the comparison

Workflow

1. Load a reference as:
 - A mesh (define its viewpoints)
 - A point cloud (with one viewpoint)
 - A ZMap
2. Load a sample, either as:
 - A point cloud (with one viewpoint)
 - A ZMap

3. Perform alignment and/or matching:
 - Optionally, align the sample and get its [E3DAlignment](#) with respect to the reference ([E3DAligner](#)).
 - Optionally, compare the sample to the reference by defining ROIs as [3DBox](#) ([E3DComparer](#)).
 - Align the sample and compare it directly to the reference by defining ROIs as an [ERegion](#) ([E3DMatcher](#)).
4. Use the transformation from sample to model to locate the sample and/or process the detected [E3DAnomalies](#).

Resources

- The example described here demonstrates how to use Easy3DMatch with Open eVision 3D tools.
- You can also find a sample application, with its source code, in `...\\Sample Programs\\MsVc samples\\3DMatcher`. Most of the illustrations are screenshots from this sample.
- The example and the sample application are based on the following resources:
 - Open eVision 2.16
 - Microsoft Visual Studio 2017
- You need the Easy3DMatch license to use it.

4.2. Alignment (E3DAligner)

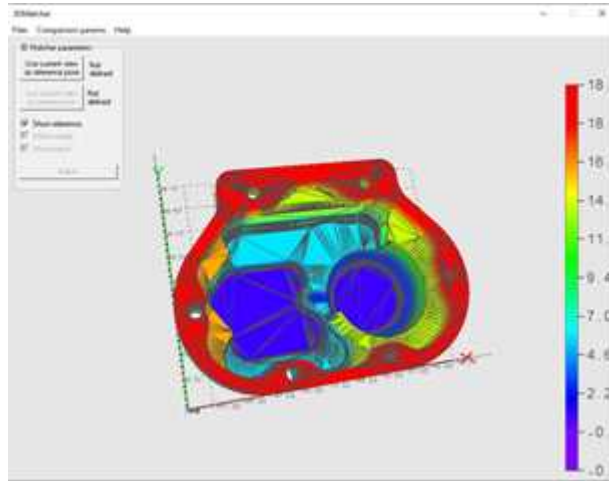
Base case

Use the class [E3DAligner](#) to load a reference and a sample and to find the transformation from the sample to the reference.

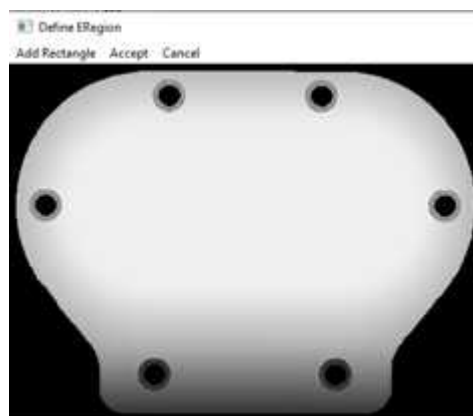
1. The first step is to set the reference using the method [SetReference](#).
 - In addition to the mesh or the cloud, this method also takes one or several [E3DPlane](#) or azimuth and elevation angles (see "[Calibration](#)" on page 280 for a definition of the azimuth and the elevation).
 - The angles are used to compute the plane and are just an easier way to specify it.

2. The goal of the plane is to specify the face(s) of the object that can be visible on the sample.
 - You must do this only once for the reference and once for the sample(s), assuming they are all taken with the same scanner.

Here is an illustration of the process:



View of the CAD



Bottom face, corresponding to the plane of the normal $(0, 0, -1)$ and equation $z = 15$ or an azimuth of 0° and an elevation of -90°



Top face of the object, corresponding to the plane of the normal $(0, 0, 1)$ and equation $z = -15$ or an azimuth of 0° and an elevation of 90° .

3. Call the method `Align` with a point cloud or a zmap and get an object `E3DAlignment` that contains:
 - The pose, an `E3DTransformMatrix` mapping the sample to the reference.
 - The error, indicating the quality of the matching.
 - The index of the reference pose that was matched. This is useful when several poses are defined (in the example above, there are 2 poses defined).

Defining a reprojection plane to improve the results

Ideally, the sample (and the point cloud or the ZMap used as reference) should be aligned on the viewpoint. This is however not always true, for example when the scanner does not lay on top of the object. In these cases, the user may specify the plane on which the object lays, either by giving an `E3DPlane` or a flat scan on which only the plane is visible.

In both cases, the plane normal must have the correct orientation (pointing upwards or downwards), that is if the plane is above the normal ($z_{\text{object}} > z_{\text{plane}}$), the z coordinate of the normal should be positive. This is specified either directly in the `E3DPlane` or by a boolean argument `objectAbovePlane` when giving a flat scan.

Code samples

Base sample

```

////////////////////////////////////
// This code snippet shows how to compute the //
// alignment between a sample and a cad reference.//
////////////////////////////////////

// load the reference mesh and define the pose
Easy3D::E3DAligner aligner;
Easy3D::EMesh cad;
cad.Load("...");
float azimuthReference = 0.f, elevationReference = 90.f;
aligner.SetReference(cad, azimuthReference, elevationReference);

// load the sample

```

```
Easy3D::EPointCloud sample;
sample.Load("...");
float azimuthSample = 0.f, elevationSample = 90.f;

// perform alignment
Easy3D::E3DAlignment alignment = aligner.Align(sample, azimuthSample, elevationSample);
```

Use a reprojection plane

```
////////////////////////////////////
// This code snippet shows how to set the //
// reprojection plane when performing alignment. //
////////////////////////////////////

// load the reference mesh and define the pose
Easy3D::E3DAligner aligner;
Easy3D::EMesh cad;
cad.Load("...");
Easy3D::E3DPlane refPlane(Easy3D::E3DPoint(0, 0, ↲), 0);
aligner.SetReference(cad, refPlane);

// define the reprojection plane
bool userKnowsPlaneAZEL = false; // depending on the user
if (userKnowsPlaneEquation)
{
    Easy3D::E3DPlane reprojectionPlane(Easy3D::E3DPoint(0, 0, ↲), ↲5);
    aligner.SetScanReprojectionPlane(reprojectionPlane);
}
else
{
    Easy3D::EPointCloud cloud;
    cloud.Load("...");
    bool objectAbovePlane = true; // is the object above the plane on the cloud
    aligner.SetFlatScan(cloud, objectAbovePlane);
}

// load the sample
Easy3D::EPointCloud sample;
sample.Load("...");
float azimuthSample = 0.f, elevationSample = 90.f;

// perform alignment
Easy3D::E3DAlignment alignment = aligner.Align(sample, azimuthSample, elevationSample);
```

Use E3DAlignment to align a sample on the reference

```
////////////////////////////////////
// This code snippet shows how to apply the //
// transformation of the E3DAlignment to the //
// sample to overlap it on the reference //
////////////////////////////////////

// perform alignment (see previous examples)
Easy3D::E3DAlignment alignment;
EPointCloud sample;

// align sample on reference
Easy3D::EPointCloud alignedSample;
Easy3D::EAffineTransformer::ApplyMatrix(alignment.GetPose(), sample, alignedSample);
```

Computation Time

The following table shows the computation time on a representative object. The first step of an alignment is to decimate the given `EPointCloud` (a very large cloud may explain important computation times).

Number of threads	Computation times
1	811 ms
2	618 ms
4	570 ms

4.3. Comparison (E3DComparer)

Base case

Use the class `E3DComparer` to load a reference and an already aligned sample (for example by using `E3DAligner`) and to find the distance map between both as well as anomalies.

- To use the `E3DComparer`:
 - Use the method `SetReference` to specify the reference with either an `EMesh` or an `EPointCloud`.
 - Set the options of the comparison (ROIs and thresholds).
 - Call the method `Compare` with the sample `EPointCloud`.
 - Use `ComputesAnomalies` to retrieve the list of `E3DAnomaly`.
 - Use `GetComparisonPointCloud` to retrieve the `EPointCloud` containing the distances.
- An `E3DAnomaly` represents a specific area in which the discrepancies between the sample and the reference are important. They are represented by:
 - An `EPointCloud` containing all the points of the anomaly and their distance to the sample.
 - The area of the anomaly.
 - Its center of gravity.
 - A bounding box around the anomaly.

Note that the `E3DAnomaly` represents points on the reference (except on `NoExtraMaterial` regions and when the distance mode is set to `EComparisonDistanceMode_Advanced` for `E3DMatcher`).

- Use `GetComparisonPointCloud` to retrieve an `EPointCloud` containing distance and/or colors that represent the distance to the sample or the reference for each of these points.

SetROI and SetDontCare

- The class `E3DComparer` can perform the comparison on only a subset of the object. This has two benefits: it is faster and it allows to ignore false-positives when detecting anomalies.
 - Use `SetROI` with a vector of `E3DBox` to define the zones on which to perform a comparison.
 - Use `SetDontCare` to specify areas that are excluded from the comparison.
 - A point belonging to `SetROI` and `SetDontCare` boxes is not compared. By default, all points are compared.

SetNoExtraMaterial

- The class [E3DComparer](#) checks, for each point of the reference (in the ROI), the distance to its nearest neighbor in the sample.
 - This avoids false positives, where we would use points of the sample that are not part of the object (the plane on which the object lays for example).
 - This allows to detect missing points on the sample.
 - A drawback of this approach is that extra material, that is points that should not be in the sample (for example, a hole that is filled) are not detected. You can solve this problem by specifying a list of [E3DBox](#) containing the areas that should not contain extra material.

SetAnomalyThresholds

- An anomaly is a sufficiently large contiguous area of points whose distance to the scan is above a threshold.
 - To specify the two thresholds (distance and area), use the method [SetAnomalyThresholds](#).
 - By default, these two thresholds are set relatively to the model size.
 - As a more advanced anomaly detection method, use [SetAnomalyHysteresis](#).

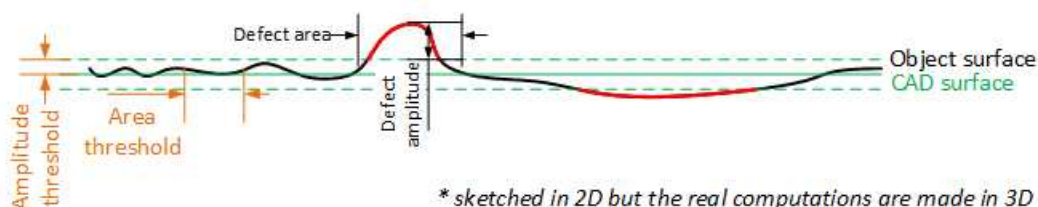


Illustration of the thresholds

SetAnomalyHysteresis

- You can use the method [SetAnomalyHysteresis](#) that is a more specific anomaly detection method.
 - With this method, a cluster of points should have a large enough subset of its points with an even larger distance to the sample to be an anomaly.
 - This may be useful if you do not want to consider as an anomaly the points with a medium distance unless they are close to points with a high distance.

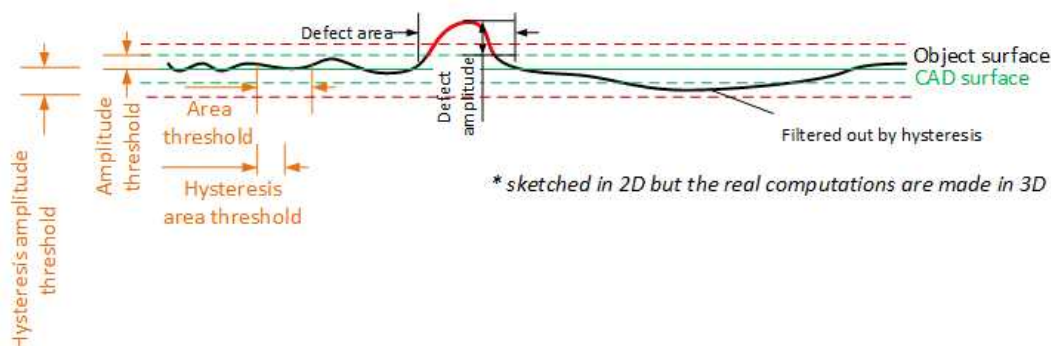


Illustration of the hysteresis thresholds

SetEnableAutomaticDecimation

- By default, the clouds are automatically decimated in such a way that the decimation error does not impact the anomalies detection.
 - This has the benefit of speeding up processing.
 - If the resolution of your point clouds is small with respect to your distance threshold, calling `SetEnableAutomaticDecimation(false)` could improve the speed as it will avoid useless decimation.

SetEnableAutomaticCrop

- By default, the sample clouds are automatically cropped around the reference to avoid computing distance for points that are not on the object (for example, the plane on which the object lays).

Prepare the reference

- If not called explicitly, the first call to `Compare` automatically computes the internal data structures.

Code samples

Minimal code

```

////////////////////////////////////
// This code snippet shows how to compare a sample //
// with a golden scan reference.                //
////////////////////////////////////

// load the reference golden scan and set reference
Easy3D::E3DComparer comparer;
Easy3D::EPointCloud cloud;
cloud.Load("...");
comparer.SetReference(cloud);

// set thresholds
float distanceThresh = .2f, areaThresh = 1.f;
comparer.SetAnomalyThresholds(distanceThresh, areaThresh);

// prepare data structures (optional)
comparer.PrepareReference();

// load the sample and perform comparison
Easy3D::EPointCloud sample;
sample.Load("...");
comparer.Compare(sample);

// compute anomalies
std::vector<Easy3D::E3DAnomaly> anomalies = comparer.ComputesAnomalies();

// TODO: if (anomalies.size() != 0u): an anomaly was detected: inspect the sample manually? throw it away?

// get cloud to inspect it manually
Easy3D::EPointCloud visualisationCloud;
comparer.GetComparisonPointCloud(visualisationCloud);

```

Advanced code

```

////////////////////////////////////
// This code snippet shows how to set the options //
// when comparing two elements with E3DComparer. //
////////////////////////////////////

// load the reference golden scan and set reference
Easy3D::E3DComparer comparer;
Easy3D::EPointCloud cloud;
cloud.Load("...");
comparer.SetReference(cloud);

// set thresholds
float distanceThresh = .2f, areaThresh = 1.f;
float hystDistanceThresh = 1.5f, hystAreaThresh = .5f;
comparer.SetAnomalyThresholds(distanceThresh, areaThresh);
comparer.SetAnomalyHysteresis(hystDistanceThresh, hystAreaThresh); // defined relatively to base thresholds

// set ROIs
std::vector<Easy3D::E3DBox> rois = { Easy3D::E3DBox(15, 15, 15) };
comparer.SetROI(rois);
std::vector<Easy3D::E3DBox> dontCare = { Easy3D::E3DBox(5, 5, 5) };
comparer.SetDontCare(dontCare);
std::vector<Easy3D::E3DBox> noExtraMaterial = { Easy3D::E3DBox(Easy3D::E3DPoint(10, 15, 20), 0, 0, 0, 5, 5, 5) };
comparer.SetNoExtraMaterial(noExtraMaterial);

// prepare data structures (optional)
comparer.PrepareReference();

// load the sample and perform comparison
Easy3D::EPointCloud sample;
sample.Load("...");
comparer.Compare(sample);

// compute anomalies
std::vector<Easy3D::E3DAnomaly> anomalies = comparer.ComputesAnomalies();

// TODO: if (anomalies.size() != 0u): an anomaly was detected: inspect the sample manually? throw it away?

// get cloud to inspect it manually
Easy3D::EPointCloud visualisationCloud;
comparer.GetComparisonPointCloud(visualisationCloud);

```

Computation time

The following table shows the computation time on a representative object. The first step of a comparison is to decimate the given `EPointCloud` (a very large cloud may explain important computation times).

If you are using a mesh as reference without a specific ROI, the mesh contains many points that have no correspondence in the scan (hidden faces), this can increase processing time a hundred-fold.

Number of threads	Computation times
1	686 ms
2	484 ms
4	395 ms

4.4. Alignment and Comparison (E3DMatcher)

Base Case

Use the class [E3DMatcher](#) to load a reference and a sample and to align and compare them at the same time.

[E3DMatcher](#) inherits from [E3DAligner](#) (see "[Alignment \(E3DAligner\)](#)" on page 327) and implements an API close to the one of [E3DComparer](#) (see "[Comparison \(E3DComparer\)](#)" on page 331) with some extra capabilities due to the usage of the reference points of view used in [E3DAligner](#).

To use the [E3DMatcher](#):

1. The first step is to set the reference using the method [SetReference](#).
 - In addition to the mesh or the cloud, this method also takes one or several [E3DPlane](#) or azimuth and elevation angles (see "[Calibration](#)" on page 280 for a definition of the azimuth and the elevation).
 - The angles are used to compute the plane and are just an easier way to specify it.
2. Set the options of the comparison (ROIs, thresholds, mode...) and optionally a reference plane.
3. Call the [Match](#) method with the sample point cloud and its reference plane.
 - This returns an [E3DMatch](#) object containing the anomalies and the [E3DAlignment](#) ([E3DMatch](#) inherits from [E3DAlignment](#))
4. Optionally, use [GetComparisonPointCloud](#) to retrieve an [EPointCloud](#) that contains the distances.
 - You can perform all these steps interactively in the [E3DMatch](#) sample.
 - The main difference between [E3DMatcher](#) and [E3DComparer](#) is that the ROIs are defined using [ERegion](#) on the [EZMap](#) corresponding to the projection of the reference on the given [E3DPlane](#).

This also allows for more advanced comparisons (see [ComparisonDistanceMode](#) and [SetEnableMissingPointAsAnomaly](#)).

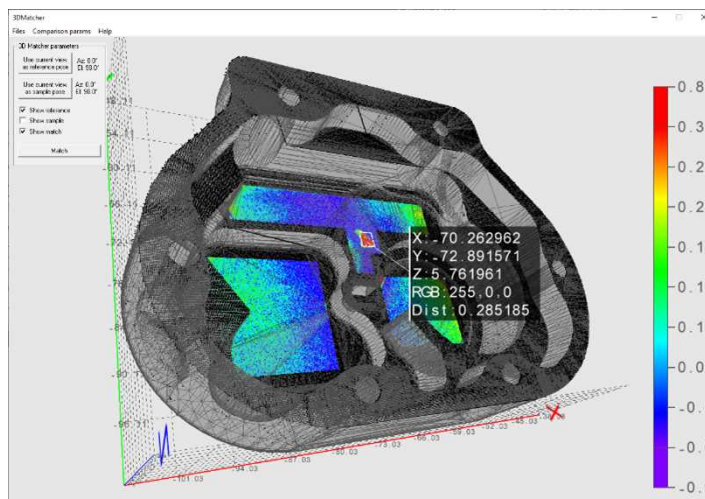
[SetComparisonDistanceMode](#)

Use [SetComparisonDistanceMode](#) to select one of the methods available in the class [E3DMatcher](#) to compute the distances:

- • [EcomparisonDistanceMode_Fast](#): the fastest method but less precise on the edges.
- • [EcomparisonDistanceMode_Normal](#): the default method.
- • [EcomparisonDistanceMode_Advanced](#): the slowest method that penalizes more bumps.

[SetAllComparisonROI](#)

- The class [E3DMatcher](#) can perform the comparison on only a subset of the object. This has two benefits: it is faster and it allows to ignore false-positives when detecting anomalies.
 - Use [SetAllComparisonROI](#) with one on several [Eregion](#) to define the zones on which to perform a comparison.
 - These [Eregion](#) should be interpreted as a masking part of the object on a projected view. Use [RetrieveReferencePoses](#) to retrieve the corresponding view.



Only the areas on the ROI are compared

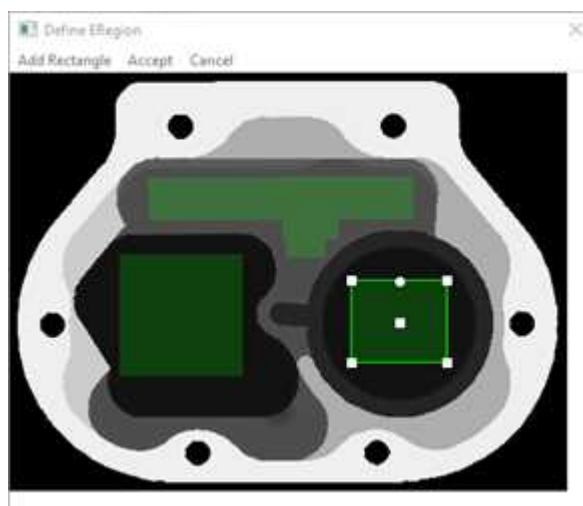
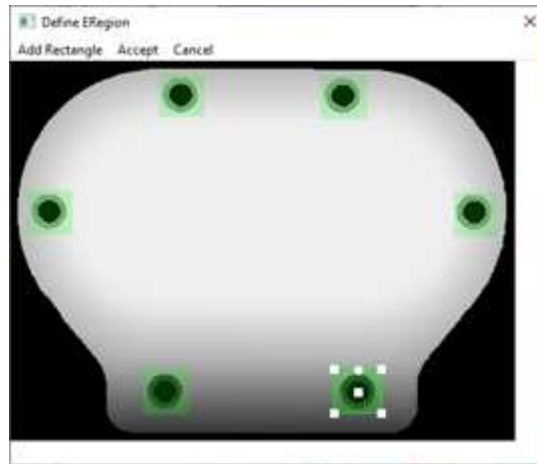


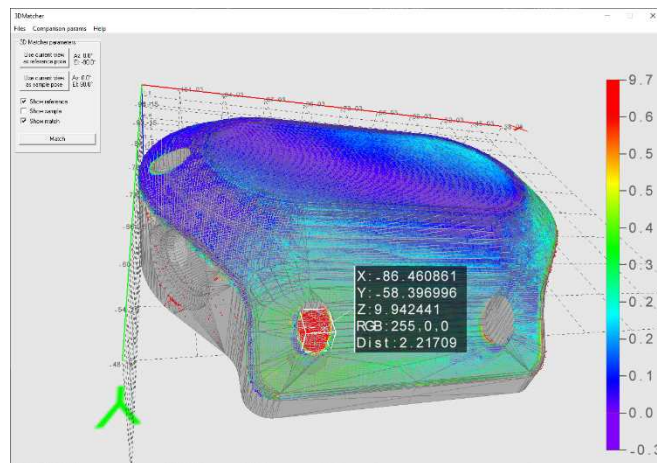
Illustration of setting ROI on a projection of the reference

SetAllComparisonNoExtraMaterial

- The class `E3DMatcher` checks, for each point of the reference (in the ROI), the distance to its nearest neighbor in the sample.
 - This avoids false positives, where we would use points of the sample that are not part of the object (the plane on which the object lays for example).
 - This allows to detect missing points on the sample.
 - A drawback of this approach is that extra material, that is points that should not be in the sample (for example, a hole that is filled) are not detected. You can solve this problem by specifying a list of `E3DBox` containing the areas that should not contain extra material.



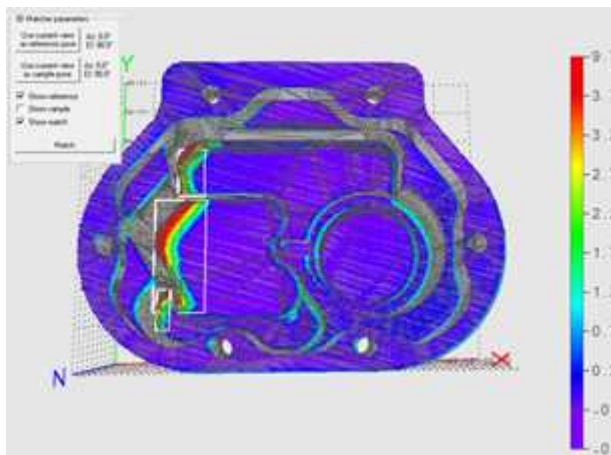
We do not want the holes to be filled



A filled hole is reported as an anomaly

SetEnableMissingPointAsAnomaly

- By default, the points missing on the scan are considered as defaults.
 - In some case this may lead to false positives (a shadow on the sample is not necessarily a default, just the absence of information).
 - In other cases these points should be taken into account (for example, a deep hole in the object can result in the absence of points instead of the presence of misplaced points).
 - Use the method [SetEnableMissingPointAsAnomaly](#) to select one of these behaviors.



False anomalies due to shadows in the sample

SetAnomalyThresholds

- An anomaly is a sufficiently large contiguous area of points whose distance to the scan is above a threshold.
 - To specify the two thresholds (distance and area), use the method [SetAnomalyThresholds](#).
 - By default, these two thresholds are set relatively to the model size.
 - As a more advanced anomaly detection method, use [SetAnomalyHysteresis](#).

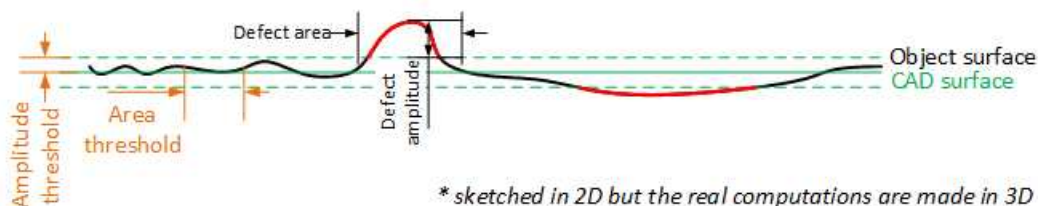


Illustration of the thresholds

SetAnomalyHysteresis

- You can use the method [SetAnomalyHysteresis](#) that is a more specific anomaly detection method.
 - With this method, a cluster of points should have a large enough subset of its points with an even larger distance to the sample to be an anomaly.
 - This may be useful if you do not want to consider as an anomaly the points with a medium distance unless they are close to points with a high distance.

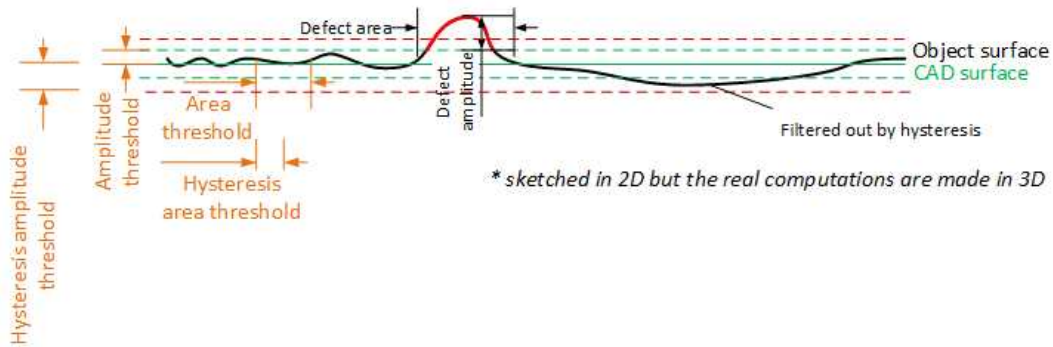


Illustration of the hysteresis thresholds

SetEnableAutomaticDecimation

- By default, the clouds are automatically decimated in such a way that the decimation error does not impact the anomalies detection.
 - This has the benefit of speeding up processing.
 - If the resolution of your point clouds is small with respect to your distance threshold, calling [SetEnableAutomaticDecimation\(false\)](#) could improve the speed as it will avoid useless decimation.

SetEnableAutomaticCrop

- By default, the sample clouds are automatically cropped around the reference to avoid computing distance for points that are not on the object (for example, the plane on which the object lays).

Prepare the reference

- If not called explicitly, the first call to [Compare](#) automatically computes the internal data structures.

Code samples

Minimal sample

```

////////////////////////////////////
// This code snippet shows how to match a sample //
// with a golden scan reference.                //
////////////////////////////////////

// load the reference golden scan and set reference
Easy3D::E3DMatcher matcher;
    
```

```

Easy3D::EPointCloud reference;
float azimuthReference = 0.f, elevationReference = 90.f;
reference.Load("...");
matcher.SetReference(reference, azimuthReference, elevationReference);

// set thresholds
float distanceThresh = .2f, areaThresh = 1.f;
matcher.SetAnomalyThresholds(distanceThresh, areaThresh);

// prepare data structures (optional)
matcher.PrepareReference();

// load the sample and perform comparison
Easy3D::EPointCloud sample;
float azimuthSample = 0.f, elevationSample = -90.f;
sample.Load("...");
Easy3D::E3DMatch match = matcher.Match(sample, azimuthSample, elevationSample);
std::vector<Easy3D::E3DAnomaly> anomalies = match.GetAnomalies();

// TODO: if (anomalies.size() != 0u): an anomaly was detected: inspect the sample manually? throw it away?

// get cloud to inspect it manually
Easy3D::EPointCloud visualisationCloud;
matcher.GetComparisonPointCloud(visualisationCloud);

```

Advanced sample

```

////////////////////////////////////
// This code snippet shows how to set the options //
// when matching two elements with E3DMatcher. //
////////////////////////////////////

// load the reference golden scan and set reference
Easy3D::E3DMatcher matcher;
Easy3D::EPointCloud reference;
float azimuthReference = 0.f, elevationReference = 90.f;
reference.Load("...");
matcher.SetReference(reference, azimuthReference, elevationReference);

// use advanced comparison mode
matcher.SetComparisonDistanceMode(EComparisonDistanceMode_Advanced);

// ignore shadows
matcher.SetEnableMissingPointAsAnomaly(false);

// set thresholds
float distanceThresh = .2f, areaThresh = 1.f;
float hystDistanceThresh = 1.5f, hystAreaThresh = .5f;
matcher.SetAnomalyThresholds(distanceThresh, areaThresh);
matcher.SetAnomalyHysteresis(hystDistanceThresh, hystAreaThresh); // defined relatively to base thresholds

// retrieve reference poses (reference must have been set)
std::vector<Easy3D::EZMap8> referencePoseProjections;
matcher.RetrieveReferencePosesProjections(referencePoseProjections);

// set ROI on the left half of the object
ERectangleRegion roiRegion(0.f, 0.f, float(referencePoseProjections[0].GetWidth()) / 2.f, float
(referencePoseProjections[0].GetHeight()));
matcher.SetComparisonROI(&roiRegion);

// set No Extra material on the whole object
ERectangleRegion noExtraMatRegion(0.f, 0.f, float(referencePoseProjections[0].GetWidth()) / 2.f, float
(referencePoseProjections[0].GetHeight()));
matcher.SetComparisonNoExtraMaterial(&noExtraMatRegion);

```

```
// prepare data structures (optional)
matcher.PrepareReference();

// load the sample and perform comparison
Easy3D::EPointCloud sample;
float azimuthSample = 0.f, elevationSample = -90.f;
sample.Load("...");
Easy3D::E3DMatch match = matcher.Match(sample, azimuthSample, elevationSample);
std::vector<Easy3D::E3DAnomaly> anomalies = match.GetAnomalies();

// TODO: if (anomalies.size() != 0u):an anomaly was detected: inspect the sample manually? throw it away?

// get cloud to inspect it manually
Easy3D::EPointCloud visualisationCloud;
matcher.GetComparisonPointCloud(visualisationCloud);
```

Computation time

The following table shows the computation time on a representative object. The first step of an alignment is to decimate the given `EPointCloud` (a very large cloud may explain important computation times).

Number of threads	Computation times
1	1335 ms
2	1180 ms
4	1165 ms

PART VII
ADVANCED PROGRAMMING

1. Multicore Processing

Multicore processing support in Open eVision

Since release 2.7, Open eVision supports 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)
EasyMatrixCode2	<code>EMatrixCodeReader::Read</code>	50%
EasyImage	Threshold on ERegion	75%
	Statistics on ERegion	75%
Easy3D	<code>EPointCloudToZMapConverter::Convert</code>	50%
	<code>EDepthMapToPointCloudConverter::Convert</code>	30%
	<code>EDepthMapToMeshConverter::Convert</code>	30%
	<code>EZMapToPointCloudConverter::Convert</code>	30%
	<code>ELaserLineExtractor::ExtractProfileFromFrame</code>	30%
EasyClassify	<code>EClassifier::Classify</code>	2%
EasySegment	<code>EUnsupervisedSegmenter::Apply</code>	4%

Library	Method	Improvement (per additional processor)
Regions	<code>ERegion::ToImage</code>	
	<code>ERegion::Union</code>	
	<code>ERegion::Intersection</code>	
	<code>ERegion::Subtraction</code>	
EasyQRCode	<code>EasyQRCode::Read</code> (see documentation for optimization)	20%

Thread-safe classes

- Open eVision supports simultaneous execution by multiple (unlimited) threads on the same CPU, but data can only be accessed by one thread at a time. So independent tasks can execute simultaneously in your application, but each bit of shared data must be controlled by a separate task.
- The following rules avoid data corruption, crashes and misbehaving programs.

Thread-safe basic types classes

Basic types	Recommendations	Restrictions
Basic pixel structures <code>EColor</code> , <code>EPeak</code> , <code>EISH</code> , <code>ELAB</code> , <code>ELCH</code> , <code>ELSH</code> , <code>ELUV</code> , <code>EBW1</code> , <code>EBW8</code> , <code>EBW8Path</code> , <code>EBW16</code> , <code>EBW16Path</code> , <code>EBW32</code> , <code>EC15</code> , <code>EC16</code> , <code>EC24</code> , <code>EC24A</code> , <code>EC24Path</code> , <code>EPath</code> , <code>ERGB</code> , <code>ERGBColor</code> , <code>EVSH</code> , <code>EXYZ</code> , <code>EYIQ</code> , <code>EYSH</code> , <code>EYUV</code> , <code>EDepth8</code> , <code>EDepth16</code> and <code>EDepth32f</code>		No
Pixel collection classes <code>EColorLookup</code> , <code>EPseudoColorLookup</code> , <code>EPeakVector</code> , <code>EBW8Vector</code> , <code>EBWHistogramVector</code> , <code>EBW8PathVector</code> , <code>EBW16PathVector</code> , <code>EBW16Vector</code> , <code>EBW32Vector</code> , <code>EC24Vector</code> , <code>EPathVector</code> , <code>EColorVector</code> , <code>EColorLookup</code> and <code>EC24PathVector</code>	No restrictions on read-only access.	A single instance may not be modified by several threads. If a thread is modifying an instance, no other thread can access it.
Image classes <code>EImageBW1</code> , <code>EImageBW8</code> , <code>EImageBW16</code> , <code>EImageBW32</code> , <code>EImageC15</code> , <code>EImageC16</code> , <code>EImageC24</code> and <code>EImageC24A</code>	No restrictions on read-only access.	A single instance may not be modified by several threads. If a thread is modifying an instance, no other thread can access it.
3D map classes <code>EDepthMap8</code> , <code>EDepthMap16</code> and <code>EDepthMap32f</code> <code>EZMap8</code> , <code>EZMap16</code> and <code>EZMap32f</code>	No restrictions on read-only access.	A single instance may not be modified by several threads.

Basic types	Recommendations	Restrictions
		If a thread is modifying an instance, no other thread can access it.
Point cloud classes EPointCloud	No restrictions on read-only access.	A single instance may not be modified by several threads. If a thread is modifying an instance, no other thread can access it.
ROI classes EROIBW1 , EROIBW8 , EROIBW16 , EROIBW32 , EROIC15 , EROIC16 , EROIC24 and EROIC24A ERegion , ERectangleRegion , ECircleRegion , EEllipseRegion and EPolygonRegion	No restrictions on read-only access.	A single instance may not be modified by several threads. If a thread is modifying an instance, no other thread can access it. Different ROI can be added or removed from an image or moved event if their parent image is the same. Consequently, different threads can work on different areas of an image possibly changing in position and size during the process.

Thread-safe library classes

Library	Recommendations	Restrictions
EasyImage and EasyColor	Static methods from this class (provided threading rules applying to their arguments are not broken).	No
EasyObject		No
EasyMatch , EasyFind , EasyQRCode and EasyOCR2 EMatcher , EMatchPosition , EPatternFinder and EFoundPattern		A single instance cannot be accessed from several threads. Search field (read-only) can be shared by different objects.
EasyGauge and Shape subclasses Gauging classes (EPointGauge , ELineGauge , ERectangleGauge , ECircleGauge , EWedgeGauge), EWorldShape and EFrameShape)		Can be attached, moved or removed from different threads, even in the same hierarchy. A single instance must not be used by different threads.

Library	Recommendations	Restrictions
Basic geometric classes (EFrame, EPoint, ECircle, ELine, ERectangle and EWedge)		Can be accessed from different threads provided that an instance is not used by two different threads simultaneously.
Gauging classes measuring and processing operations	May be executed in different threads with no blocking even if these gauges perform their measuring operations in the same image. Multiple CPU usage will be optimal.	A single instance cannot be read / modified by two threads .
EWorldShape		A single instance cannot be read / modified from different threads.
EasyOCR and EasyBarCode EOOCR and EBarCode	Different instances may be created and used from different threads.	A single instance cannot be accessed from several threads.
EChecker	Different instances may be used from different threads.	
EasyMatrixCode EMatrixCodeReader	Multiple CPU usage will be optimal.	A single instance cannot be used by several threads. A single MatrixCode cannot be used in multiple threads.

2. 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
DepthMap ₆	EGrabberBridge::EGrabberDepthMap ₆	Coord3D_C ₆
DepthMap8	EGrabberBridge::EGrabberDepthMap8	Coord3D_C8
EImageBW ₆	EGrabberBridge::EGrabberImageBW ₆	Mono ₆
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.

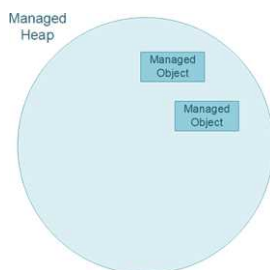
Examples and code snippets

- 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](#)).

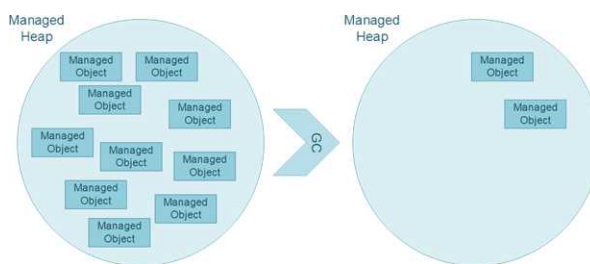
3. Handling the Memory in .NET

The memory management in .NET

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

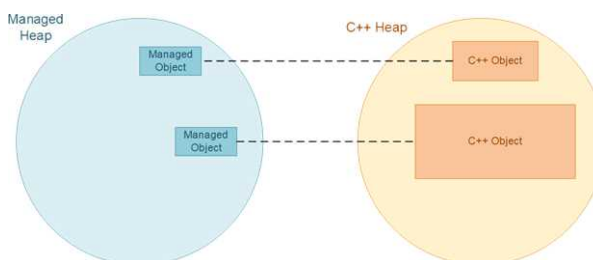


- If that managed heap becomes too full, a specialized process called the *garbage collector* (GC) starts. This process cleans and removes from the memory the objects that are no longer in use.

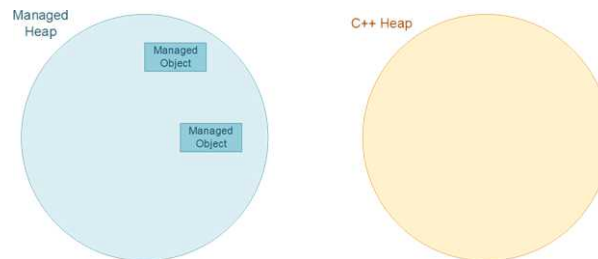


Open eVision specificity

- 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 of too.
 - If called a few times, it does not 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 is important to dispose of it before its host.

- As an example, let's 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 = 50;
encoder.GrayscaleDoubleThresholdSegmenter.LowThreshold = 50;

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

- `GrayscaleDoubleThresholdSegmenter` is a segmenter object nested inside the `EImageEncoder` class.
- Accessing it with the `encoder.GrayscaleDoubleThresholdSegmenter.HighThreshold = 50` 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 = 50;
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 believe that it is not necessary to call `.Dispose()` on a corresponding object.

However, the object is not automatically disposed of 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(↵28, ↵28);
    ...
}

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 of.
 - 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 of the objects in the same scope as their creation.

- The correct safe code is:

```

void UseImage(EImageBW8 imageInFunction)
{
    imageInFunction.SetSize(↵28, ↵28);
    ...
}

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;

```

4. Using Open eVision in a DLL

Follow these guidelines to use Open eVision inside one or several DLLs. Ignoring this may lead to random crashes usually due to incompatibilities between the memory representations and/or the conflicting CRT.



NOTE

Until Open eVision 2.14.1, a bug in the method `Easy::Terminate` may lead to a crash even if you follow these guidelines.

To avoid this, please use Open eVision 2.15 or later.

Depending on your configuration, implement one of the following scenarios. There are presented in the order of increasing complexity:

1. A single DLL called from an application NOT compiled with Open eVision

Inside your DLL, it is mandatory to follow these rules:

- Call `Easy::Initialize` before any other call to Open eVision (with the exception of `Preconfiguration` calls).
- Call `Easy::Terminate` when you do not need Open eVision anymore.
- Destroy any object created in the DLL.
- In the DLL API, do not expose any Open eVision object (otherwise the application is compiled with Open eVision).
- If possible, use only standard C types.

In your application:

- Never use an Open eVision object as a global object.
 - Create the instance only after the DLL initialization.
 - You may use a global pointer or a reference to such an object.
- Never create an Open eVision object or make an Open eVision call inside `DLLMain`.

2. A single DLL called from an application compiled with Open eVision

Inside your DLL and in the application, it is mandatory to follow these rules:

- Call `Easy::Initialize` before any other call to Open eVision (with the exception of `Preconfiguration` calls).
- Call `Easy::Terminate` when you do not need Open eVision anymore.
- Destroy in the DLL any object created in the DLL.
- Destroy in the application any object created in the application.

- In the DLL API, expose Open eVision objects only as pointers or references.
 - You may pass these pointers and references between the DLL and the application if and only if the compilation and linking parameters are strictly the same between the DLL and the application.
 - As this is difficult to achieve, avoid it as much as possible.
 - Be very careful when considering the lifetime of these objects.
- It is highly recommended to use the same exact version (all of Ma.Mi.Re.Bu) of Open eVision, the DLL and the application.
 - This is mandatory if you pass objects around.

In your application:

- Never use an Open eVision object as a global object.
 - Create the instance only after the DLL initialization.
 - You may use a global pointer or a reference to such an object.
- Never create an Open eVision object or make an Open eVision call inside `DLLMain`.

3. Multiple DLLs called from an application NOT compiled with Open eVision

Inside all your DLLs, it is mandatory to follow these rules:

- Call `Easy::Initialize` before any other call to Open eVision (with the exception of `Preconfiguration` calls).
- Call `Easy::Terminate` when you do not need Open eVision anymore.
- Destroy in the same DLL any object created in a DLL.
- In the DLL API, expose Open eVision objects only as pointers or references.
 - You may pass these pointers and references between the DLLs if and only if the compilation and linking parameters are strictly the same between the DLLs.
 - Be very careful when considering the lifetime of these objects.
- In a DLL API, do not expose any Open eVision object (otherwise the application is compiled with Open eVision).
- If possible, use only standard C types.
- It is highly recommended to use the same exact version (all of Ma.Mi.Re.Bu) of Open eVision, the DLL and the application.
 - This is mandatory if you pass objects around.

In your application:

- Never use an Open eVision object as a global object.
 - Create the instance only after the DLL initialization.
 - You may use a global pointer or a reference to such an object.
- Never create an Open eVision object or make an Open eVision call inside `DLLMain`.

4. Multiple DLLs called from an application compiled with Open eVision

Inside your DLLs and in the application, it is mandatory to follow these rules:

- Call `Easy::Initialize` before any other call to Open eVision (with the exception of `Preconfiguration` calls).
- Call `Easy::Terminate` when you do not need Open eVision anymore.
- Destroy in the same DLL any object created in a DLL.
- Destroy in the application any object created in the application.
- In the DLL API, expose Open eVision objects only as pointers or references.
 - You may pass these pointers and references between the DLLs if and only if the compilation and linking parameters are strictly the same between the DLLs.
 - You may pass these pointers and references between a DLL and the application if and only if the compilation and linking parameters are strictly the same between the DLL and the application.
 - As this is difficult to achieve, avoid it as much as possible.
 - Be very careful when considering the lifetime of these objects.
- It is highly recommended to use the same exact version (all of Ma.Mi.Re.Bu) of Open eVision, the DLL and the application.
 - This is mandatory if you pass objects around.

In your application:

- Never use an Open eVision object as a global object.
 - Create the instance only after the DLL initialization.
 - You may use a global pointer or a reference to such an object.
- Never create an Open eVision object or make an Open eVision call inside `DLLMain`.