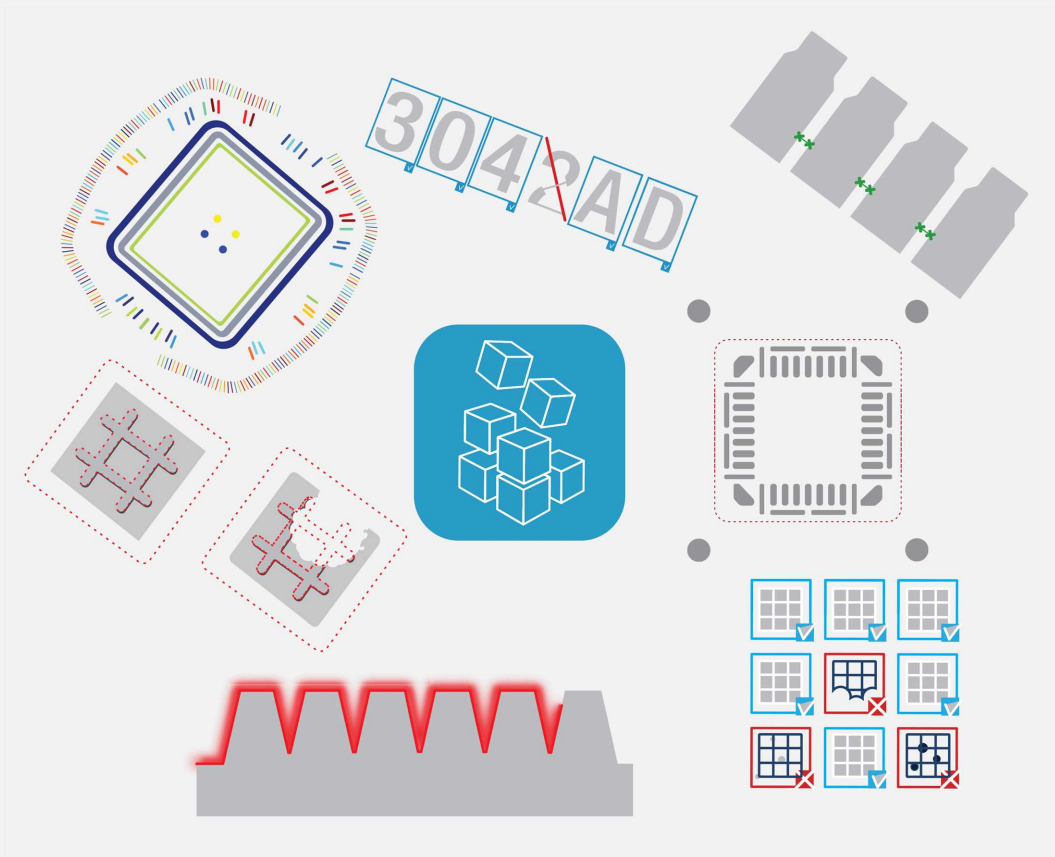


Open eVision

Examples in C++



This documentation is provided with **Open eVision 24.02.0** (doc build **1198**).
www.euresys.com

This documentation is subject to the General Terms and Conditions stated on the website of **EURESYS S.A.** and available on the webpage <https://www.euresys.com/en/Menu-Legal/Terms-conditions>. The article 10 (Limitations of Liability and Disclaimers) and article 12 (Intellectual Property Rights) are more specifically applicable.

Contents

PART I : STARTING UP	8
1. Installing Open eVision	9
1.1. Installing on Windows	9
1.2. Installing on Linux	12
2. Managing the Licenses	15
2.1. Activating the Licenses	15
2.2. Selecting the Licensing Model	15
3. Conventions	17
3.1. Conventions for Strings	17
3.2. Image Coordinate Systems	17
3.3. Image and Depth Map Buffer	19
4. Basic Operations	21
4.1. Memory Allocation	21
4.2. Loading a Pixel Container File	22
4.3. Saving a Pixel Container File	23
4.4. Drawing in Open eVision	25
4.5. 3D Rendering of 2D Images	28
4.6. Vector Types and Main Properties	29
4.7. ROI Main Properties	33
4.8. Arbitrarily Shaped ROI (ERegion)	35
4.9. Flexible Masks	57
4.10. Profile	61
PART II : GENERAL PURPOSE LIBRARIES	63
1. EasyImage - Pre-Processing Images	64
1.1. Intensity Transformation	64
1.2. Thresholding	67
1.3. Arithmetic and Logic	68
1.4. Linear Filtering	71
1.5. Non-Linear Filtering	72
1.6. Geometric Transforms	77
1.7. Noise Reduction and Estimation	79
1.8. Scalar Gradient	82
1.9. Vector Operations	82
1.10. Canny Edge Detector	84
1.11. Harris Corner Detector	85
1.12. Overlay	87
1.13. Operations on Interlaced Video Frames	87
1.14. Flexible Masks in EasyImage	88
1.15. Computing Image Statistics	89
1.16. Fourier Transform	93
1.17. Gabor Filter	96
2. EasyColor - Pre-Processing Color Images	102
2.1. Bayer Conversion	105
2.2. LUT for Gain/Offset (Color)	109
2.3. LUT for Color Calibration	109
2.4. LUT for Color Balance	110

PART III : MATCHING AND MEASUREMENT TOOLS	112
1. EasyObject - Analyzing Blobs	113
1.1. Image Segmenters	116
1.2. Image Encoder	120
1.3. Holes Construction	122
1.4. Normal vs. Continuous Mode	124
1.5. Selecting and Sorting Blobs	126
1.6. Object Template Matcher	128
1.7. Advanced Features	130
Computable Features	130
Draw Coded Elements	136
Flexible Masks in EasyObject	136
2. EasyGauge - Measuring down to Sub-Pixel	139
2.1. Workflow	139
2.2. Gauge Definitions	140
2.3. Find Transition Points Using Peak Analysis	147
2.4. Find Shapes Using Geometric Models	152
2.5. Gauge Manipulation: Draw, Drag, Plot, Group	154
2.6. Calibration and Transformation	156
2.7. Calibration Using EWorldShape	157
2.8. Advanced Features	160
2.9. Unwarp an Image	162
3. EasyFind - Matching Geometric Patterns	165
3.1. Introduction	165
3.2. Purpose and Principles	165
3.3. Workflow	167
3.4. Using EasyFind	169
3.5. Learn the Model from Images	169
3.6. Learn the Model from Vectors	172
3.7. Find Instances of the Model	174
3.8. Open eVision Studio Tools	178
3.9. Use "Don't Care Areas" in the Model	178
3.10. Setting the Parameters	180
3.11. Learning Parameters	180
3.12. Finding Parameters	185
3.13. Vector Model Parameters	193
4. EasyMatch - Matching Area Patterns	196
4.1. Workflow	196
4.2. Learning Process	197
4.3. Matching Process	199
4.4. Advanced Features	200
5. EChecker2 - Validating Golden Templates	202
5.1. EChecker2	202
5.2. Creating a Model	202
5.3. Inspecting an Image	205
PART IV : TEXT AND CODE READING TOOLS	207
1. List of Supported Codes	208
2. ECodeReader - Unified Interface	211
2.1. Reading Codes	211
2.2. Reading Using a Grid	213
3. EasyBarCode - Reading Bar Codes	214

3.1. Reading Bar Codes	214
3.2. Reading Mail Bar Codes	218
4. EasyBarCode2 - Reading Bar Codes (Improved)	222
4.1. EasyBarCode2 vs EasyBarCode	222
4.2. Reading Bar Codes	223
4.3. Grading Bar Codes	225
4.4. Advanced Features	226
5. EasyMatrixCode - Reading Matrix Codes	230
5.1. EasyMatrixCode vs EasyMatrixCode2	230
5.2. EasyMatrixCode	230
Specifications	230
Supported Symbols	231
Workflow	233
Reading a Matrix Code	233
Learning a Matrix Code	233
Computing the Print Quality	235
Using GS1 Data Matrix Codes	235
5.3. EasyMatrixCode2	236
Specifications	236
Supported Symbols	238
Workflow	240
Reading a Matrix Code	240
Learning a Matrix Code	241
Computing the Print Quality	242
Using GS1 Data Matrix Codes	242
Asynchronous Processing	243
Reading Using a Grid	244
Returning Unreadable Codes	245
Advanced Parameters	245
6. EasyQRCode - Reading QR Codes	247
6.1. Workflow	247
6.2. QR Codes Specifications	248
6.3. Reading QR Codes	252
7. EasyOCR - Reading Texts	256
7.1. Workflow	256
7.2. Learning Process	257
7.3. Segmenting	258
7.4. Recognition	259
8. EasyOCR2 - Reading Texts (Improved)	262
8.1. Introduction	262
8.2. Purpose and Principles	262
8.3. Workflow	263
8.4. EasyOCR2 vs EasyOCR	264
8.5. Using EasyOCR2	264
8.6. Detect the Characters	264
8.7. Set the Topology	269
8.8. Learn the Characters	272
8.9. Recognize the Characters	276
8.10. Open eVision Studio Tools	279
8.11. View Elements in Open eVision Studio	279
8.12. View Results in Open eVision Studio	280
8.13. Setting the Parameters	281
8.14. Segmentation Parameters	281
8.15. Detection Parameters	284
8.16. No Topology Parameters	288

9. Code Grading	290
9.1. What Is Grading?	290
9.2. How to Compute the Grading with Open eVision	290
9.3. ISO/IEC 15416 for 1D Bar Codes	293
9.4. ISO/IEC 15415 for Data Matrix and QR Codes	296
9.5. ISO/IEC 29158 for Data Matrix and QR Codes	302
9.6. SEMI T10-0701 for Data Matrix Codes	304
9.7. Implementation Specifics and Limitations	307
9.8. References	308
PART V : DEEP LEARNING INSPECTION TOOLS	309
1. Deep Learning Tools - Inspecting Images with Deep Learning	310
1.1. Purpose and Workflow	310
1.2. Deep Learning Studio and Additional Resources	313
1.3. Engines and Hardware Support (CPU/GPU)	317
1.4. Managing the Dataset and the Annotations	322
Images and Labels	322
Adding Images	325
Editing the Label of an Image	327
Editing the Segmentation of an Image	328
Editing the Objects of an Image	329
ROI and Mask	331
1.5. Managing the Dataset Splits	334
1.6. Using Data Augmentation	337
1.7. Training a Deep Learning Tool	340
1.8. Using a Deep Learning Tool	343
1.9. Benchmarking a Deep Learning Tool	344
2. EasyClassify - Classifying Images	349
2.1. Tool and Images	349
2.2. Validating the Results	353
2.3. Classifying New Images	355
2.4. Benchmarks for EasyClassify	358
3. EasySegment - Detecting and Segmenting Defects	363
3.1. Unsupervised vs Supervised Modes	363
3.2. EasySegment Unsupervised	364
Tool and Configuration	364
Validating the Results	367
Applying the Tool to New Images	369
Benchmarks for EasySegment Unsupervised	370
3.3. EasySegment Supervised	373
Tool and Configuration	373
Using the Supervised Segmenter	375
Evaluating the Results	378
Benchmarks for EasySegment Supervised	382
4. EasyLocate - Locating Objects and Defects	386
4.1. Tool and Configuration	386
4.2. Locating Objects	391
4.3. Validating the Results	394
4.4. Benchmarks for EasyLocate	396
PART VI : 3D PROCESSING TOOLS	400
1. Easy3D - Using 3D Toolset	401
1.1. Basic Concepts	401

1.2. Static Methods	406
1.3. Point Cloud	409
Mapping Attributes	409
Normals and Curvatures	409
Coordinates Transformations	409
Reducing a Point Cloud	410
Managing Planes	412
Aligning	415
Using Spheres	418
1.4. Mesh	418
1.5. ZMap	420
Generating a ZMap	420
Creating a Point Cloud from a ZMap	422
Managing the Coordinates	423
1.6. 3D Viewer	424
1.7. Photometric Stereo	431
Photometric Stereo and Process	431
Calibration	433
Computation and Results	435
Processing the Results with Open eVision Tools	438
Optimizing your Setup	440
Improving the Results	442
2. Easy3DLaserLine - Laser Line Extraction and Calibration	445
2.1. Laser Triangulation	445
2.2. The Laser Line 3D Acquisition Pipeline	446
2.3. Laser Line Extraction	447
2.4. Software vs Hardware Line Extraction	451
2.5. Calibration	454
2.6. Object-Based Calibration Guidelines	457
3. Easy3DObject - Extracting 3D Objects	466
3.1. Purpose and Workflow	466
3.2. Object Features	467
3.3. Extracting and Using Objects	473
3.4. Use Case - Inspecting a PCB	477
4. Easy3DMatch - 3D Alignment and Comparison	483
4.1. Purpose and Workflow	483
4.2. Alignment (E3DAligner)	485
4.3. Comparison (E3DComparer)	489
4.4. Alignment and Comparison (E3DMatcher)	494
4.5. 3D Sensor Fusion (EPointCloudMerger)	502
PART VII : ADVANCED PROGRAMMING	506
1. Multicore Processing	507
2. EGrabberBridge - Using Images from eGrabber Sources	511
3. VimbaXBridge - Using Images from VimbaX Sources	513
4. Handling the Memory in .NET	515
5. Using Open eVision in a DLL	520

PART I
STARTING UP

1. Installing Open eVision

1.1. Installing on Windows

[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 them. Only maintenance releases (with the same major and minor versions but a different revision number) 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].

Supported platforms and requirements

Open eVision in C++ - Using the global header

Include the main **Open eVision** header (`Open_eVision.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.

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

- To include the main header 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.

When using only a subset of the libraries and tools, you can include smaller headers to improve the compilation times.

These headers are:

- Easy.h
- Easy3D.h
- EasyBarCode.h
- EasyColor.h
- EasyDeepLearning.h
- EasyFind.h
- EasyGauge.h
- EasyImage.h
- EasyMatch.h
- EasyMatrixCode.h
- EasyObject.h
- EasyOcr.h
- EasyOcv.h
- EasyQRCode.h
- These headers are required to use the tools corresponding to the license sharing their name if you don't use the global header.
- You can include multiple headers together.
 - Thus, an application reading bar codes, QR codes and matrix codes compiles faster if you include EasyBarCode.h, EasyMatrixCode.h and EasyQRCode.h instead of Open_eVision.h.
- As reference, the C++ samples include only the relevant headers.

[Open eVision in C++ - Using library-focused headers](#)

[Open eVision in .NET](#)

Add a reference to the Open_eVision_NetApi.dll in the development environment. You do not need to copy any other DLL.

Retrieving the installation folders programmatically

If you need to programmatically retrieve the installation folders, use the following methods of the **Easy** object:

- `Easy.GetSampleImagesRootPath` to retrieve the sample images installation path.
- `Easy.GetSampleProgramsRootPath` to retrieve the sample programs installation path.
- `Easy.GetResourcesRootPath` to retrieve the resources installation path.

Deprecation warnings in Open eVision

- The **Open eVision** tools and libraries are an evolving product, and regular improvements are performed.
 - Sometimes, to implement these improvements, the API of a library must evolve too.
 - In this case, the old API is preserved alongside the new one, but is marked as deprecated, indicating that it might be removed in the future.
- The deprecations are tagged within the **Open eVision** headers using the `[[deprecated]]` attribute (`[obsolete]` for .NET).
 - **Open eVision** raises warnings if you use these deprecated methods in your code.
 - You can safely ignore these warnings as they only indicate that you should update these API features because, in the future, they might be removed.

1.2. Installing on Linux

Two archive files are provided for the Linux operating systems:

- `open_evision-linux-*.tar.gz` contains the **Open eVision** libraries and the **Neo License Manager**.
- `neo-linux-license-manager-*.tar.gz` contains only the **Neo License Manager**.

NOTE: The packages are available in deb and rpm formats, for Intel x86-64 and ARM64 architectures.

Supported OS

- **Open eVision** and the **Neo License Manager** are designed to be distribution-independent on `x86_64` platforms.
- **Open eVision** is expected to work with all deb/rpm based distributions with `glibc` version 2.17 or newer.
- This release has been validated with the following distributions and their default `gcc` compilers:
 - **Ubuntu LTS** 16.04 to 20.04
 - **CentOS** 7 and 8
 - **Fedora** 33 to 35
- **Open eVision** and the **Neo License Manager** need an SSE4 compatible CPU, 2 GB of RAM of RAM and 2 GB of HDD.

NOTE: 8 GB of RAM are recommended to compile an **Open eVision** application.

Installing the Open eVision library and the Neo License Manager

- Use the apt package manager to install the deb packages:

```
(bash)
# apt install ./neo-linux-license-manager-(x86_64|arm64)-(version).deb ./open_evision-linux-(x86_64|arm64)-(version).deb ./codemeter-lite_7.40.4990.500_(amd64|arm64).deb
```

- Use the dnf or the yum package manager to install the rpm packages:

```
(bash)
# (dnf | yum) install ./neo-license-manager-(version).x86_64.rpm ./open_evision-(version).x86_64.rpm
./Codemeter-lite_7.40.4990-500.x86_64.rpm
```

- **Open eVision** and the **Neo License Manager** depend on CodeMeter, provided by **Euresys** and others packages (such as libc, libgcc, libssl, ca-certificates...) provided by the Dep/RPM repositories.

Installing Qt and QtCreator

- Run the following command:

```
(bash)
# apt install qt5-default qtcreator
```

Installing OpenGL

- Depending on your distribution, it may be required to install additional packages to use the **Neo License Manager** and the 3D Viewer:

```
(bash)
# apt install libglu1-mesa-dev freeglut3-dev mesa-common-dev
```

Using the Neo License Manager

- Start the **Neo License Manager** from your desktop menu.
- If you cannot find the desktop entry, execute the following script:

```
(bash)
# /opt/euresys/neo_license_manager_X_Y/NeoLicenseManager
```

- For more details, see [What is the Neo Licensing System?](#)

Using the Open eVision library

- **Open eVision** is located in /opt/euresys/Open_eVision_X_Y
- The main header of **Open eVision** is Open_eVision.h
- Sample programs are provided in console mode as well as using the (Qt) GUI (the .pro Qt projects files are included).
 - The samples programs are located in /opt/euresys/Open_eVision_X_Y/Sample Programs
 - The sample images are located in /opt/euresys/Open_eVision_X_Y/Sample Images

Using library-focused headers

When using only a subset of the libraries and tools, you can include smaller headers to improve the compilation times.

These headers are:

- Easy.h
 - Easy3D.h
 - EasyBarCode.h
 - EasyColor.h
 - EasyDeepLearning.h
 - EasyFind.h
 - EasyGauge.h
 - EasyImage.h
 - EasyMatch.h
 - EasyMatrixCode.h
 - EasyObject.h
 - EasyOcr.h
 - EasyOcv.h
 - EasyQRCode.h
- These headers are required to use the tools corresponding to the license sharing their name if you don't use the global header.
 - You can include multiple headers together.
 - Thus, an application reading bar codes, QR codes and matrix codes compiles faster if you include EasyBarCode.h, EasyMatrixCode.h and EasyQRCode.h instead of Open_eVision.h.
 - As reference, the C++ samples include only the relevant headers.

Retrieving the installation folders programmatically

If you need to programmatically retrieve the installation folders, use the following methods of the **Easy** object:

- [Easy.GetSampleImagesRootPath](#) to retrieve the sample images installation path.
- [Easy.GetSampleProgramsRootPath](#) to retrieve the sample programs installation path.
- [Easy.GetResourcesRootPath](#) to retrieve the resources installation path.

2. Managing the Licenses

2.1. Activating the Licenses

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 License Manager**, 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.
- For more details, see [What is the Neo Licensing System?](#)

[Previous licensing systems](#)

- The dongle-based licensing system used before 2.13 is 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.

2.2. 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.
 - All (default): this licensing model enables all the others (Neo + LegacyDongle).

3. Conventions

3.1. Conventions for Strings

Since **Open eVision** 23.08, the only character encoding used in the **Open eVision** libraries and tools is UTF-8.

- All methods taking `std.string` as argument expect an UTF-8 encoded `std.string`.
- All methods returning a `std.string` always return it as UTF-8 encoded.

[Backward compatibility on Windows](#)

On **Windows** (but not on **Linux**), there is also a sanitization process to preserve backward compatibility with older releases that didn't use the UTF-8 encoding.

- The content of each input string is checked to ensure it is UTF-8 encoded.
If it is not the case:
 - The string is assumed to be encoded using the current Windows Language for Non-Unicode Programs parameter.
 - It is converted to UTF-8.
- The output strings of all libraries and tools are always UTF-8.



TIP

Despite the presence of this backward compatibility layer it is recommended to use exclusively UTF-8 to interact with **Open eVision** on all platforms to ensure the best performance and compatibility.

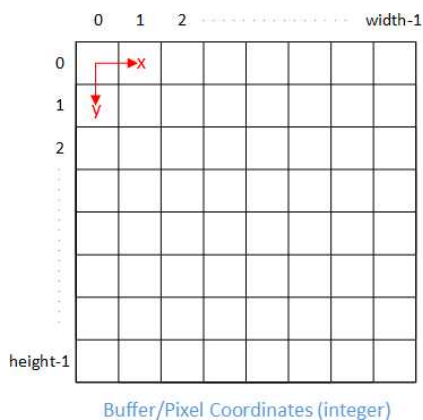
3.2. Image Coordinate Systems

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 library.

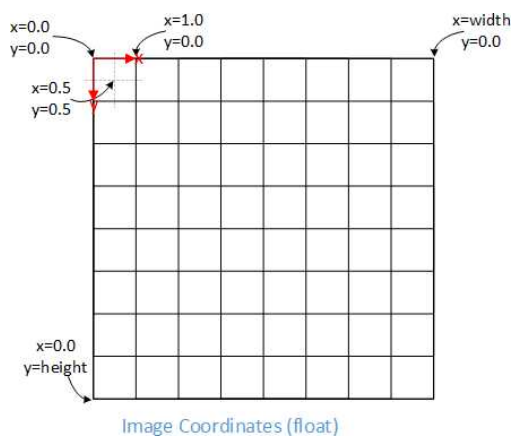
Integer coordinates

- 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.

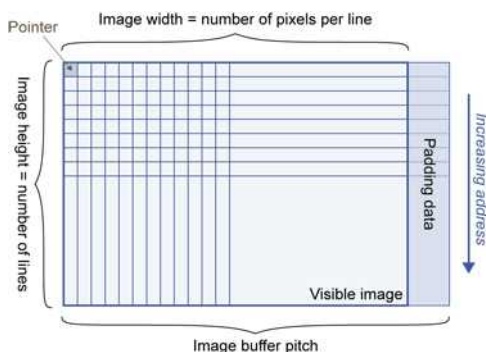


3.3. Image and Depth Map Buffer

The pixels of an image and of a depth map are stored contiguously into a buffer, from left to right and from top to bottom, in the Windows bitmap format (top-down DIB -device-independent bitmap-).

The buffer address is a pointer to the address that contains the top left pixel of the image.

- Image buffer pitch
 - The alignment must be a multiple of 4 bytes.
 - The default pitch in **Open eVision** is 32 bytes for performance reasons.



Memory layout

Image format	Layout	Illustration
EImageBW1	Stores 8 pixels in 1 byte	
EImageBW8 EDepthMap8	Store 1 pixel in 1 byte	
EImageBW16	Stores 1 pixel in 2 bytes	
EImageC15	Stores 1 pixel in 2 bytes - Each color component is coded with 5 bits - The 16th bit is unused	

Image format	Layout	Illustration
EImageC16	Stores 1 pixel in 2 bytes - The colors 1 and 3 are coded with 5 bits - The color 2 is coded with 6 bits	
EImageC24	Stores 1 pixel in 3 bytes - Each color component is coded with 8 bits	
EImageC24A	Stores 1 pixel in 4 bytes. - Each color component is coded with 8 bits - The alpha channel is coded with 8 bits	
EDepthMap32f	Stores 1 pixel in 4 bytes (float format)	

4. Basic Operations

4.1. Memory Allocation

You can construct an image using an internal or an external memory allocation.

Internal memory allocation

The image object dynamically allocates and deallocates a buffer:

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

To declare an image with an internal memory allocation:

1. Construct an image object, for instance `EImageBW8`, either with width and height arguments or using the `SetSize` function.
2. Access a given pixel using one of the multiple available functions.
For example, use `GetImagePtr` to retrieve a pointer to the first byte of the pixel at the given coordinates.

External memory allocation

Control the buffer allocation or link a third-party image in the memory buffer to an **Open eVision** image.

- You must specify the image size and the buffer address.
- When an image object is destroyed, the buffer is unaffected.

 For details, see "[Image and Depth Map Buffer](#)" on page 19 and [Interfacing Third-Party Images](#).

To declare an image with an external memory allocation:

1. Declare an image object, for instance [EImageBW8](#).
2. Create a suitably sized and aligned buffer.
3. Assign the buffer to the image with [SetImagePtr](#).

**NOTE**

Using the copy constructor of the `EImage` object to copy the externally allocated image does not copy the buffer. The copied image points to the same external buffer as the original image.

**NOTE**

If your buffer rows are not aligned on 4 bytes, use [InitializeFromUnalignedBuffer](#) instead of [SetImagePtr](#). Please note that this allocates the memory internally and copies the external buffer into the internal one instead of using the external one.

4.2. Loading a Pixel Container File

Loading images and depth maps

- Use the method [Load](#) to load image data into an image object.
 - It has only the argument path that includes the path, filename and file name extension.
 - The file type is determined by the file format.
 - The destination image is automatically resized according to the size of the image on disk.
- [Load](#) throws an exception when:
 - The file type identification fails.
 - The file type is incompatible with the pixel type of the image object.

NOTE: 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 (supported in ASCII and binary modes).
- [EPointCloud.LoadPLY](#): PLY file (supported only in ASCII mode).
- [EPointCloud.LoadXYZ](#): XYZ file.

4.3. Saving a Pixel Container File

Images and depth maps

- Use the method `Save` of an image or the method `SaveImage` of a depth map or a ZMap to save image data of the object into a file.
 - The argument `Path` includes the path, file name and file name extension.
 - The argument `Image File Type` can be omitted. In this case, the file name extension is used.
- `Save` throws an exception when:
 - The requested image file format is incompatible with the pixel type of the image object.
 - The file name extension is not supported while using the Auto file type selection method.

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



TIP

The images with a width or a height larger than 65,536 must be saved in **Open eVision** proprietary format.

Image File Type arguments

Argument	Image file type
<code>EImageFileType_Auto</code>	(Default) Automatically determined by the file name 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

If the argument is `EImageFileType_Auto` or is missing, the assigned image file type is:

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

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.

4.4. Drawing in Open eVision

Introduction

- Whenever relevant, the **Open eVision** tools provide methods `Draw` to render their contents and/or configuration. This is, for instance, the contents of an `EImage` or the frame of an `EROI`.
- A given tool can have multiple methods `Draw`, usually one for each feature available.
- The **Open eVision** methods `Draw` take an object `DrawAdapter` as their main parameter, and additional parameters for zoom and pan:

```
Tool::Draw(EDrawAdapter* adapter, float zoomX, float zoomY, float panX, float panY);
```

- `zoomX` and `zoomY` are expressed in percentage, 1 is the default value and means no zoom.
- It can be different in the horizontal and vertical directions (which can be useful in the case of non-square pixels for instance).
- If you don't provide a vertical zoom, or set it to 0, it will be set identical to the horizontal one.
- `panX` and `panY` are expressed in pixels, but in image coordinates. It means that the value you pass to `panX` and `panY` are multiplied by the corresponding zoom before being applied.

Example: How to draw an image and a ROI frame on a window under Windows:

```
EImageBW8 image;
EROIBW8 roi;
EWindowsDrawAdapter adapter(windowHdc);
image.Draw(adapter);
roi.DrawFrame(adapter);
```

Graphical interactions

- You can configure some of the **Open eVision** tools graphically and use the provided methods to put your configuration in place.
- Graphical Interaction-enabled tools provide special parameters to some of their methods `Draw` to draw handles on the tool representation.
- To capture the user interactions with those handles, these tools also provide two specialized methods:
 - `HitTest` detects if a handle is under the mouse when providing it with the current cursor coordinates. You typically use this test during a mouse button down event.
 - `Drag` moves the detected handle to the given coordinates. This in turn modifies the tool configuration to match the new handle position. `Drag` is typically associated with the mouse button up event.

NOTE: `HitTest` and `Drag` use the same zoom and pan parameters as `Draw`. You must set them the same way (with the same values) to achieve the desired result.

Draw adapters

- The draw adapters are objects that, in addition to representing the context in which to draw, provide methods to draw the selected primitives in that context.
- They are initialized by providing the targeted context to the constructor.

- Some of the drawing methods provided by the draw adapters are (but are not limited to):
 - `EDrawAdapter::Line` / `Lines` draws one or more lines on the context
 - `EDrawAdapter::Rectangle` / `FilledRectangle` draws a rectangle, filled or not, on the context
 - `EDrawAdapter::Ellipse` / `FilledEllipse` draws an ellipse, filled or not, in the context
 - `EDrawAdapter::Text` / `BackedText` renders a text in the context, with or without background
 - `EDrawAdapter::Image` renders an image in the context
- For more information about the drawing primitives provided by the draw adapters, please refer to the reference documentation.
- To set the color of the primitives, provide a pen and/or a brush and use the methods `EDrawAdapter::SetPen` and `EDrawAdapter::SetBrush`.
 - If you do not provide a pen and/or a brush, the default colors are used.
- To set the font of the text, provide a font with the method `EDrawAdapter::SetFont`.

Standard draw adapters

Open eVision provides a set of off-the-shelf draw adapters that you can use in different situations:

- `EWindowsDrawAdapter` allows to draw on **Windows** systems. To draw on a window, provide the window's HDC to its constructor, or, to draw in an EImage buffer, provide that EImage.
 - It relies on GDI and GDI+ to provide its services.
 - This is the preferred way to draw on **Windows**.
- `QtDrawAdapter` allows you to draw using **Qt** on a QPainter context. To draw on a QPainter context, provide the QPainter to the constructor, or, to draw on an EImage buffer, provide that EImage.
 - You can use the `QtDrawAdapter` both on **Windows** and **Linux**.
 - This is the preferred way to draw on **Linux**.

NOTE: `QtDrawAdapter` is using an external resource (namely **Qt**) and as such is provided as source code in its own header rather than in the global **Open eVision** header. For more information about external and custom draw adapters, see below.
- `EGenericDrawAdapter` is a draw adapter that can only render on an EImage, but it can do it in a consistent manner on all supported OSes.
 - It is available on both **Windows** and **Linux**.

Drawing in an EImage

- As said above, you can draw in an EImage (usually an `EImageBW8` or `EImageC24`) by initializing a draw adapter with that image and using either the **Open eVision** methods `Draw` or the draw adapter drawing primitives:

```
EImageBW8 image;
EMatrixCode code;
EWindowsDrawAdapter adapter(image);
code.DrawPosition(adapter);
```

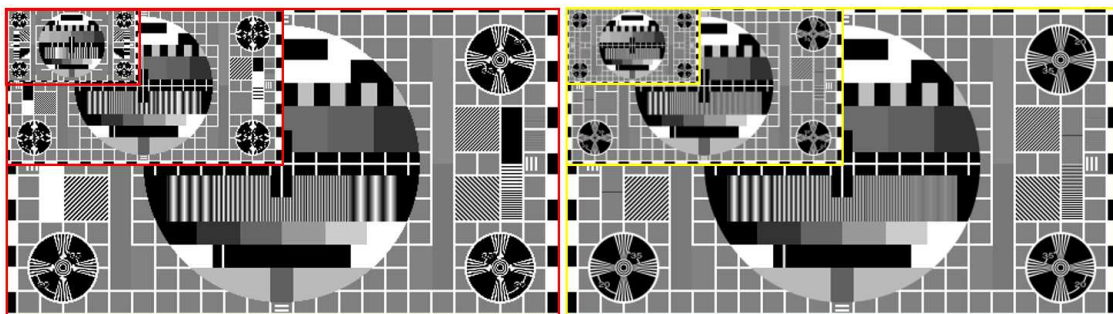
Custom draw adapters

- If you require a draw adapter to render in a specific, unsupported type of context (for ex. a DirectDraw surface, an OpenGL context...), you can build your own draw adapter by deriving from the interface `EExternalDrawAdapter` provided by **Open eVision** and implementing all the required methods.
- Once this work is done, you will be able to use your new, custom draw adapter in the same way as the off-the-shelf ones, taking advantage of **Open eVision** methods `Draw`.
- The provided `QtDrawAdapter` is a draw adapter built using that mechanism, you can use it as a reference on how to build a custom draw adapter. The sources of the `QtDrawAdapter` are bundled with the Qt Samples.

Enhanced Image Display

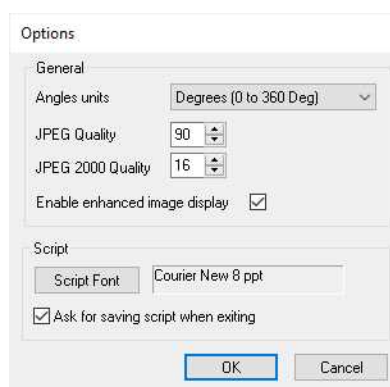
When the enhanced image display mode is enabled, a high-quality interpolation method is used to display the resized images.

- Set `Easy::SetEnableEnhancedImageDisplay(bool)` to `TRUE`, to enable the enhanced image display.
- By default, this option is disabled.
- Enhanced image display has a significant impact on display speed, the drawing can be 4x to 10x slower.
- The drawing of images with `EBW8Vector` or `EC24Vector` used as Look Up Table doesn't support enhanced image display



EnhancedImageDisplay disabled (left) and enabled (right)

- **Open eVision Studio** exposes this option in `View > Option` dialog:

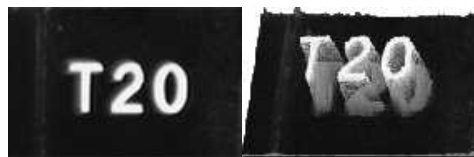


4.5. 3D Rendering of 2D Images

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.

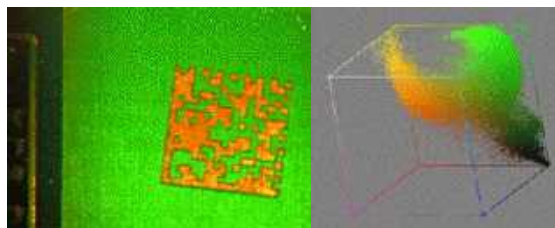


3D rendering

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.

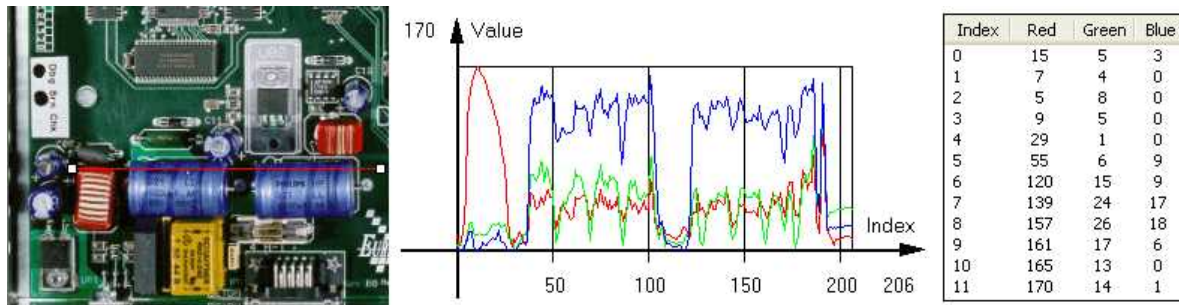


Color histogram rendering

4.6. 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](#))

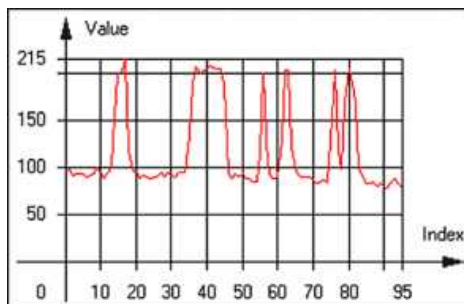
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

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.

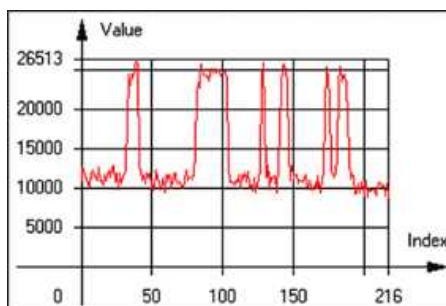
Vector types

- **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...`).



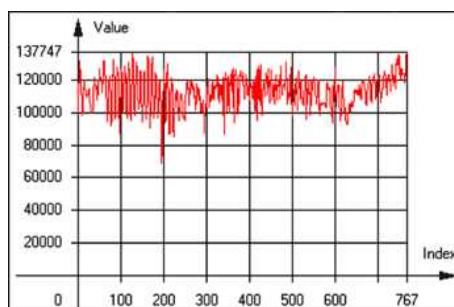
Graphical representation of an **EBW8Vector** (see `Draw` method)

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



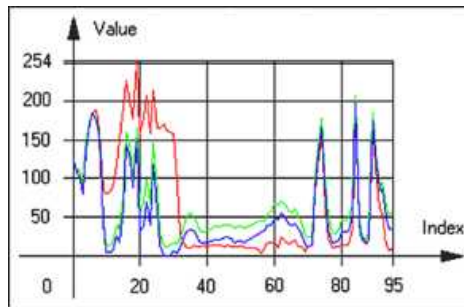
Graphical representation of an **EBW16Vector**

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



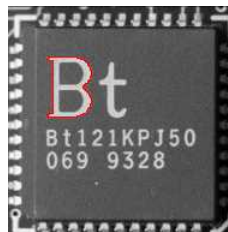
Graphical representation of an **EBW32Vector**

- **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`, ...).



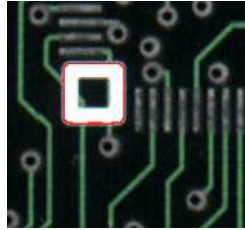
Graphical representation of an **EBW8PathVector** (see `Draw` method)

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



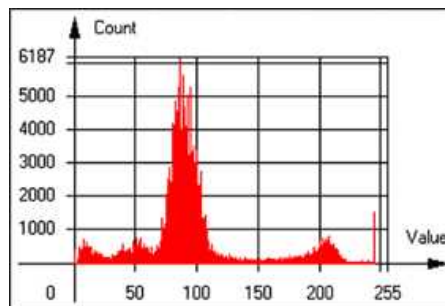
Graphical representation of an **EBW16PathVector** (see `Draw` method)

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



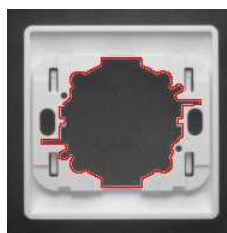
Graphical representation of an `EC24PathVector` (see `Draw` method)

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



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

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



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

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

4.7. ROI Main Properties

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

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

The ROI must be wholly contained in its parent image.

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

Save and load

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

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

ROI Classes

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

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

- [EROIBW1](#)
- [EROIBW8](#)
- [EROIBW16](#)
- [EROIBW32](#)
- [EROIC15](#)
- [EROIC16](#)
- [EROIC24](#)
- [EROIC24A](#)

Attachment

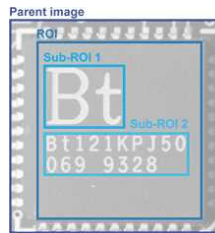
An ROI must be [attached](#) to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.

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

Nesting

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

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



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

Cropping

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



NOTE

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

Resizing and moving

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

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

4.8. 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

Library	Method
	EasyImage::AutoThreshold

Library	Method
	EasyImage: :Copy

Library	Method
	EasyImage: :ConvolKernel

Library	Method
	EasyImage::ConvolSymmetricKernel

Library	Method
	EasyImage: :ConvolveLowpass1

Library	Method
	EasyImage: :ConvolveLowpass2

Library	Method
	EasyImage: :ConvolveLowpass3

Library	Method
	EasyImage::ConvolUniform

Library	Method
	EasyImage::ConvolGaussian

Library	Method
	EasyImage: :ConvolHighpass1

Library	Method
	EasyImage: :ConvolHighpass2

Library	Method
	EasyImage::ConvolGradientX

Library	Method
	EasyImage::ConvolGradientY

Library	Method
	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
	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
	EasyImage::ImageToLineSegment
	EasyImage::ImageToPath

Library	Method
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

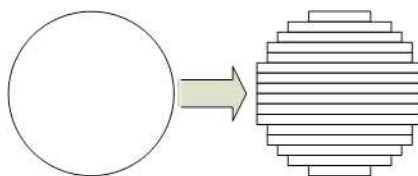
In the future **Open eVision** releases, the support of ERegions will be gradually extended to all operators.

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**

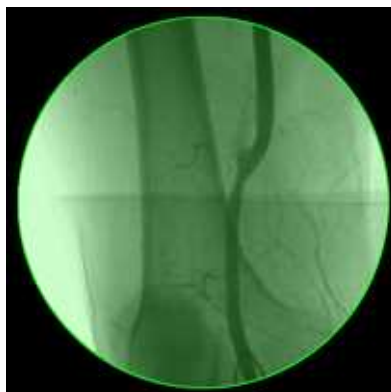
- The contour of an **ERectangleRegion** class is a rectangle.
- Define it using its center, width, height and angle.
- Alternatively, use an **ERectangle** instance, such as one returned by an **ERectangleGauge** instance.



Rectangle region separating a bar code from the background

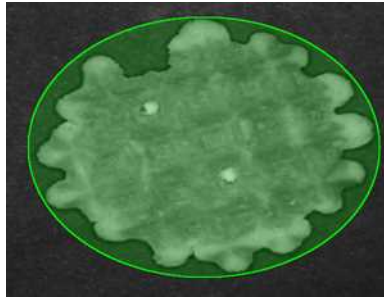
- **ECircleRegion**

- 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](#)

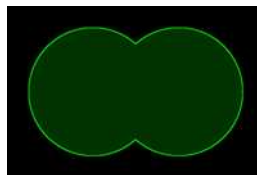
**TIP**

When compatible, **Open eVision** also provides specialized constructors for the geometry-based regions. For instance, [ECircleRegion](#) provides a constructor using an [ECircle](#).

Combining regions

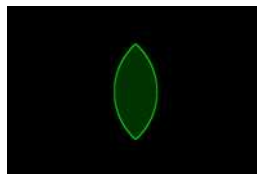
Use the following operations to create a new region by combining existing regions:

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



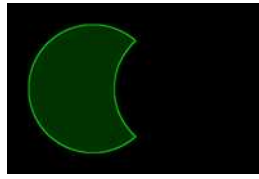
Union of 2 circles

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



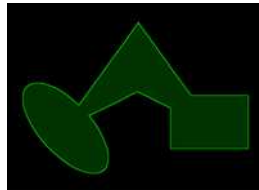
Intersection of 2 circles

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



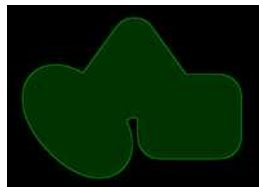
Subtraction of 2 circles

Morphological operations on regions



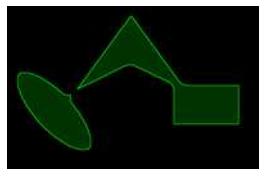
The initial arbitrary region used to illustrate the different morphological operations

- 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.



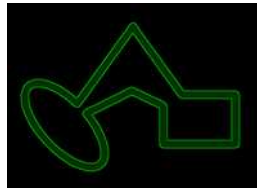
Grow of the arbitrary region

- 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.

4.9. Flexible Masks

ROIs vs flexible masks

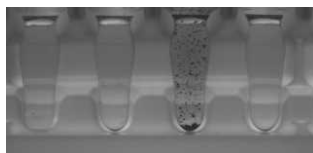
ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 33 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.

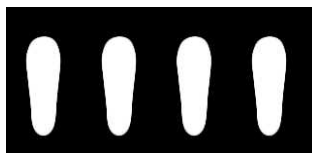
Flexible Masks

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):

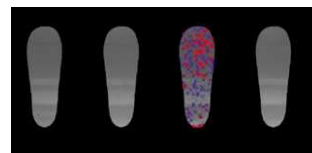
- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image



Associated mask

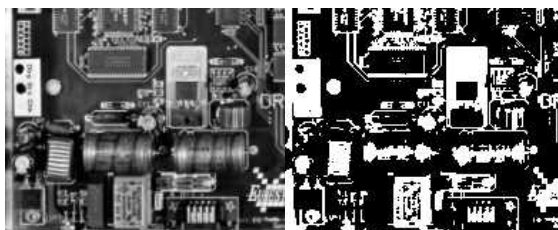


Processed masked image

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

Flexible Masks in EasyImage

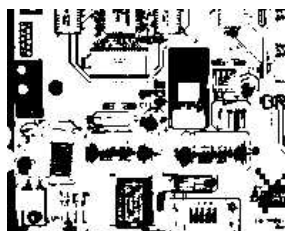
Code Snippets



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.

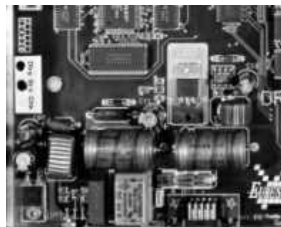
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

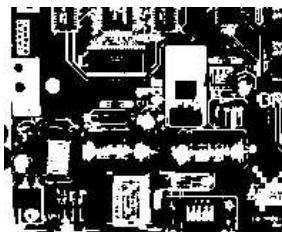


Source image

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())`.

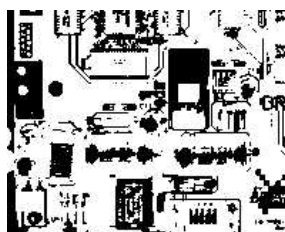
5. Exploit the flexible mask as an argument to `ECodedImage2::RenderMask`.



BW8 resulting image that can be used as a flexible mask

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

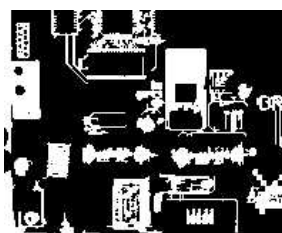
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

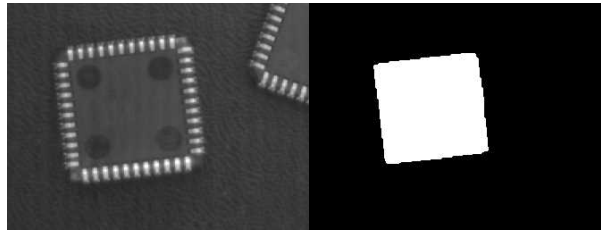
3) `EObjectSelection::RenderMask`: from a selection of blobs

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



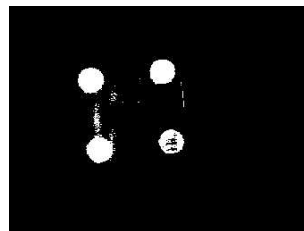
BW8 resulting image that can be used as a flexible mask

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.

4.10. 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

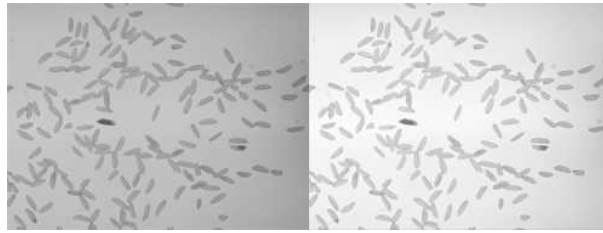
These EasyImage functions change the gray-levels of pixels to increase contrast.

Gain offset

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

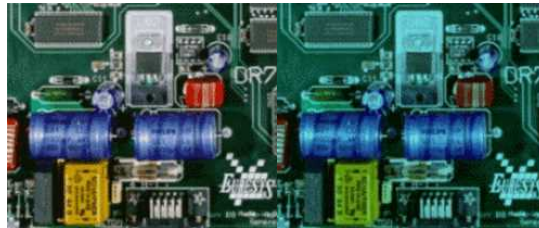
- **gain** adjusts **contrast**. It should remain close to 1.
- **offset** adjusts **intensity** (brightness). It can be positive or negative.
- The resulting values are always saturated to range [0..255].

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).

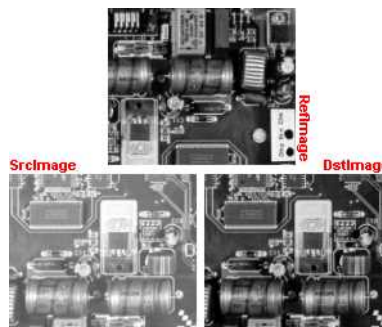


Example of gain/offset applied on a color image

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.

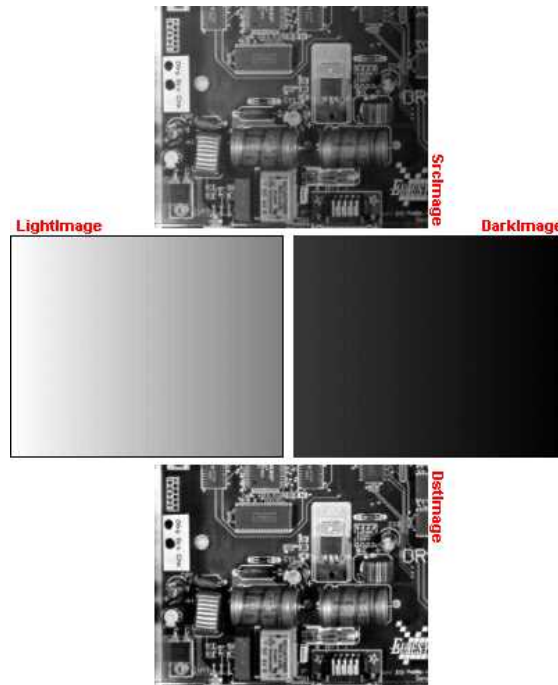


The reference image (from which the average and standard deviation are computed),
the source image (too bright),
and the normalized image (contrast and brightness are the same as the reference image)

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.

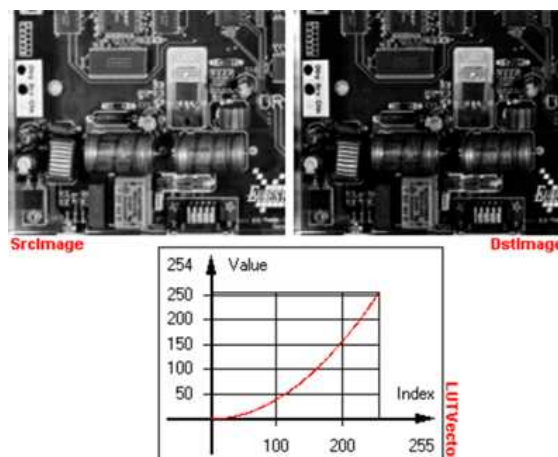
- When one reference image is used, the transformation is similar to an adaptive (space-variant) gain; each pixel in the reference image encodes the gain for the corresponding pixel in the source image.
- When two reference images are used, the transformation is similar to an adaptive gain and offset;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

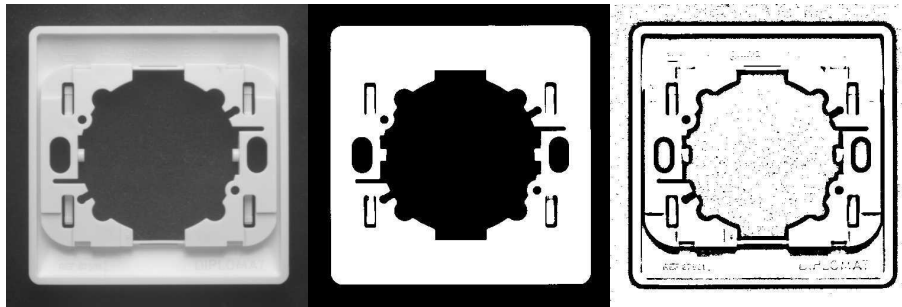
Lut uses a lookup table of new pixel values to replace the current ones - efficient for BW8 and BW16 images. If the transform function never changes, it is best to use a lookup table.



Example of a transform

1.2. Thresholding

Code Snippets



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

- "Thresholding" on page 67 (BW8 and BW16 images only)
- "Thresholding" on page 67 (BW8 and BW16 images only)
- "Thresholding" on page 67 using one or two threshold values
- "Thresholding" on page 67 (computed before using the thresholding function)

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

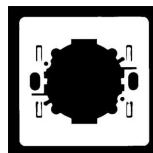
Keys to successful thresholding

- 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

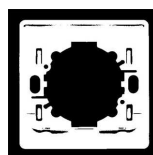
Automatic thresholding

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 [HistogramThresholdBW16](#)) and using that value in a manual thresholding operation.

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

AutoThreshold

When no source image histogram is available, [AutoThreshold](#) can still calculate a threshold value using these [threshold modes](#): `EThresholdMode_Relative`, `_MinResidue`, `_MaxEntropy` and `_Isodata`.

This function supports flexible masks.

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

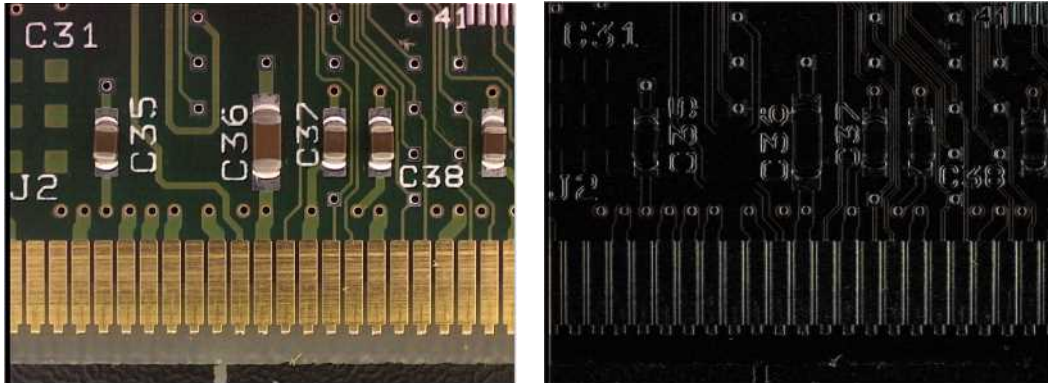
Set

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. Linear Filtering

The convolution functions use linear combinations of neighboring pixels to highlight some features in an image.



A gradient X filter source and the destination image

- All the convolution functions can be destructive, meaning that the destination image overwrites the source image.
 - These destructive operations are faster.
- Most of these functions have an EImageBW8, an EImageBW16 and an EImageC24 equivalent.
- **Open eVision** offers 2 ways of doing convolutions:
 - Use a predefined method with a predefined filter, and in some cases, a kernel size.
 - Create your own [EKernel](#) and use the function [ConvKernel](#).

Predefined filters

The available predefined filters are:

- | | | | |
|---------------------------------|----------------------------------|--------------------------------|--------------------------------|
| • ConvLowpass1 | • ConvGradientX | • ConvPrewitt | • ConvRoberts |
| • ConvLowpass2 | • ConvGradientY | • ConvPrewittX | • ConvUniform |
| • ConvLowpass3 | • ConvLaplacian4 | • ConvPrewittY | • ConvGaussian |
| • ConvHighpass1 | • ConvLaplacian8 | • ConvSobel | • ConvGabor |
| • ConvHighpass2 | • ConvLaplacianX | • ConvSobelX | |
| • ConvGradient | • ConvLaplacianY | • ConvSobelY | |

Customized EKernel

When you use your own kernel:

- You can choose the width and height of the kernel.
 - Use [SetKernelData](#) to fill the convolution coefficients.

- **Open eVision** automatically normalizes the kernel coefficients so that their sum is 1.
 - If you define a **Gain** (multiplying all resulting pixels by that value) and an **Offset** (adding that value to all resulting pixels), they are applied after that normalization.
 - Note that using any gain other than 1 can lead to a saturation in the resulting image and an overflow or an underflow during the internal calculations.
- To rectify any value (to keep only the positive ones for instance), set an **EKernelRectifier**.
- The outside value is used for the border calculations (when the kernel needs values from out of the image on the border of the image to compute the result).
- Another way to create a kernel is by giving the constructor an **EKernelType**. This is quite similar to the way of doing a convolution with a predefined filter.

 The following tutorial illustrates an application using a custom **EKernel** in **Open eVision Studio**: "[Enhancing an X-ray image](#)" on page 1.

1.5. Non-Linear Filtering

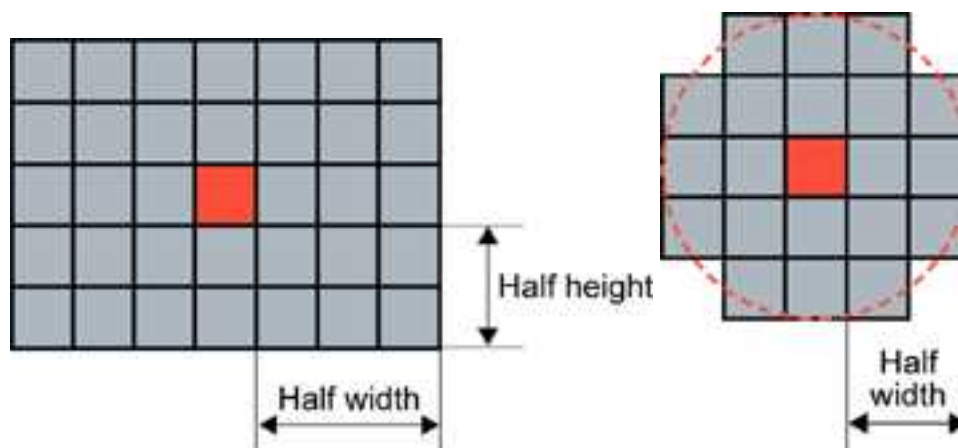
These functions use non-linear combinations of neighboring pixels to highlight a shape, or to remove noise.

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 "[Non-Linear Filtering](#)" on page 72 (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. *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).*

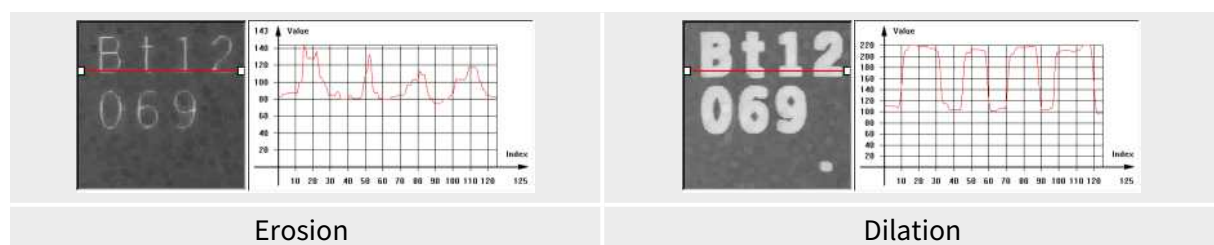
Erosion, Dilation, Opening, Closing, Top-Hat and Morphological Gradient operations all use rectangular or circular kernels of odd size. Kernel size has an important impact on the result.

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.

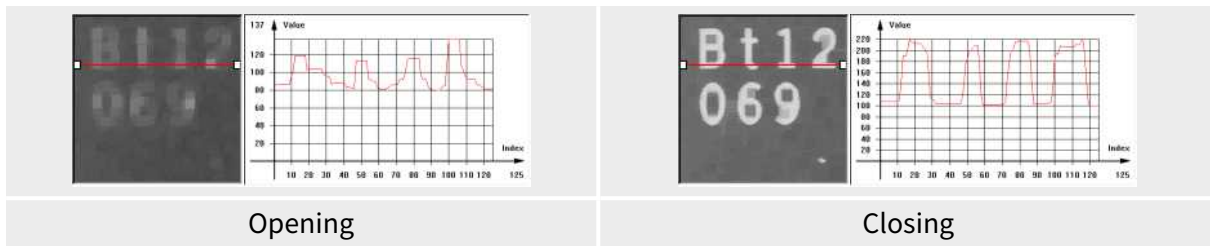


Erosion thins white objects by removing a layer of pixels along the objects edges: [ErodeBox](#), [ErodeDisk](#). As the kernel size increases, white objects disappear and black ones get fatter.

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:

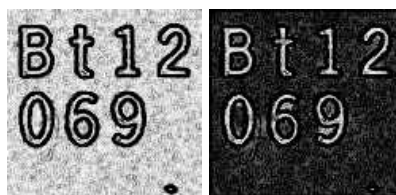
- Thinning: can help edge detectors by reducing lines to single pixel thickness.
- Thickening: can help determine approximate shape, or skeleton.

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

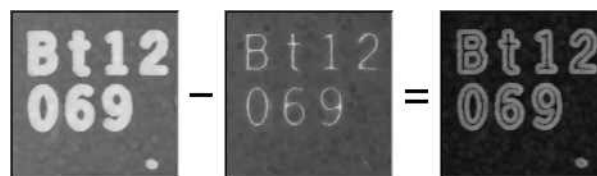
Distance computes the morphological distance (number of erosion passes to set a pixel to black) of a binary image (0 for black, non 0 for white) and creates a destination image, where each pixel contains the morphological distance of the corresponding pixel in the source image.

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.

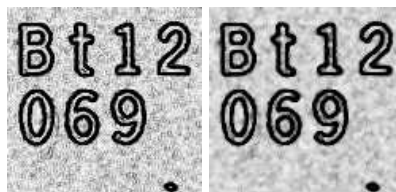
[MorphoGradientBox](#), [MorphoGradientDisk](#).



Dilation – Erosion = Gradient

Median

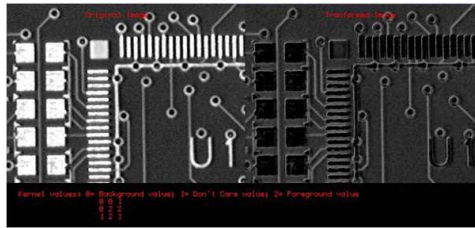
The **Median** filter removes impulse noise, whilst preserving edges and image sharpness. It replaces every pixel by the median (central value) of its neighbors in a 3x3 or larger kernel, thus, outer pixels are discarded.



Median filter: source and destination images

Hit-and-Miss transform

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



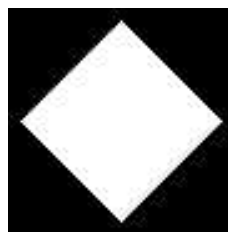
Hit-and-miss transform

The `HitAndMiss` function has three arguments:

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

Example: detecting corners in a binary image.

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

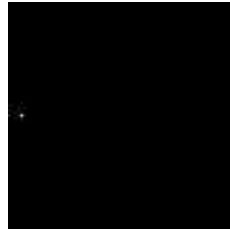


Binary source image

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



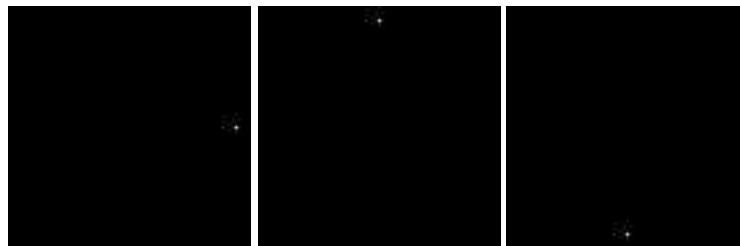
2. Apply the filter on the source image. Note that the resulting image should be properly sized.



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

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

4. Detect the right, top and bottom corners.



1.6. 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).

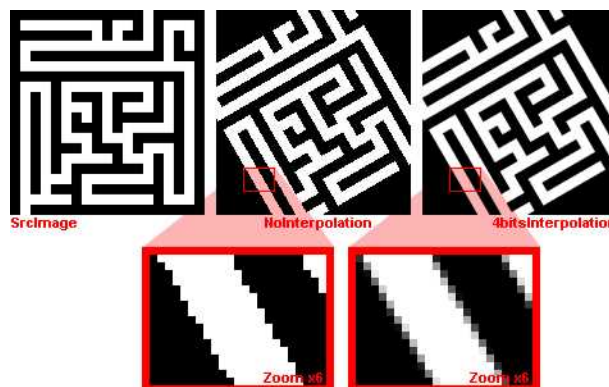
Translation, Scaling and Rotation

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 unwarp a circular ring-wedge shape (such as text on CD labels) into a straight rectangle. A ring-wedge is delimited by two concentric circles and two straight lines passing through the center.
- `EasyImage::SetCircleWarp` prepares warp images for use with function `EasyImage::Warp` which moves each pixel to locations specified in the "warp" images which are used as lookup tables.
- To warp back the image to the circular ring:
 - a. Use `EasyImage::SetInvCircleWarp` with the same parameters as above.
 - b. Use the method `EasyImage::Warp`.
- Additionally, if the LUT-based unwarping is more peculiar:
 - a. Use the method `EasyImage::SetupInverseWarp` to inverse any invertible LUT
 - b. Use it with the method `EasyImage::Warp`.

1.7. 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.

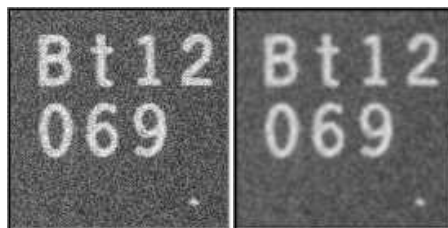
Spatial noise reduction (if you only have 1 image)

Reduces uniform and impulse noise but blurs edges.

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

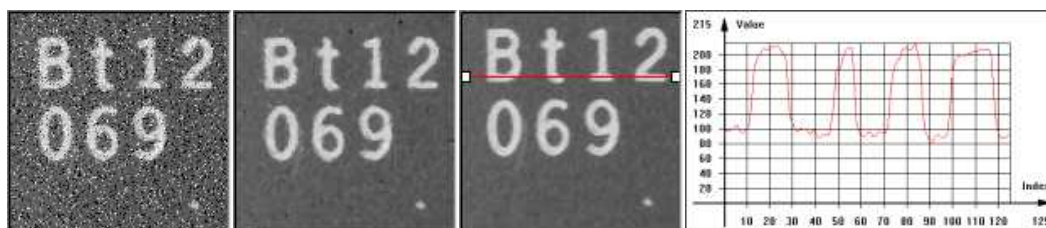
Uses the correlation between neighboring pixel values to perform convolution or median filtering:

- **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 (for example: 5-th largest value in a 3x3 neighborhood). This reduces impulse noise and keeps sharpness.



Impulse noise reduction by median filtering

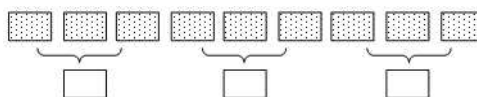
- `EasyImage::Median`
- `EasyImage::BiLevelMedian`

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.

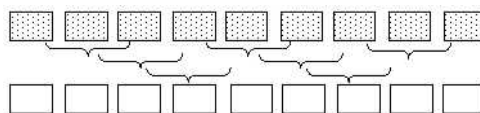
EasyImage provides three ways to minimize noise by means of several images:

- **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.



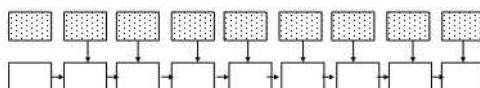
Simple average

- **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.

1.8. Scalar Gradient

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

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

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

This function generates a gradient direction or gradient magnitude map (gray-level image) from a given gray-level image.

For efficiency, a pre-computed lookup-table is used to define the desired transformation.

This lookup-table is stored as a standard [EImageBW8/EImageBW16](#).

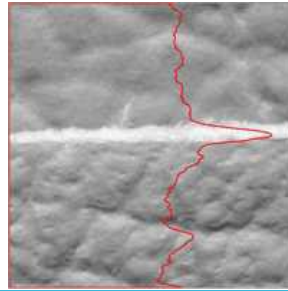
Use [EasyImage::ArgumentImage](#) or [EasyImage::ModulusImage](#) once before calling [GradientScalar](#).

1.9. 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:

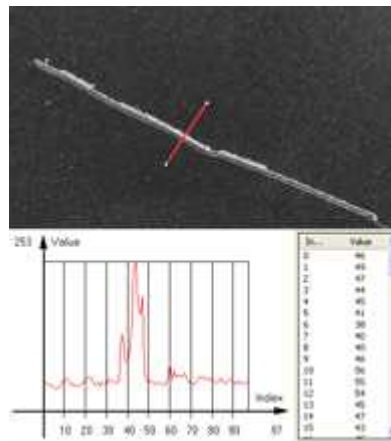
Projection



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](#), [EBW16Vector](#) 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.

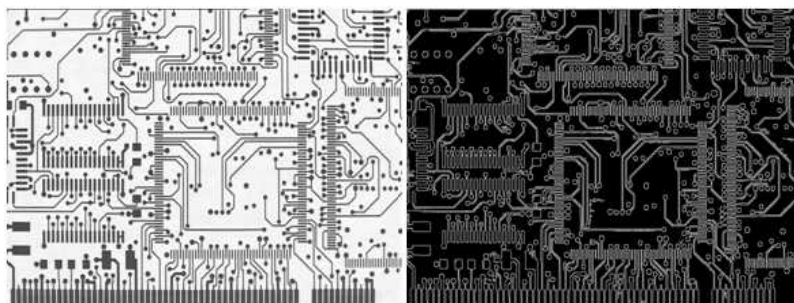
`EasyImage::PathToImage` copies pixel values from a vector or a constant to the pixels of a given path.

1.10. Canny Edge Detector

Code Snippets

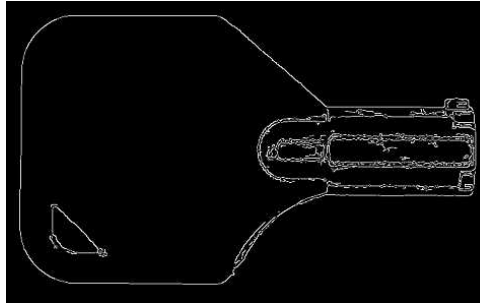
The Canny edge detector facilitates:

- 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:

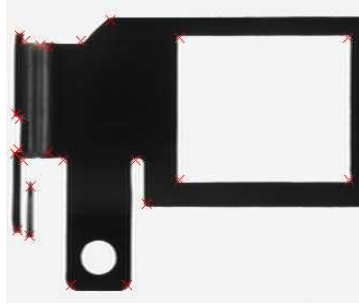
- [Apply](#): applies the Canny edge detector on an image/ROI.
- [GetHighThreshold](#): returns the high hysteresis threshold to consider that a pixel is an edge.
- [GetLowThreshold](#): returns the low hysteresis threshold to consider that a pixel is an edge.
- [GetSmoothingScale](#): returns the scale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

1.11. 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 σ_d .
- Gradient value along the X-axis with regards to the differentiation scale σ_d .
- Gradient value along the Y-axis with regards to the differentiation scale σ_d

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.

1.12. 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.

1.13. Operations on Interlaced Video Frames

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

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

[EasyImage::GetFrame](#) extracts the frame of given parity from an image while [EasyImage::SetFrame](#) replaces the frame of given parity in an image.

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

[EasyImage::RebuildFrame](#) rebuilds one frame of the image by interpolation between the lines of the other frame.

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

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

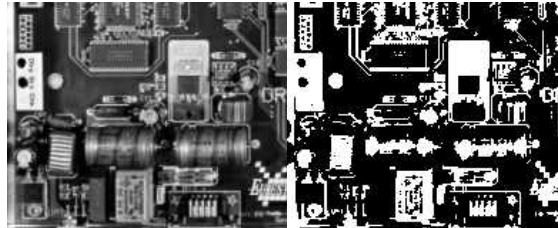
The size of the destination image is determined as follows:

$$dstImage_Width = srcImage_Width$$

$$dstImage_Height = (srcImage_Height + 1 - odd) / 2$$

1.14. Flexible Masks in EasyImage

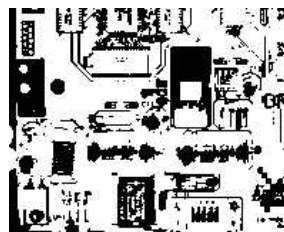
Code Snippets



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).

1.15. 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.

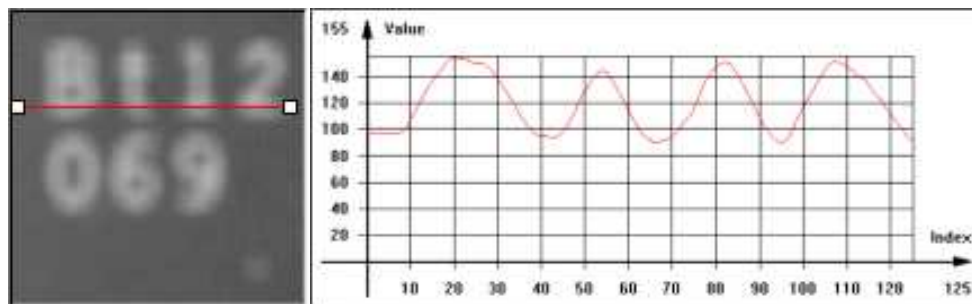


NOTE

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

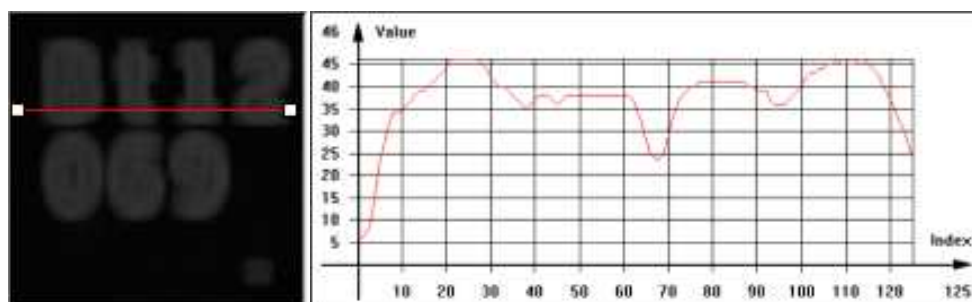
The result of the operation is another image.

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



Sliding window average

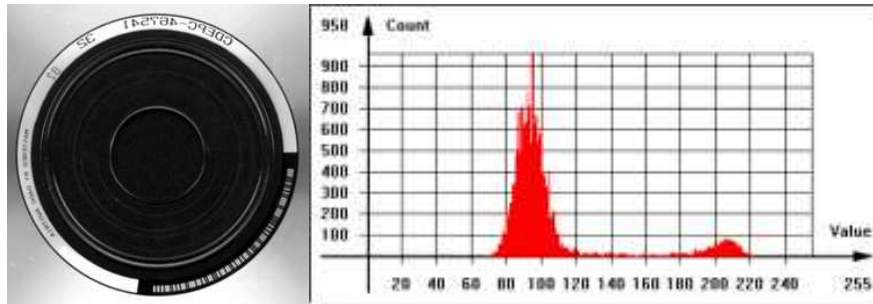
The **local standard deviation**, `EasyImage::LocalDeviation` enhances the regions with a high frequency contents, such as noisy or textured areas.



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 its 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

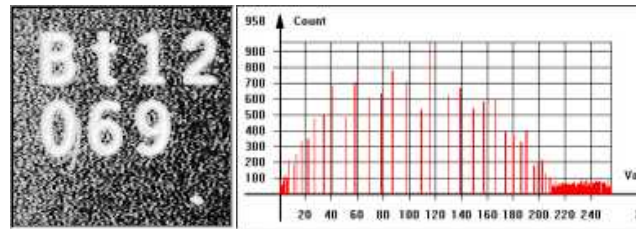
`EasyImage::AnalyseHistogramBW16` provide statistics and thresholding values:

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

Histogram equalization

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

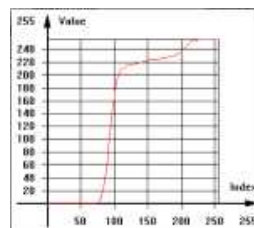
This may be useful to maximize image contrast, or reveal a lot of image details in dark areas.



Equalized image and histogram

Setup a lookup table

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



Equalization lookup table

Image focus

`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.

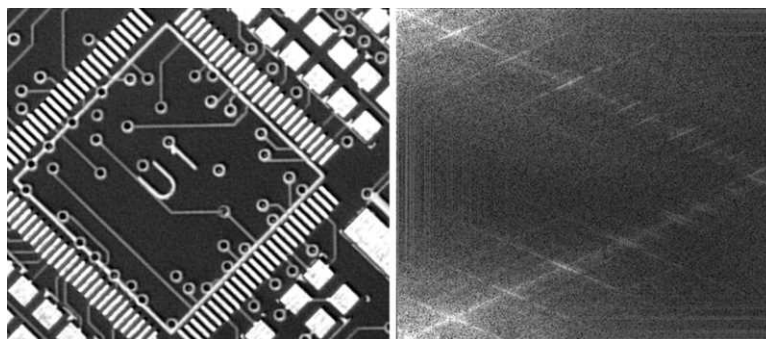
Number of different pixels by comparing 2 images

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

1.16. Fourier Transform

Basics

- The Fourier transform consists in representing an image by its frequential components. You can then process the frequential components image (also called the Fourier image) to perform specific filtering.



Left: the image of a PCB, oriented at 45°
 Right: the corresponding Fourier Image (scaled for visualization)
 with the frequential structures that are scaled as well

- The Fourier images contain complex numbers and are represented by a floating point image (`EImageBW32f`).
- NOTE:** As the floating point images are seldom used in **Open eVision**, you have to process these images manually.
- In **Open eVision**, use an object `EFourierTransformer` to perform the conversion between spatial and Fourier images.
 - Use the method `DirectTransform` to convert a spatial image `EImageBW8`, `EImageBW16` or `EImageBW32f` to an image `EImageBW32f`.
 - Use the method `InverseTransform` to convert images the other way round.

- When converting to a Fourier image and back to a spatial image, you should apply a scale factor of $1/(height \times width)$.
 - The method `DirectTransform` does not apply any scale factor.
 - The method `InverseTransform` applies a scale factor of $1/(height \times width)$.

Frequency format and scaling

As the Fourier images contain redundant data, there are different possible representations.

- **Open eVision** supports the *Packed* and *Complex Extended* formats.
- Use the method `SetFrequencyDomainFormat` to select a format.

The Complex Extended format

The *Complex Extended* is the simplest of these two formats.

- The only subtlety is that the complex numbers are stored in a floating point array, so the complex extended image is twice larger than the spatial image.
- Each odd column contains the imaginary part of a complex number.
- The following table is an illustration of an image in *Complex Extended* format, where the complex pixel at row i and column j is:

$$Re(i,j) + i \times Im(i,j)$$

$Re(0,0)$	$Im(0,0)$...	$Im(0,w-1)$
$Re(1,0)$	$Im(1,0)$...	$Im(1,w-1)$
...
$Re(h-1,0)$	$Im(h-1,0)$...	$Im(h-1,w-1)$

A Fourier image in the Complex Extended format

The Packed format

The *Packed* format is also called *Complex Conjugate Symmetrical*.

- It takes benefit of the conjugate symmetric properties of the Fourier transform of a real image to reduce the Fourier image size and to be the same as the spatial image size.

- The following table is an illustration of an image in *Packed* format, where the complex pixel at row i and column j is:

$$Re(i,j) + i \times Im(i,j)$$

NOTE: There is a difference between images of odd and even height:

$Re(0,0)$	$Re(0,1)$	$Im(0,1)$...	$Re(0,(w-1)/2)$	$Im(0,(w-1)/2)$	$Re(0,w/2)$
$Re(1,0)$	$Re(1,1)$	$Im(1,1)$...	$Re(1,(w-1)/2)$	$Im(1,(w-1)/2)$	$Re(1,w/2)$
$Im(1,0)$	$Re(2,1)$	$Im(2,1)$...	$Re(2,(w-1)/2)$	$Im(2,(w-1)/2)$	$Im(1,w/2)$
...
$Re(h/2,0)$	$Re(h-2,1)$	$Im(h-2,1)$...	$Re(h-2,(w-1)/2)$	$Im(h-2,(w-1)/2)$	$Re(h/2,w/2)$
$Im(h/2,0)$	$Re(h-1,1)$	$Im(h-1,1)$...	$Re(h-1,(w-1)/2)$	$Im(h-1,(w-1)/2)$	$Im(h/2,w/2)$

A Fourier image in packed format, odd height

$Re(0,0)$	$Re(0,1)$	$Im(0,1)$...	$Re(0,(w-1)/2)$	$Im(0,(w-1)/2)$	$Re(0,w/2)$
$Re(1,0)$	$Re(1,1)$	$Im(1,1)$...	$Re(1,(w-1)/2)$	$Im(1,(w-1)/2)$	$Re(1,w/2)$
$Im(1,0)$	$Re(2,1)$	$Im(2,1)$...	$Re(2,(w-1)/2)$	$Im(2,(w-1)/2)$	$Im(1,w/2)$
...
$Re(h/2-1,0)$	$Re(h-3,1)$	$Im(h-3,1)$...	$Re(h-3,(w-1)/2)$	$Im(h-3,(w-1)/2)$	$Re(h/2-1,w/2)$
$Im(h/2-1,0)$	$Re(h-2,1)$	$Im(h-2,1)$...	$Re(h-2,(w-1)/2)$	$Im(h-2,(w-1)/2)$	$Im(h/2-1,w/2)$
$Re(h/2,0)$	$Re(h-1,1)$	$Im(h-1,1)$...	$Re(h-1,(w-1)/2)$	$Im(h-1,(w-1)/2)$	$Re(h/2,w/2)$

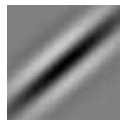
A Fourier image in packed format, even height

- The other coefficients are obtained by using the following properties:
 - For $i = 1, \dots, h-1$ and $j = 1, \dots, w-1$: $Re(i,j) = Re(h-i,w-j)$ and $Im(i,j) = -Im(h-i,w-j)$
 - For $j = 1, \dots, w-1$: $Re(0,j) = Re(0,w-j)$ and $Im(0,j) = -Im(0,w-j)$
 - For $i = 1, \dots, h-1$: $Re(i,0) = Re(h-i,0)$ and $Im(i,0) = -Im(h-i,0)$

1.17. Gabor Filter

Basics

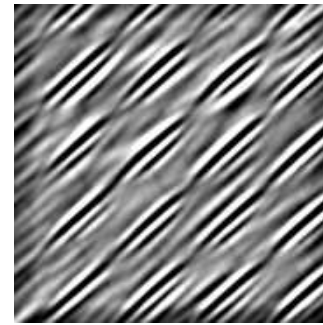
- The Gabor filter is a tool used to analyze textures or to create features for a classifier. It operates by convolving the input image with a Gabor wavelet to detect specific frequency content in an image in a given direction within a localized region around the point or region of analysis.



The Gabor wavelet



The input image

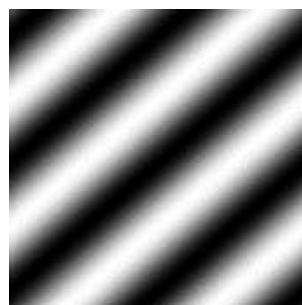


The input image filtered by convolving the wavelet

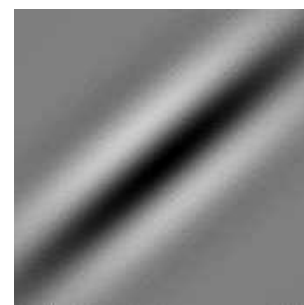
- A key aspect of the Gabor wavelet is that it is composed of two distinct structures which are multiplied with one another:
 - A *Gaussian surface* (a bell-shaped surface) that provides a smooth window. Its spread, its ellipticity and its orientation are influenced by the sigma (σ), the gamma (γ) and the theta (θ) parameters respectively.
 - A *sinusoidal plane wave* with a regular oscillation pattern. Its wavelength and phase are controlled by the lambda (λ) and psi (ψ) parameters respectively. Its orientation is determined by the theta (θ) parameter.



The Gaussian surface



The sinusoidal plane wave

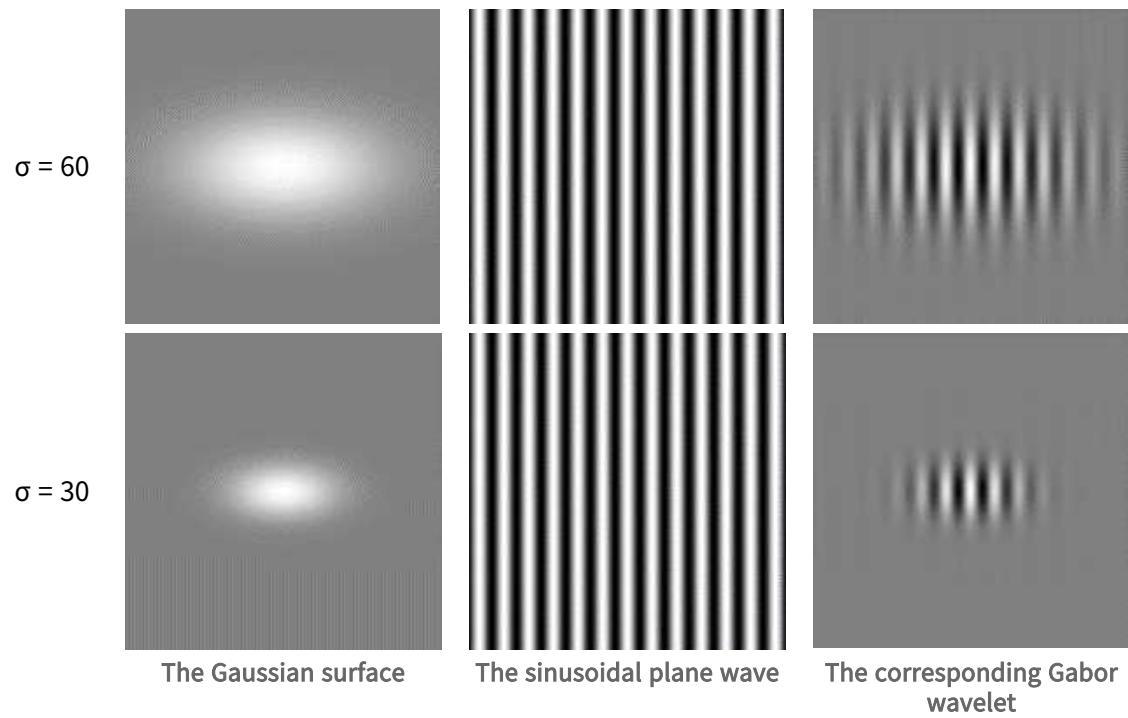


The corresponding Gabor wavelet

Parameter descriptions

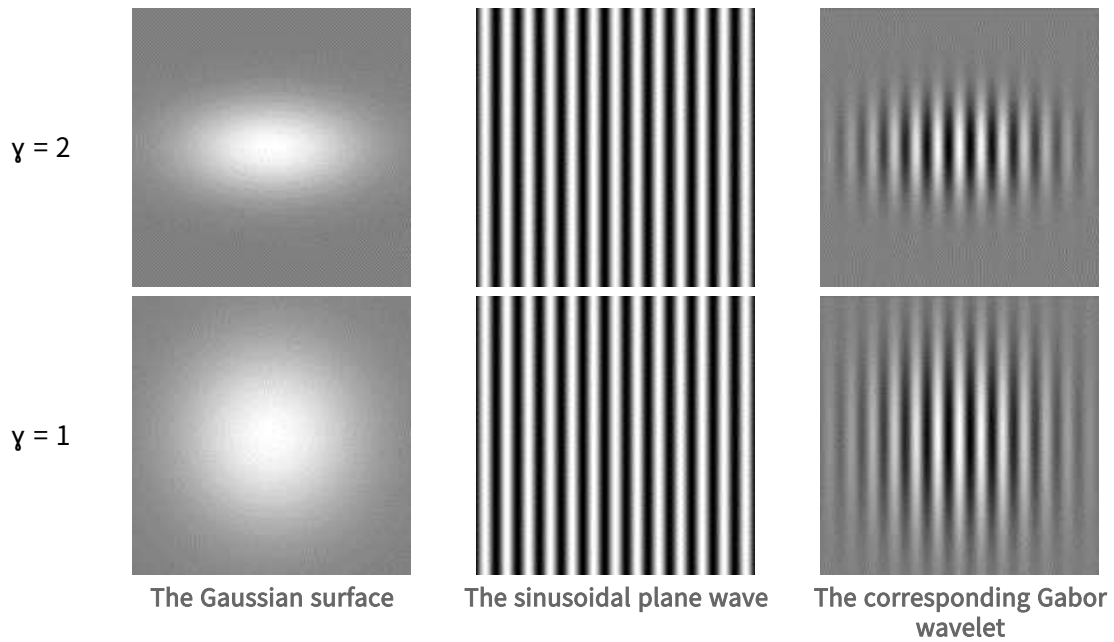
Sigma (σ)

- This parameter influences the spread of the Gaussian surface.
 - A larger σ value results in a wider spread, capturing larger structures in the image.
 - A smaller σ value results in a narrower spread.



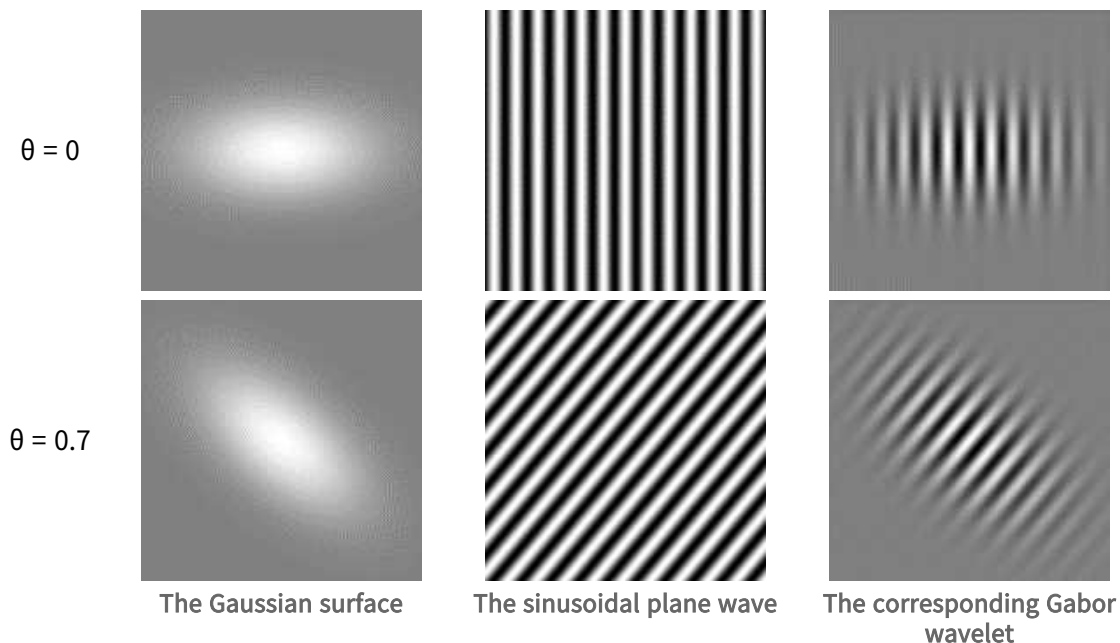
Gamma (γ)

- This parameter specifies the ellipticity of the Gaussian surface.
- With $\gamma = 1$, the support is circular and as γ deviates from 1, the support becomes more elongated.
 - A larger γ value results in a more elongated Gaussian in a direction perpendicular to the stripes.
 - A smaller γ value results in a more elongated Gaussian along the orientation of the parallel stripes of the wave.



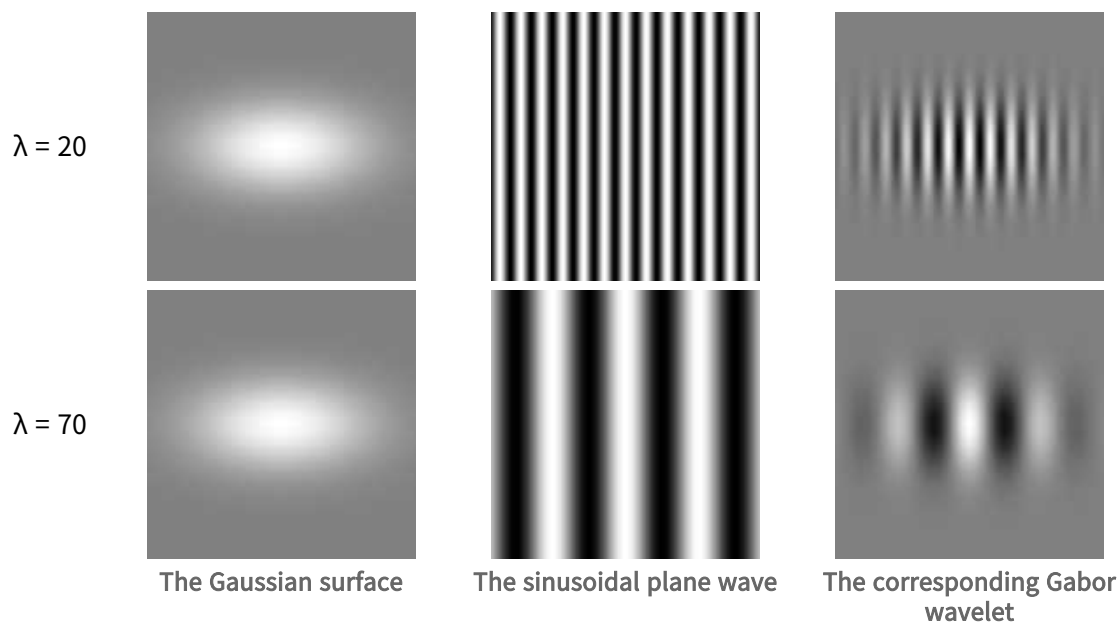
Theta (θ)

- This parameter determines the orientation of both the Gaussian surface and the sinusoidal plane wave.



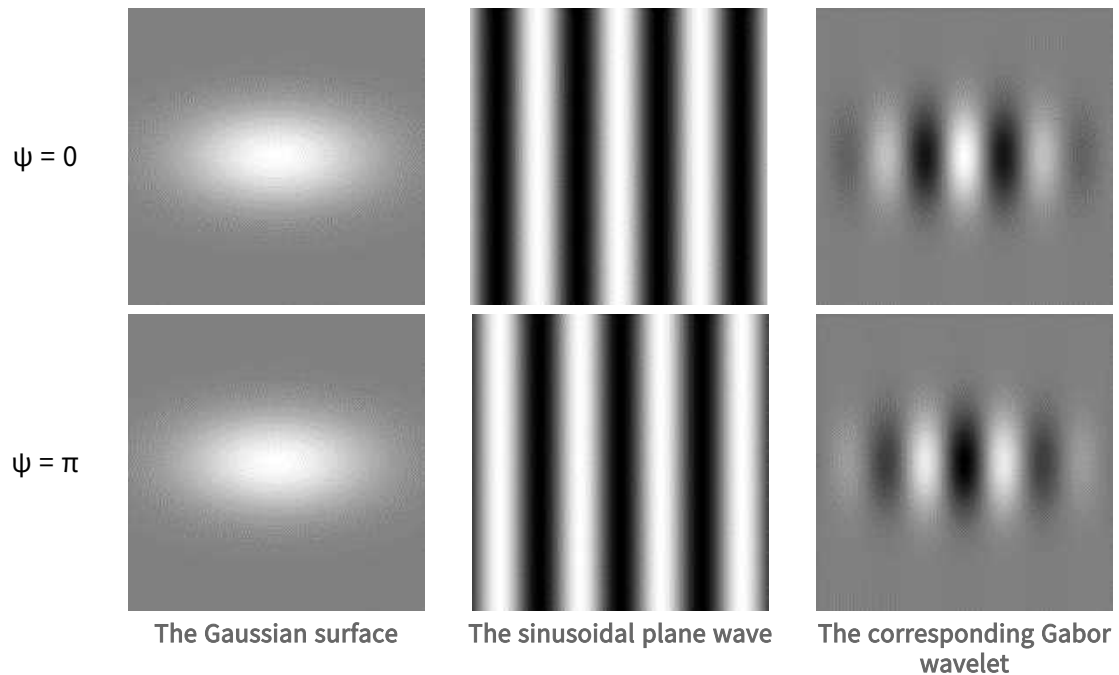
Lambda (λ)

- This parameter controls the wavelength of the sinusoidal plane wave.



Psi (ψ)

- This parameter shifts the sinusoidal wave in the direction perpendicular to the stripes. By adjusting psi, you can control the phase of the sinusoidal plane wave.



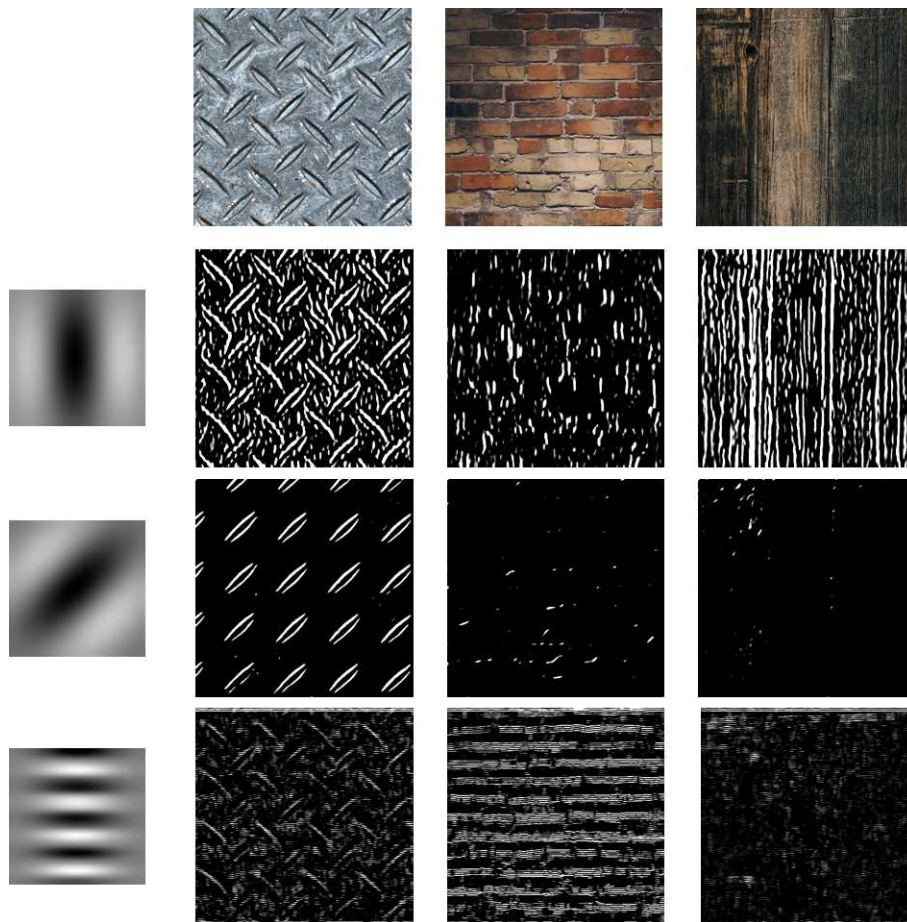
Usage

In **Open eVision**, use the method [ConvolveGabor](#) to apply a Gabor filter to an image.

This method takes several parameters including the source image, kernel size and Gabor parameters:

- Use the method with or without a destination image.
 - If no destination image is provided, the source image is modified in-place.
- You can use the method with a region of interest (ROI).
 - If an ERegion object is provided, the Gabor filter is only applied to the pixels within the region.
- Set the parameter `normalize` to `True` to ensure that the sum of all the values in the Gabor kernel equals 1 by applying a uniform scaling factor to the kernel.
 - If not specified, the default value is `False`, meaning that no normalization is applied.

This visual arrangement displays input images interacting with distinct Gabor wavelets. Each tile reveals the filtered output and illustrates how adjusting the parameters influences the texture extraction during the image processing.



2. EasyColor - Pre-Processing Color Images

Code Snippets

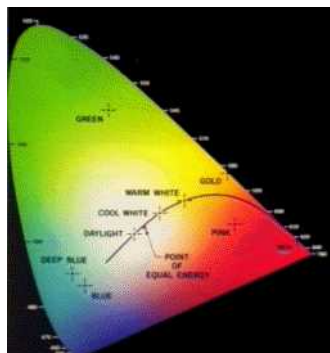
EasyColor makes color image processing as efficient as possible by detecting, classifying and analyzing objects. Several conversion functions mean that any color system can be processed.

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 109: `EasyImage::GainOffset`,
- "LUT for Color Calibration" on page 109: `EColorLookup::Calibrate`,
- "LUT for Color Balance" on page 110: `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.

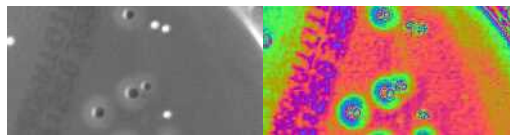
**NOTE**

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.

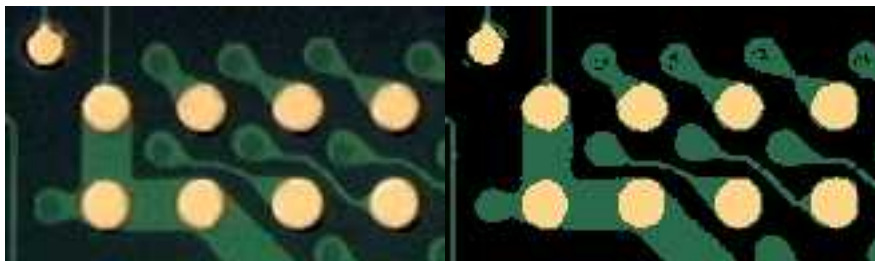
To define pseudo-color shades, you specify a trajectory in the color space of an arbitrary system. You can then pseudo-color using the drawing functions color palette (see Image and Vector Drawing) then save and/or transform it like any other color image.



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



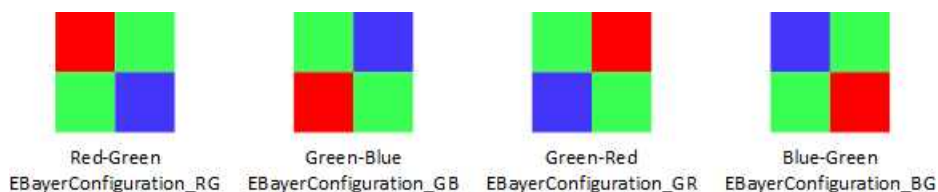
TIP

- 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:



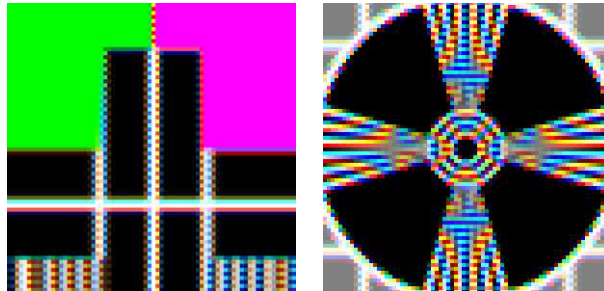
Several methods are available to reconstruct the missing pixels.

- 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.

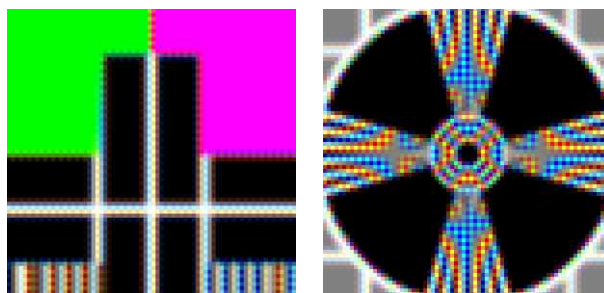
Interpolation modes

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

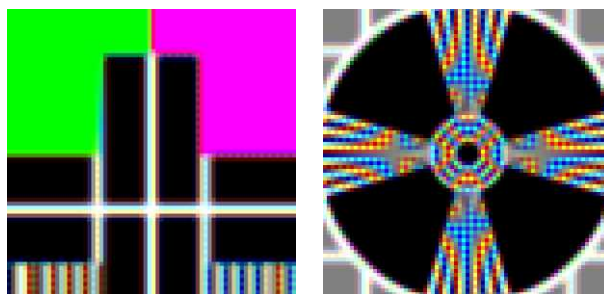
- Mode 0
 - No interpolation
 - Frame rate: 943



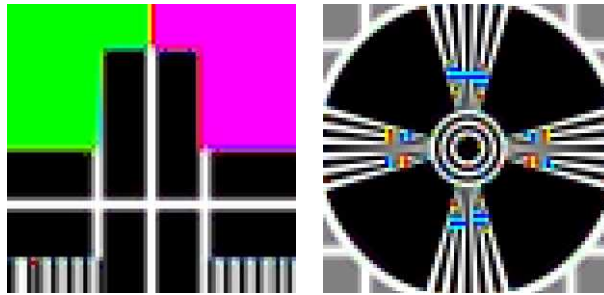
- Mode 1
 - Linear interpolation on a 3x3 kernel
 - Frame rate: 2159



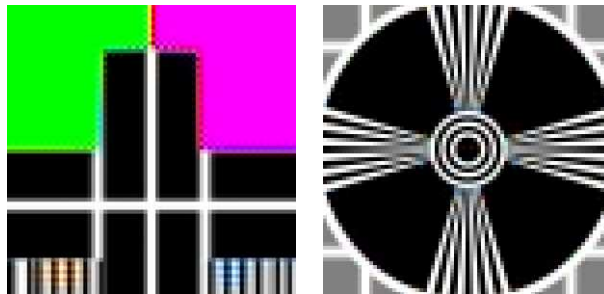
- Mode 2
 - Advanced interpolation on a 3x3 kernel
 - Frame rate: 1303



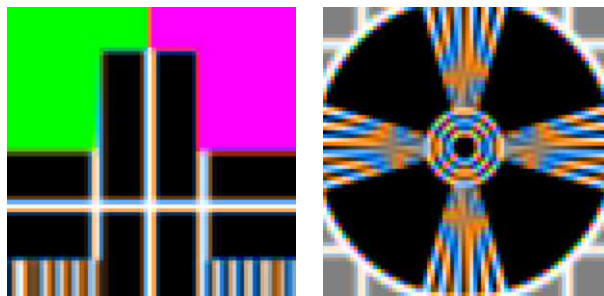
- Mode 3
 - Interpolation on a 5x5 kernel
 - Frame rate: 449



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



- Mode 5
 - Linear interpolation on 2x2 kernel
 - Frame rate: 950



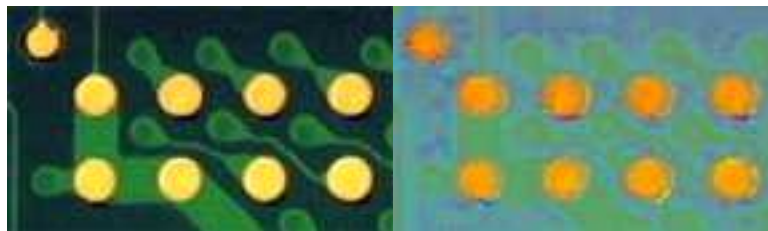
The method `EasyColor::C24ToBayer` is the reciprocal function of `EasyColor::BayerToC24`. It converts RGB color pixel images to Bayer images using the given `EBayerConfiguration`.

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.

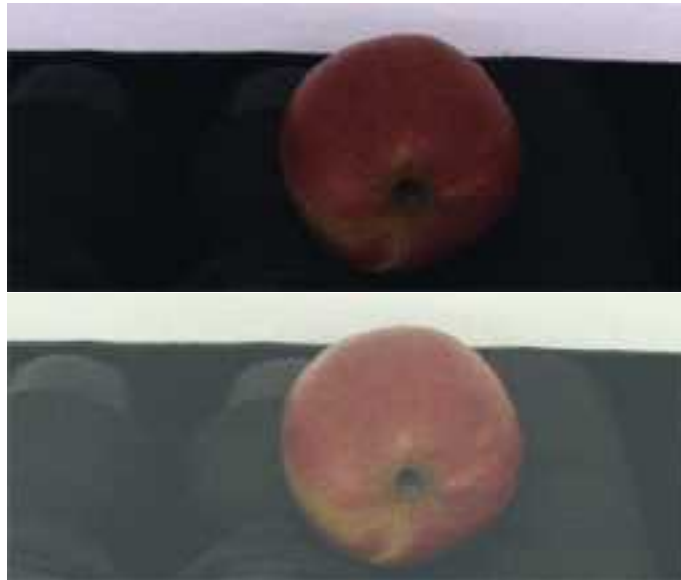
Gamma precompensation cancellation

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.



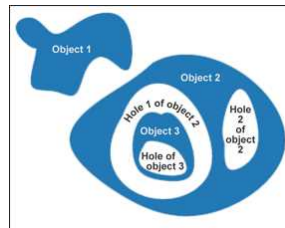
Raw image, and image with white balance and gamma precompensation

PART III
MATCHING AND
MEASUREMENT TOOLS

1. EasyObject - Analyzing Blobs

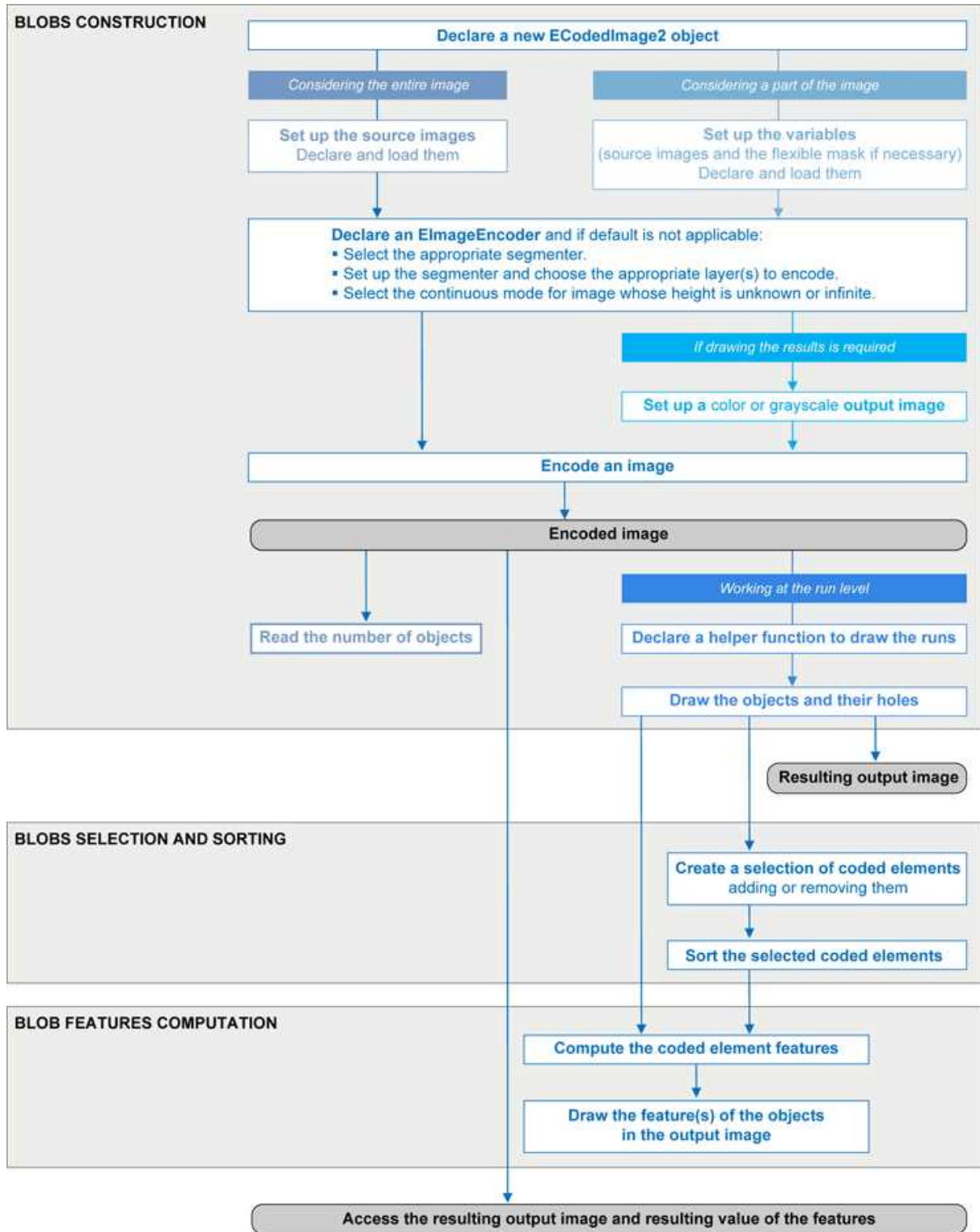
[Reference](#) | [Code Snippets](#)

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:

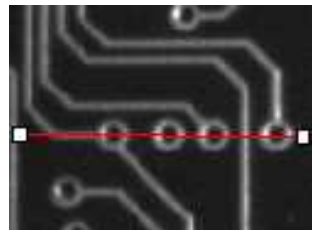
- when the background is not uniform enough,
- when the illumination is not uniform across the image,
- when only differences between the image and a reference image (ideal) are to be enhanced,

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.

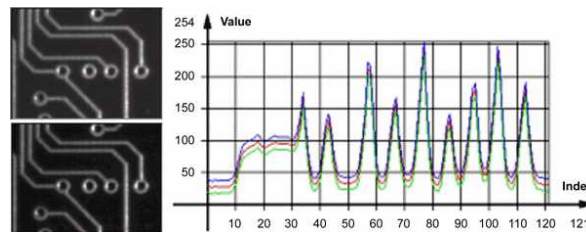
Preparing high and low reference images

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



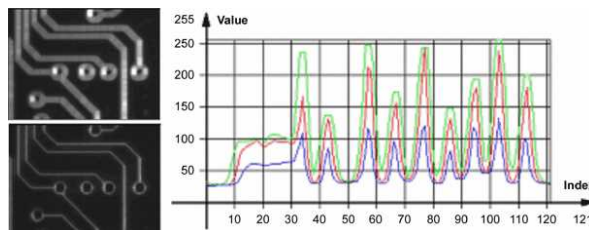
Source image

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



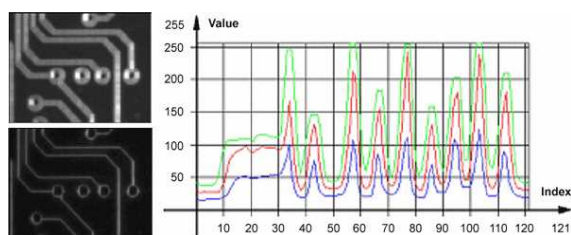
Gray-level tolerance margins

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: [SetLowImageBW16](#) [GetLowImageBW16](#)
- 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 [SetHighImageBW16](#) [GetHighImageBW16](#)
- 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: [SetReferenceImageBW16](#), [GetReferenceImageBW16](#)
- C24: [SetReferenceImageC24](#) , [GetReferenceImageC24](#)

6. Labeled Image

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

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

7. Binary Image

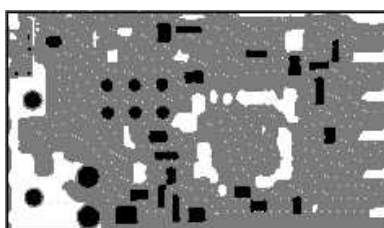
[EBinaryImageSegmenter](#) is applicable to BW1 binary images; it produces coded images with two layers:

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

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

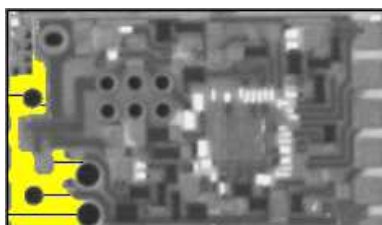
Layer	Set LayerEncoded function name	Get LayerEncoded function name	Default value
Black layer	SetBlackLayerIndex	IsBlackLayerIndex	0
White layer	SetWhiteLayerIndex	IsWhiteLayerIndex	1

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

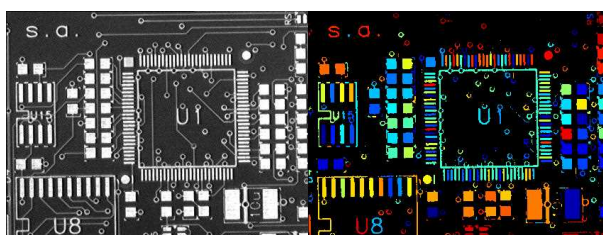
For the sake of computational efficiency, the objects are described as lists of runs. A run is a sequence of adjacent pixels that share homogeneous properties (such as being above a given threshold). These runs are merged in objects by the image encoder.



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.



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

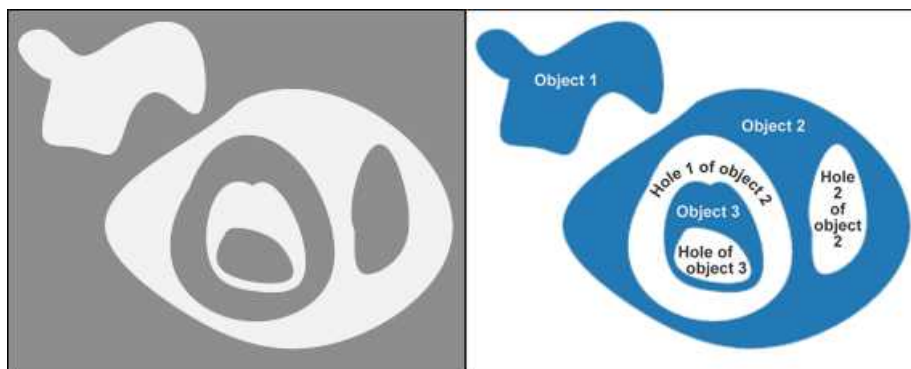
1.3. Holes Construction

Code Snippets

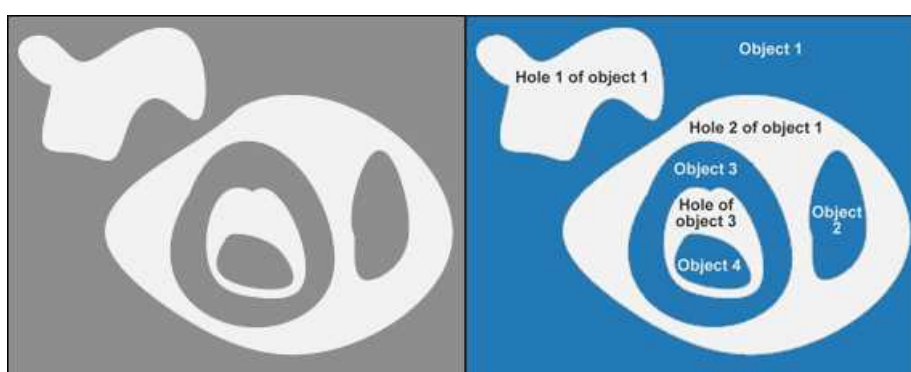
A hole is a set of connected pixels that are entirely surrounded by a parent object (4 or 8 pixels depending on the connexity mode).

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.



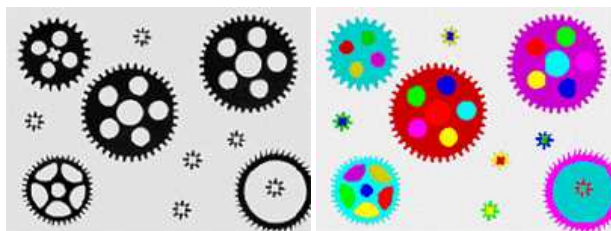
Encoding the white layer (3 objects and 3 holes)



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.

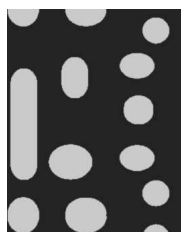
Continuous Mode

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

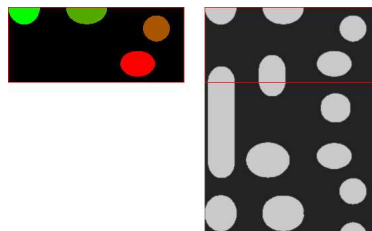


Original image

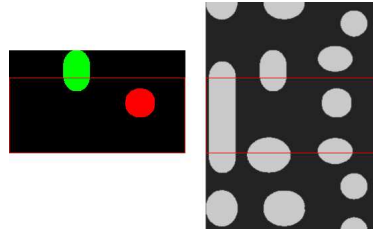


Three chunks of the image

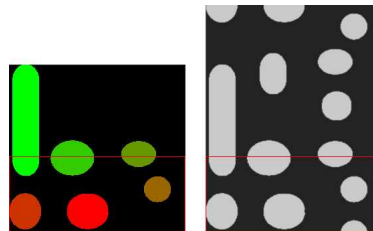
1. **Draw the objects encoded in a layer of a coded image.** This code is essentially the same as in "Browsing Runs" code snippet. The only difference is that an offset can be applied along the Y-axis.
2. **Define a function to draw the objects of a layer.** If a coded image contains objects that were 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, because they touch the border of the chunk.



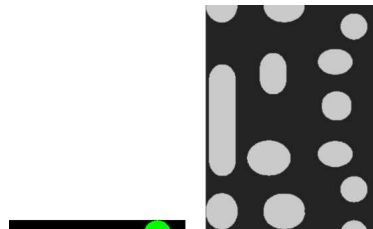
When count reaches 1, one of these two objects becomes completed, which leads to the encoding of the following image. Two other objects are not encoded yet at this time. Here is the result of the encoding of the last chunk (count = 2).



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

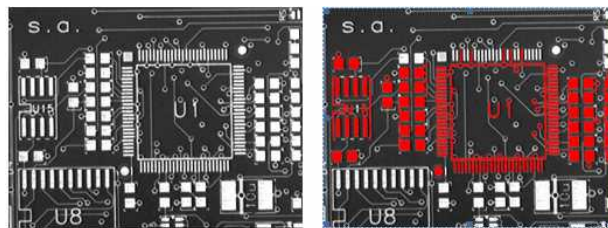
You can clear a previous selection using `EObjectSelection::Clear`.

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.

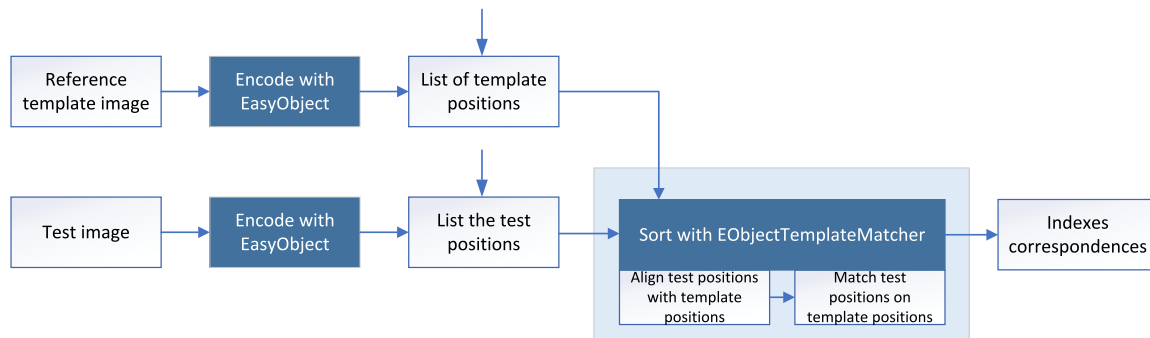


Source image, and selection of objects

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

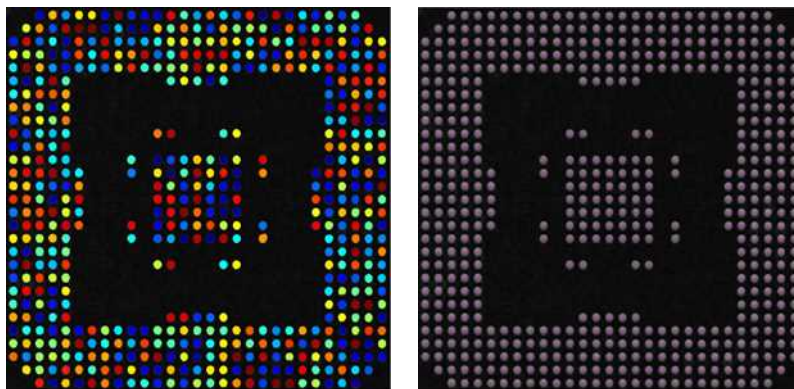
The class `EObjectTemplateMatcher` is a tool designed to align and match the output of `EasyObject` to a reference template. It is designed and developed to handle efficiently thousand of objects.



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>`).

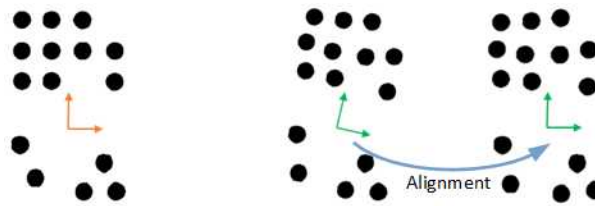


An encoding of the reference image for use as the template.
And the center positions of each object for use in the matching process.

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.

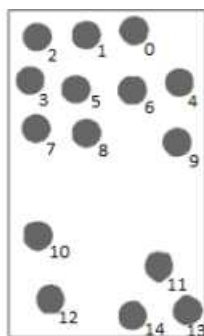


Left: the template.
Right: the alignment of the submitted selection.

- 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.

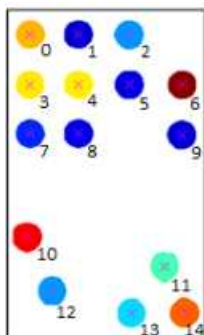
Retrieving the Sorting Results

- 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.

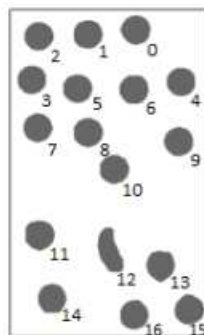


```
NumberOfPairedObjects = 15
SelectionIndexes =
2, 1, 0, 3, 5, 6, 4, 7, 8, 9, 10, 11, 12, 14, 13
TemplateIndexes =
2, 1, 0, 3, 6, 4, 5, 7, 8, 9, 10, 11, 12, 14, 13
UnpairedObjects =
  onlyInTemplate = <>
  onlyInSelection = <>
```

Selection with moved objects

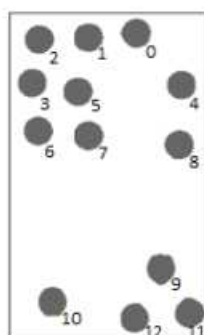


Template with object indexes



```
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, 12, 11
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

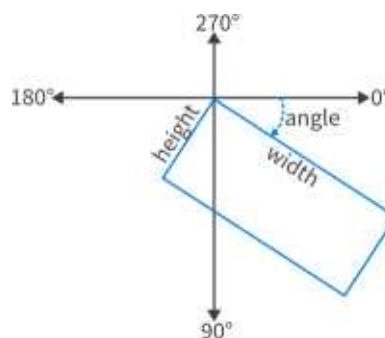
Area (pixel count)	ECodedElement::Area
Feret box (center X and Y, height, width with distinct orientation angles at 22, 45, 68 degrees)	ECodedElement::ComputeFeretBox ECodedElement::GetFeretBox22Box ECodedElement::GetFeretBox22Center ECodedElement::GetFeretBox22CenterX ECodedElement::GetFeretBox22CenterY 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

<p>Bounding box (center X and Y, height, width)</p>	<p><code>ECodedElement::GetBoundingBox</code> <code>ECodedElement::GetBoundingBoxCenter</code> <code>ECodedElement::GetBoundingBoxCenterX</code> <code>ECodedElement::GetBoundingBoxCenterY</code> <code>ECodedElement::GetBoundingBoxHeight</code> <code>ECodedElement::GetBoundingBoxWidth</code></p>
<p>Min. enclosing rectangle (angle, center X and Y, height, width)</p>	<p><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></p>

Feret box

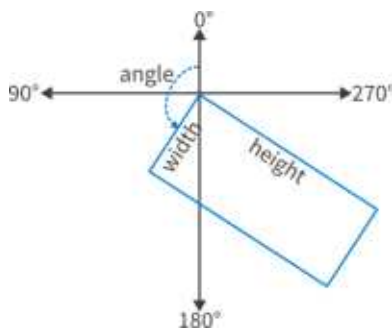
A Feret box is a rectangle with the minimum surface rotated at a specified angle that contains all the pixels center points of an object.

- **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.
- Use `ECodedElement.ComputeFeretBox` to compute a Feret box with an arbitrary angle.
 - The angle is measured clockwise from the X axis to the width side of the rectangle as shown in the image below:



- In the legacy **EasyObject** library, the class `ECodedImage` does not follow the same conventions. The main differences are:
 - The minimum enclosing rectangle corresponds to the Feret box of the legacy **EasyObject** library (features `ELegacyFeature_Feret*`).
 - The method `ECodedElement.ComputeFeretBox` corresponds to the `ECodedImage.LimitAngle` in the legacy **EasyObject** library.

- The angle, width and height in the legacy **EasyObject** library are defined as shown in the image below:



Miscellaneous

Starting point of the object contour (X and Y)	<code>ECodedElement::GetContour</code> <code>ECodedElement::GetContourX</code> <code>ECodedElement::GetContourY</code>
Path of the object contour	<code>ECodedElement::GetContourPath</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):

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

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

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

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

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

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

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

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

Convex Hull

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

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



Graphic Representation

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

Objects	Graphic	Objects	Graphic
Bounding box		Feret box with an angle of 45°	
Contour		Feret box with an angle of 68°	
Convex hull		Gravity center	
Ellipse		Minimum enclosing rectangle	
Feret box		Weighted gravity center	
Feret box with an angle of 22°			

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.

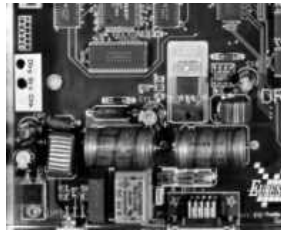
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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.

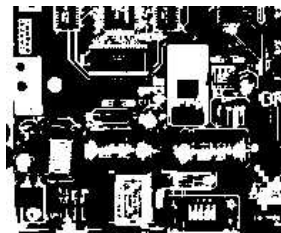
EasyObject functions that create flexible masks



Source image

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

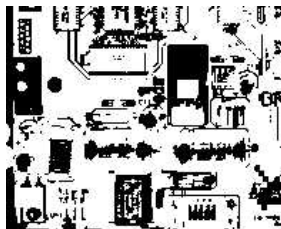
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method GrayscaleSingleThreshold is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers using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

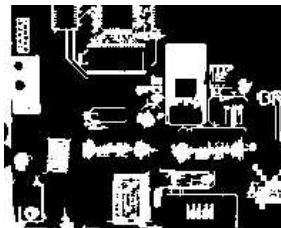
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

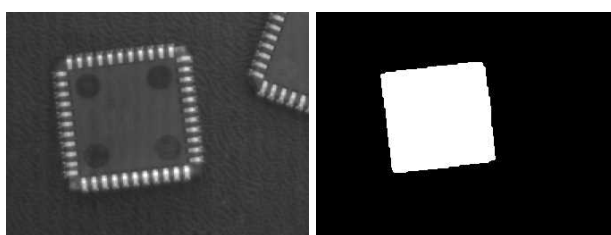
3. EObjectSelection::RenderMask: from a selection of blobs

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



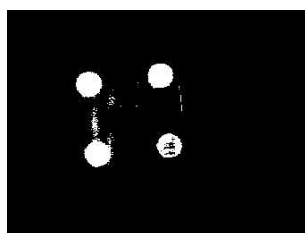
BW8 resulting image that can be used as a flexible mask

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](#):



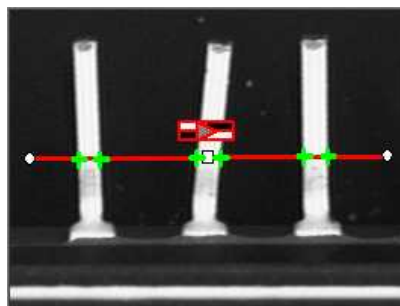
5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

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.



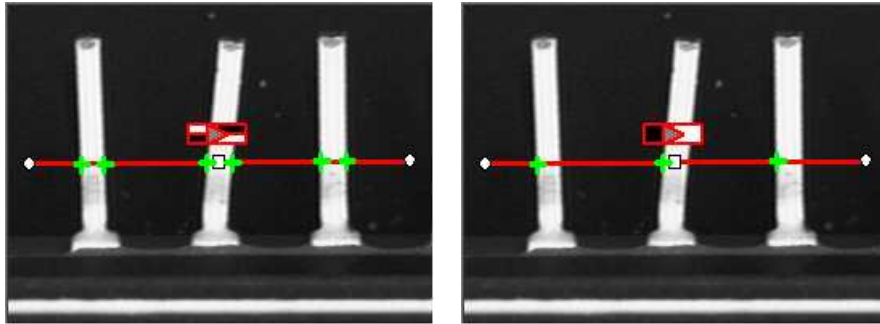
NOTE

Unlike the rest of **Open eVision**, **EasyGauge** uses a pixel center origin (see "[Image Coordinate Systems](#)" on page 17). 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.



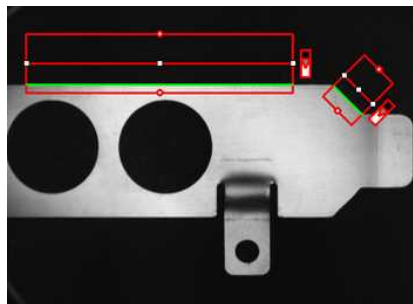
Point location. Contrast-based selection

Usage

- Define and position the gauge, then fit the points with [Measure](#).
- Use the methods [EPointGauge.GetNumMeasuredPoint](#) and [EPointGauge.GetMeasuredPoint](#) to access the results.

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.
 - The method [ELineGauge.GetFound](#) returns TRUE if a suitable line is found.
 - 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.

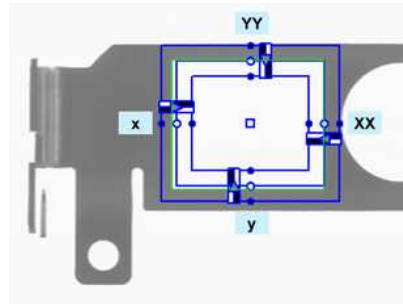
Rectangle Gauge

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.

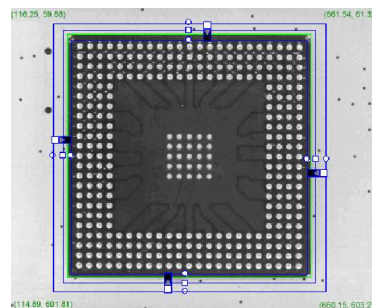
These rules allow to set different parameters for different sides, and measure parallel sides or a corner point instead of the whole rectangle. The four sides are denoted by letters "x", "y", "XX" and "YY" respectively.



Naming conventions for the sides of a rectangle gauge

Usage

- Define and position the gauge, then use [Measure](#) to fit the lines.
 - The method [ERectangleGauge.GetFound](#) returns TRUE if a suitable rectangle is found.
 - 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.

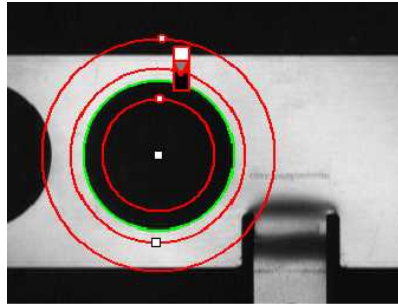


Locating a rectangle's corners

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.
 - The method [ECircleGauge.GetFound](#) returns TRUE if a suitable rectangle is found.
 - 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.

Wedge gauge

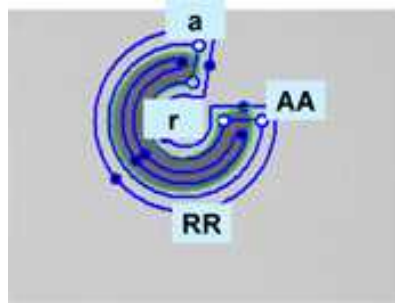
The placement of a [wedge gauge](#) is defined by its nominal position (given by the coordinates of its [center](#)), its nominal [inner](#) and [outer radius](#) ([inner](#) and [outer diameter](#)), its [breadth](#) (difference between radii), the [angular position](#) from where it extends and its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



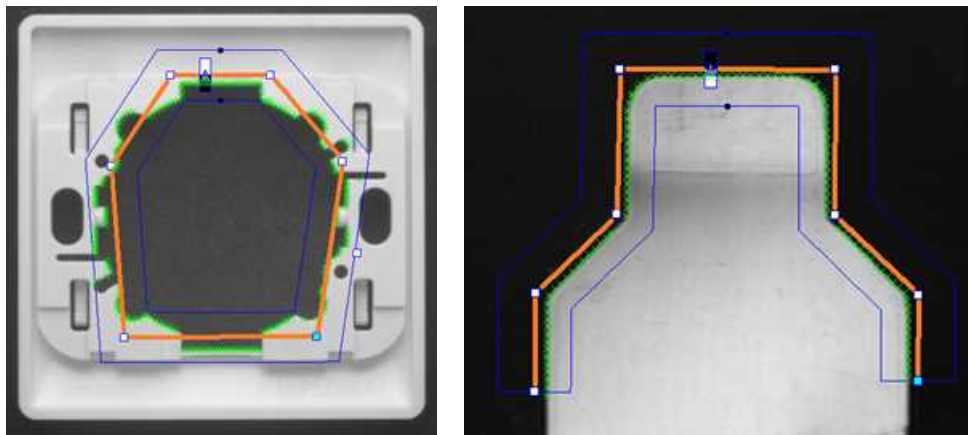
Naming conventions for the sides of a wedge gauge

Usage

- Define and position the gauge, then use [Measure](#) to fit the lines.
 - The method [EWedgeGauge.GetFound](#) returns TRUE if a suitable rectangle is found.
 - 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.

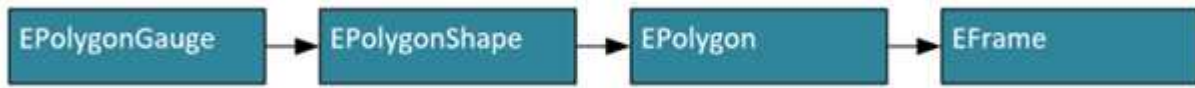
Polygon gauge

- A polygon is defined as a list of connected vertices, forming a closed or an open shape. [EPolygonGauge](#) computes several transition points along each edge of the polygon, like with a line gauge. Then from these transition points, a measured polygon is computed with a strategy depending on the measurement mode.



In orange, the closed or open polygon gauge and, in green, the measured transition points

- The position of the vertices of the polygon are defined in the local [EFrame](#) of the polygon.
 - An [EFrame](#) is a coordinate system, with a position, an orientation and a scale.
 - The class [EPolygon](#) stores the vertices and other attributes.
 - The class [EPolygonShape](#) handles the hierarchy of the shapes and the drawing methods.
 - The class [EPolygonGauge](#) is the main class for the measurement process.



Class dependency of the class [EPolygonGauge](#)

- To setup the polygon geometry, use the following methods:
 - [EPolygon.SetCenter](#) (derived from [EPoint.SetCenter](#)), [EPolygon.SetAngle](#) (derived from [EFrame.SetAngle](#)) and [EPolygon.SetScale](#) (derived from [EFrame.SetScale](#)) to change the [EFrame](#) reference coordinate system.
 - [EPolygon.SetIsClosed](#) to choose between close or open polygon configuration.
 - [EPolygon.AppendVertex](#) to add a new vertex to the polygon.
 - [EPolygon.SetVertex](#) to change the position (x, y) of a vertex.
 - [EPolygon.InsertVertex](#) to insert a vertex before the given index.
 - [EPolygon.RemoveVertex](#) to remove a vertex at the given index.
 - [EPolygonShape.AddVertexAtDisplayPosition](#) to insert a vertex depending on a display coordinate (typically a mouse click). This is useful for the graphical edition of a polygon.
 - [EPolygonShape.RemoveVertexAtDisplayPosition](#) to remove the closest vertex to a display coordinate.



TIP

[EPolygonGauge](#) has 3 measurement modes [EPolygonMeasurementMode](#).

The typical use cases are:

- For [Global](#): the precise positioning of parts.
- For [Edge](#): the verification of specific shapes defined by angles and lengths.
- For [Point](#): the measure of complex object contours.

- Use the method `EPolygonGauge.SetMeasurementMode` to select the measurement mode:
 - **Global**: The position, the orientation and optionally the scale of the polygon (`EFrame`) are adjusted to fit the detected transition points.

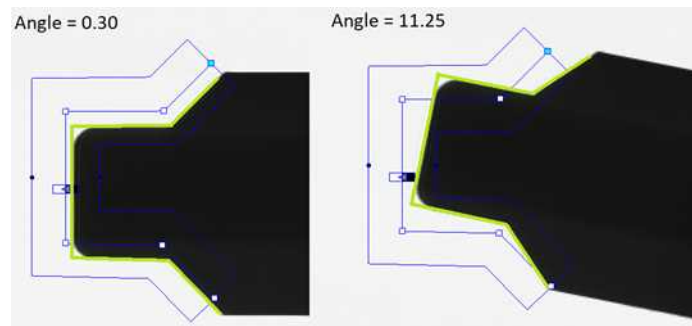
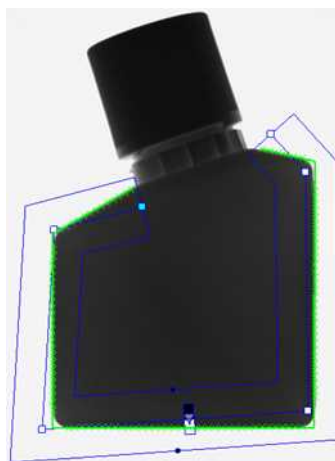


Illustration of the Global measurement mode:
the measured polygon is a translation and a rotation of the input polygon

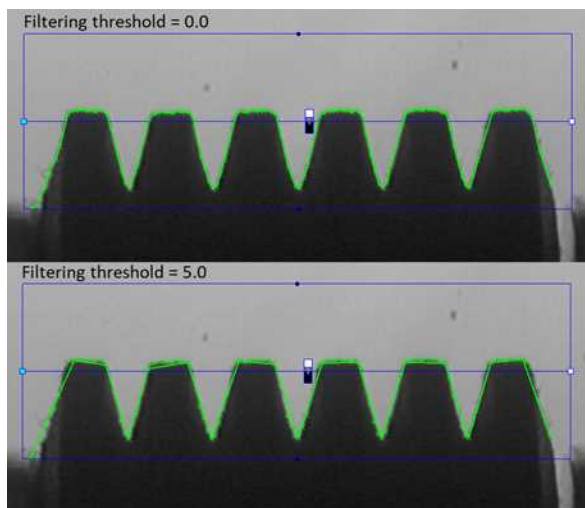
By default, the global transformation is rigid:

- Only the translation and the rotation are optimized to match the contour of the object.
- If scaling is required, you must explicitly enable it with the method `EPolygonGauge.SetEnableScaling`.
- The global measurement doesn't change the polygon vertex positions, it only adjusts the polygon frame.
- **Edge**: Each polygon edge is measured individually.
 - A polygon composed by the union of these fitted edges is generated.
 - The number of edges of the measured polygon is the same as the input polygon.



The input polygon (blue) and the fitted polygon (green)

- **Point**: All transition points on the object contour are used to build a new polygon, with an optimized number of edges.
 - The resulting polygon is simplified using a filtering threshold to reduce the number of edges while keeping the shape of the object.
 - The resulting number of edges depends on the geometry of the contour of the object and on the filtering threshold.
 - Set the filtering threshold with `EPolygonGauge.SetFilteringThreshold`.



A polygon gauge with a single edge produces a measured polygon that follow the contour of the object. Depending on the filtering threshold, the number of vertices is reduced.

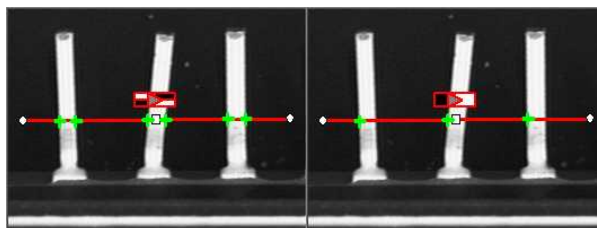
- Other parameters of the class [EPolygonGauge](#).
 The class [EPolygonGauge](#) inherits other common **EasyGauge** parameters:
 - [EPolygonGauge.SetNumFilteringPasses](#) sets the number of filtering passes, then removes outliers before the line fitting operation. This parameter is only available in Global et Edge measurement modes.
 - [EPolygonGauge.SetMinNumFitSamples](#) sets the minimum number of samples required for the fitting on each edge of the polygon. The measure fails if the minimum is not achieved.
 - [EPolygonGauge.SetTransitionType](#) sets the transition type ([ETransitionType](#)): white to black, black to white or both.
 - [EPolygonGauge.SetTransitionChoice](#) selects which peak is used when there are several transitions.
 - [EPolygonGauge.SetSamplingStep](#) gives the distance, in pixels, between two sample points.

Usage

- Sets the parameters of the gauge, then use [EPolygonGauge.Measure](#) to fit the polygon.
 - The method [EPolygonGauge.GetFound](#) returns TRUE if a suitable polygon is found.
 - To obtain the polygon geometry, use the method [EPolygonGauge.GetMeasuredPolygon](#).

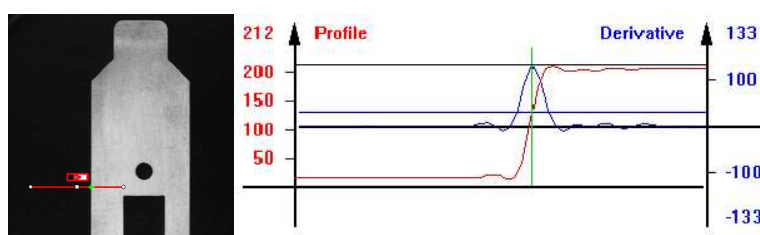
2.3. Find Transition Points Using Peak Analysis

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.

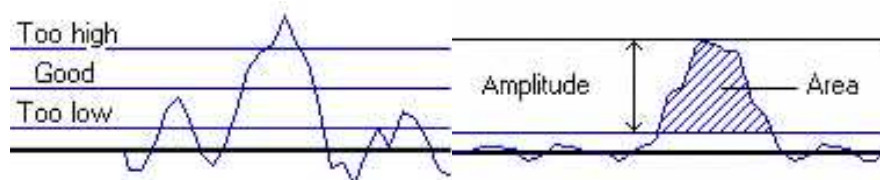
GetMeasuredPeak: Returns **EPeak** containing the peak's **Area** and **Amplitude**, and the delimiting coordinates along the probe segment (**Start**, **Length** and **Center** values).

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.

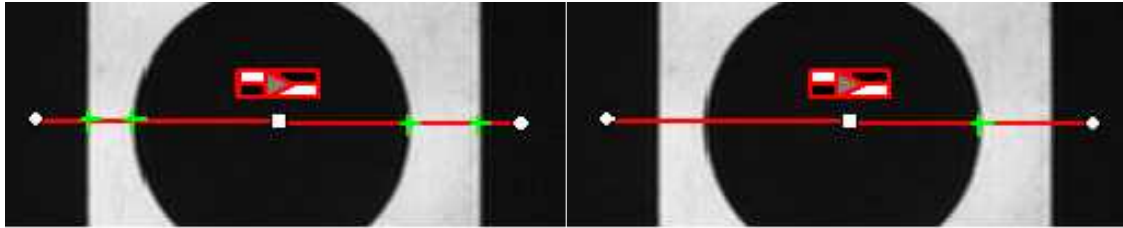
To resolve this dilemma, the EasyGauge peak selection mechanism can reject low contrast or false edges: transition strength is measured by peak *amplitude* and *area*. Every edge measurement determines peak amplitude and area. If either value falls below the **minimum amplitude** or **minimum area**,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



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

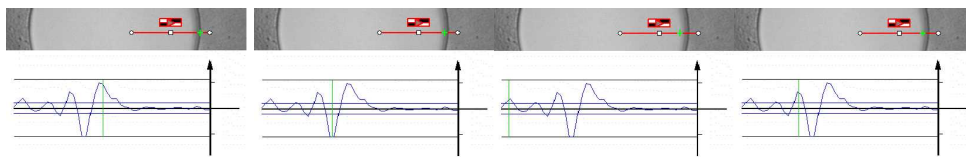
Multiple versus single transition

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.

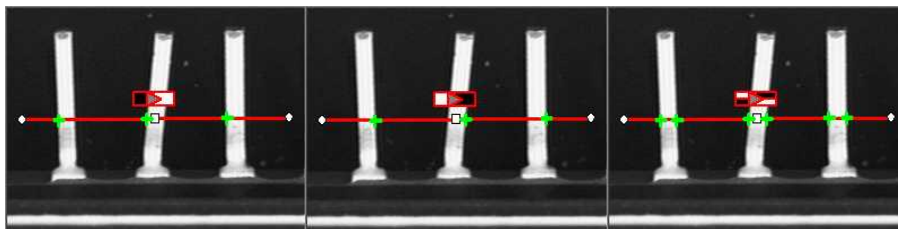


Choice: (1) best area, (2) best amplitude, (3) closest, (4) 3rd from the start

NOTE: When several peaks have the same value (same area, same amplitude...), the last peak is selected.

Positive or negative peak selection

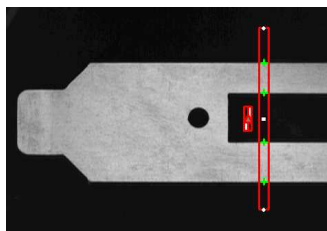
Peak selection can also be refined by choosing the transition polarity: White to Black or Black to White (i.e. positive or negative peak), or indifferent.



Black to white, white to black or indifferent polarities

Prefiltering

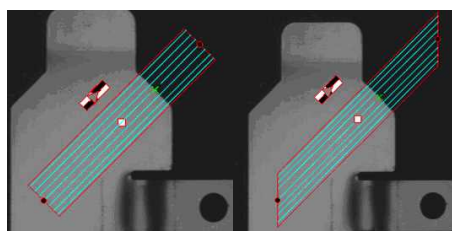
Prefiltering the image locally can reduce noise effects. Transverse (lengthwise) filtering averages several parallel lines when sampling the image. Longitudinal (crosswise) uniform filtering can also be applied to the resulting profile curve.



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.

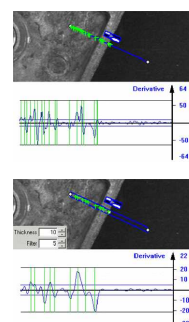
Tuning Point Measurement Parameters for unclear edges

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.

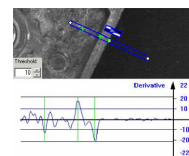
The center position and orientation are easy to decide based on a sample image or on coordinate considerations. The tolerance depends on the edge position variations. A larger tolerance increases the likelihood of hitting an edge, but it may be a false edge or extraneous feature.

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.

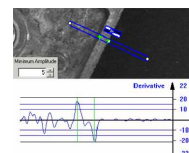


When these coefficients are set, the gray-level profile will not change anymore.

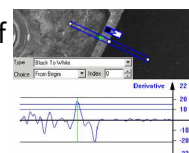
3. **Set the threshold value** to be low enough for useful parts of the peaks to cover enough pixels (to achieve better sub-pixel accuracy), but not lower than the ambient image noise.



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.



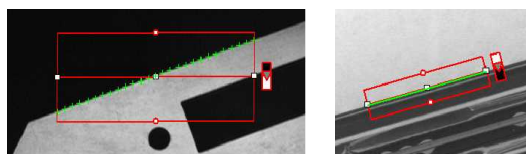
5. **Choose whether all transition points are needed or just the most relevant.** If all are required, they can be queried one after another. Otherwise, a point selection strategy should be chosen based on strength, order or transition polarity (black to white and/or conversely).



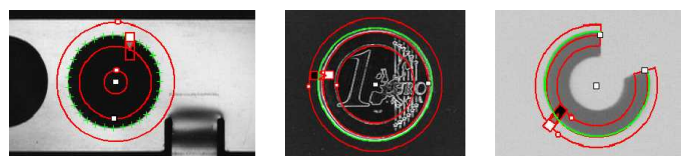
2.4. Find Shapes Using Geometric Models

The predefined geometric models [ELineGauge](#), [ECircleGauge](#), [ERectangleGauge](#), [EWedgeGauge](#) or [EPolygonGauge](#) 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.

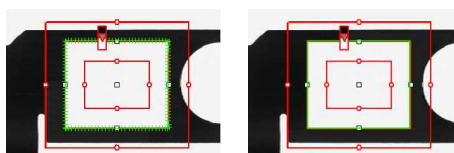
- **Line:** Measures position and orientation of straight edges.



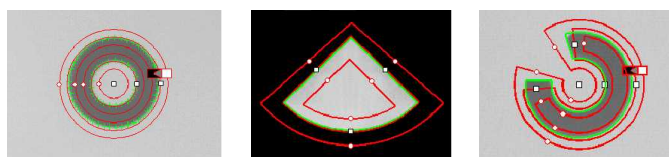
- **Circle:** Measures position and curvature of a circle or arc.



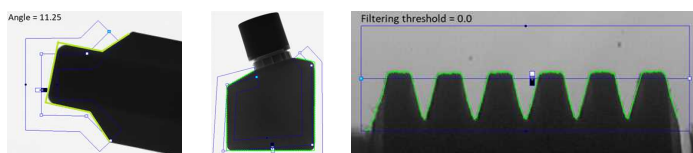
- **Rectangle:** Measures position, orientation and size of a rectangle.



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

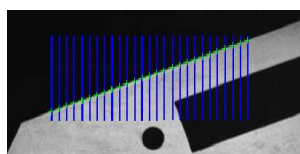


- **Polygon:** Fits a polygon to an object and measures the exact position of the polygon, the relative orientation of the edges or the path of the contour.



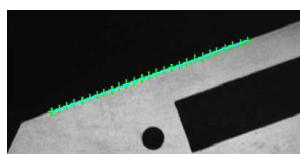
All gauge types share these common features:

- **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.

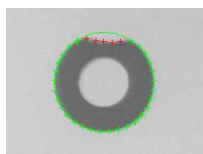


Sampling paths and sampled points

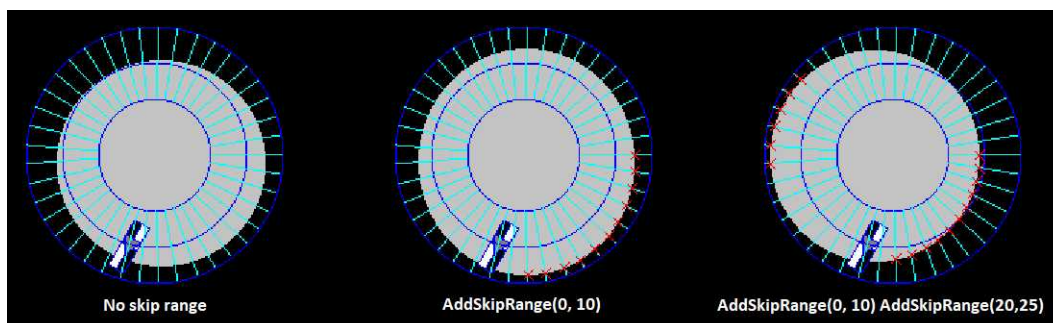
- **Model fitting**
 - The model is adjusted to minimize error residue and provide the best edge parameter estimates. Rectangles and wedges have parallelism and concentricity constraints. Image shows sampled points and fitted line.



- **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](#).



- **Skip range**
 - The skip ranges define exclusion ranges for the paths. The sample point indexes in the skip ranges are not taken into account to fit the model.
 - The skip ranges apply to all gauge models: [line](#), [circle](#), [rectangle](#), [wedge](#) and [polygon](#).
 - To define several skip ranges, use [AddSkipRange](#).
 - To manage the skip ranges, use [RemoveSkipRange](#), [RemoveAllSkipRanges](#) and [GetSkipRange](#).
 - You must call the method [Measure](#) to take a skip range into account.
 - Use the attribute [EDrawingMode_PointsInSkipRange](#) with the method [Draw](#) to display the skipped points.



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.

In addition, if you are interacting with a polygon gauge, you can:

- Use the method `EPolygonGauge.AddVertexAtDisplayPosition` (derived from `EPolygonShape.AddVertexAtDisplayPosition`) to create a new vertex at a chosen position (typically at a mouse click)
- Use the method `EPolygonGauge.RemoveVertexAtDisplayPosition` (derived from `EPolygonShape.RemoveVertexAtDisplayPosition`) to remove a vertex.

Plot

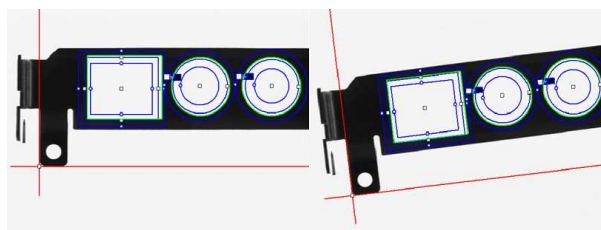
EasyGauge can Plot gray-level values along the sampled paths and/or its derivative - useful for parameter tuning.

- 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>	

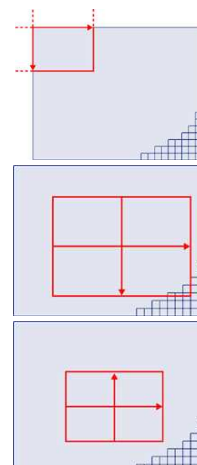
Before World-to-Sensor Transform

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

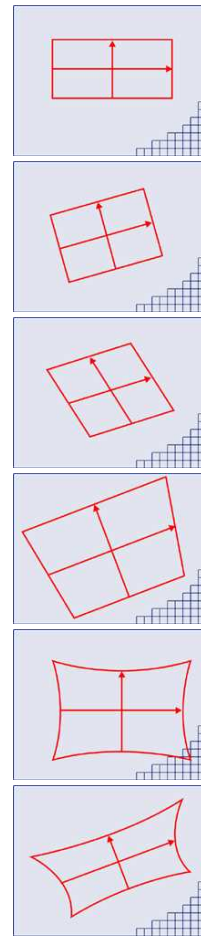
- adjust sweep frequency or scanning speed to avoid non-square pixels.
- adjust optical alignment to minimize perspective effect. The field of view should be parallel to the sensor plane.
- use long focal distances and good quality lenses to minimize Optical distortion.
- use appropriate scale factor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

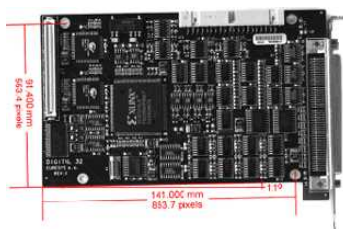
Estimate (feasible if no distortion correction is required and accuracy requirements are low)

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.

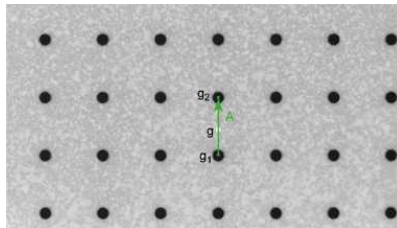
Analyze a Calibration target

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.

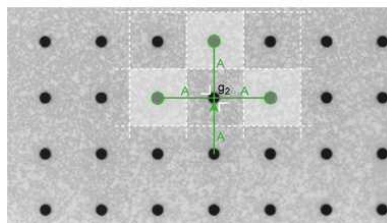
Open eVision relies on the use of a specific target holding a rectangular grid of symmetrical dots (of any shape) with no other object on the grid.

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 .



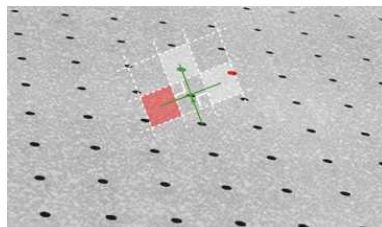
- b. Starting from the extremity of the reference segment (g_2), the algorithm determines 3 tolerance areas (white squares in the figure), in perpendicular directions. The tolerance areas are centered at a distance A (length of the reference segment) from (g_2). They are square, with a side-length of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

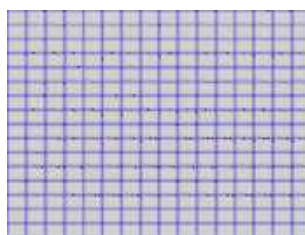
$$\begin{aligned} f\text{-}o\text{-}v \text{ width} &= \text{pixel width} * \text{sensor width} \\ f\text{-}o\text{-}v \text{ height} &= \text{pixel height} * \text{sensor height} \end{aligned}$$

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



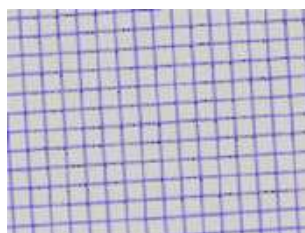
Anisotropic aspect ratio

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

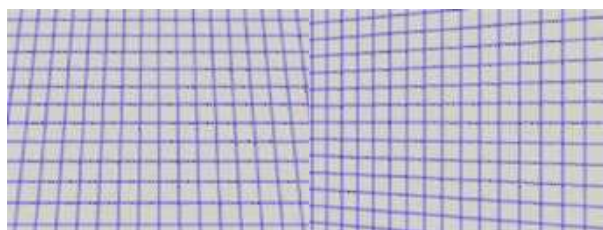


NOTE

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

X and Y tilt angles

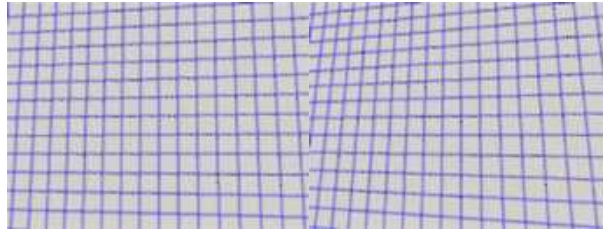
The **X** and **Y** tilt angles describe the viewing plane direction. They correspond to the required rotations around X and Y axis that bring the Z axis parallel to the optical axis.



Tilt X and tilt Y angles

Perspective strength

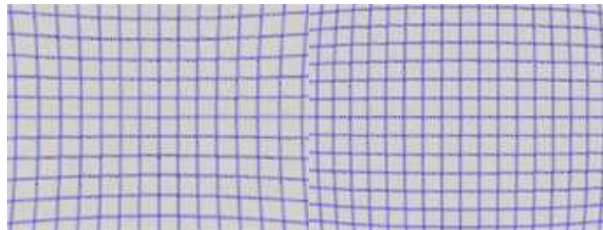
The **perspective strength** gives a relative measure of the perspective effect. The shorter the focal length, the larger the value.



Weak and strong perspective

Distortion strength

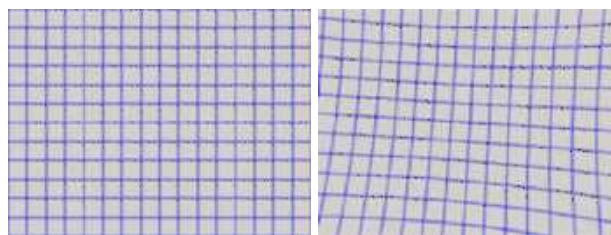
Distortion strength gives a relative measure of radial distortion in the image corners, that is the ratio of image diagonal length with and without distortion.



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

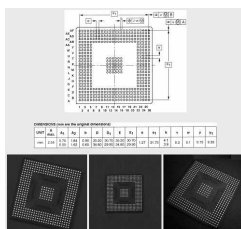
An **EWorldShape** object manages a field-of-view calibration context. Such an object is able to represent the relationship between world coordinates (physical units) and sensor coordinates (pixels), and account for the distortions inherent in the image formation process.

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.

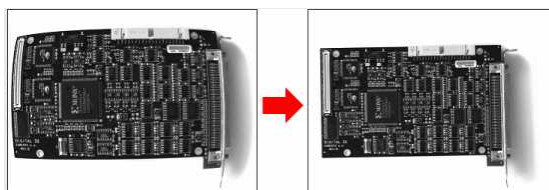
The goal of calibration is twofold:

- 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.



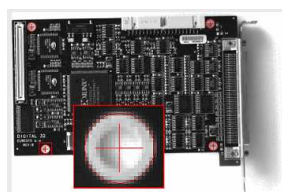
Single model versus multiple viewing conditions

- 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.

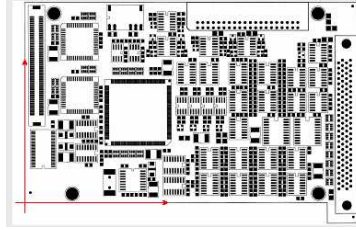


Feature point in sensor space

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.

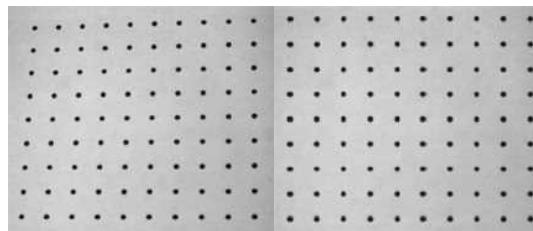
They require a reference frame to be defined.



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. Introduction

3.2. Purpose and Principles

Purpose

Based on an innovative feature-point technology, **EasyFind** is a matching tool designed to rapidly locate one or more instances of a reference model in an image. With an adjustable accuracy up to sub-pixel level, it reports very precise information about the instances found, such as their location, rotation angle, scale and matching score.

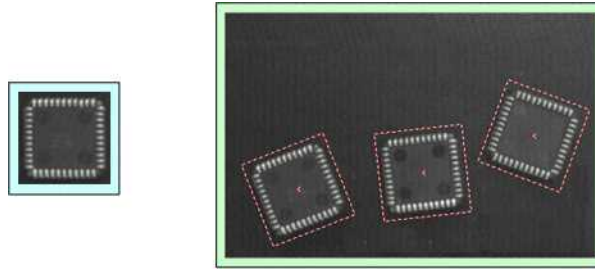
- **EasyFind** supports “don't-care” areas, a feature that allows the creation of complex pattern shapes.
- Fast Processing and Improved Robustness:

EasyFind is based on a new feature-point technology. Instead of comparing the reference model to the sample image pixel-wise, it carefully selects relevant feature points in the model. This method allows **EasyFind** to match only the areas that convey valuable information, resulting in faster processing and much improved robustness.

- Training on vector patterns:

In this mode, the learning is done on collections of 2D geometrical shapes rather than on rasterized patterns. The learning model is constructed using the new class `EVectorModel` either by loading it from a DXF file or, programmatically, by using **Open eVision** `EShape` objects.

This extension is well- suited to find objects with a known geometry.



Example of a model (left) and 3 found instances (right)



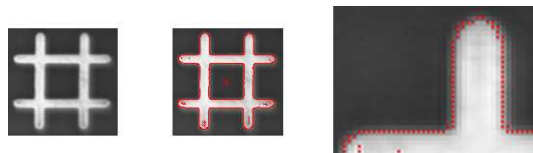
TIP

Compared to **EasyMatch**, **EasyFind** is computationally fast, robust against noise, occlusion, blurring, missing parts and illumination variations.

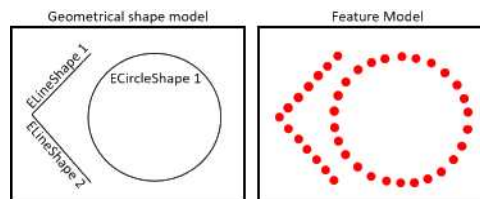
Edge feature points

- **EasyFind** uses edge feature points to find instances in a search field:
 - An **edge feature point** is an abrupt change of gray level between two regions.
 - It indicates that there is an edge at this location in the search field.
- To start the finding function, **EasyFind** needs either a model image on which it computes the edge feature points or directly a geometrically-defined feature model.

NOTE: The optimal model depends on the type of pattern that you are searching.



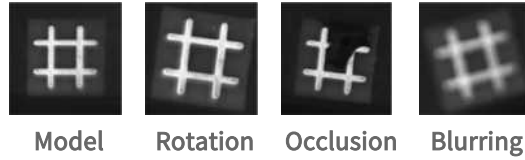
An image model and the computed edge feature points



A vector model and the computed edge feature points

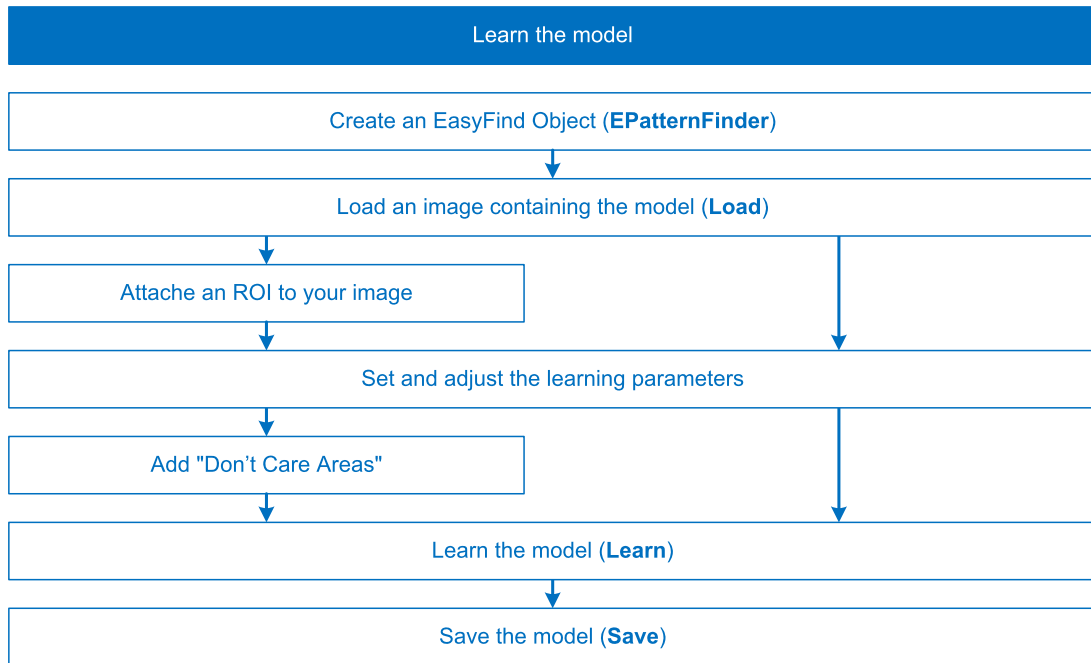
- The point-by-point scores improve the robustness and the computation time of the finding phase.
- To use this tool:
 - The models must be well contrasted with sharp edges.
 - They should be substantially different from the rest of the expected search fields.
 - They can be scaled or rotated.

- **EasyFind** is very robust regarding:
 - Blurring
 - Noise
 - Occlusion
 - Illumination variation

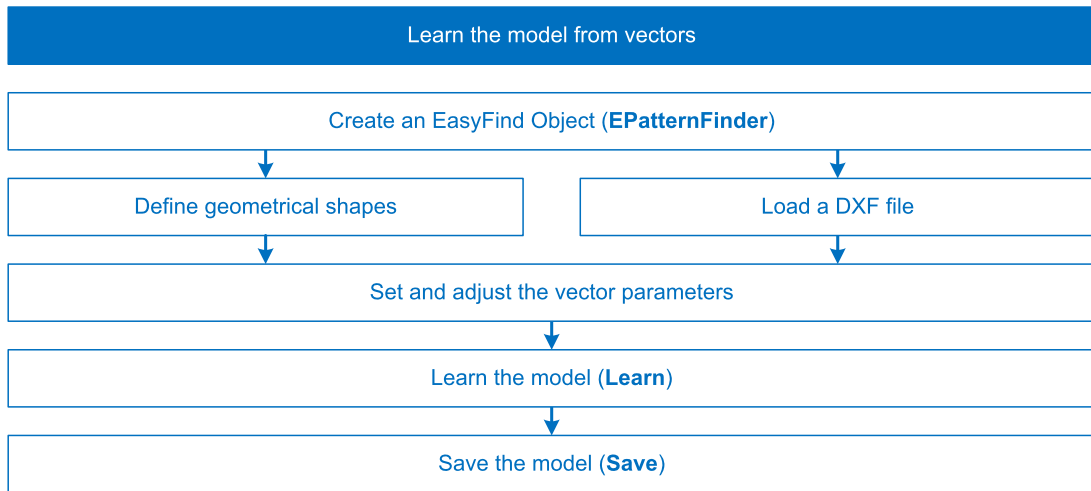


3.3. Workflow

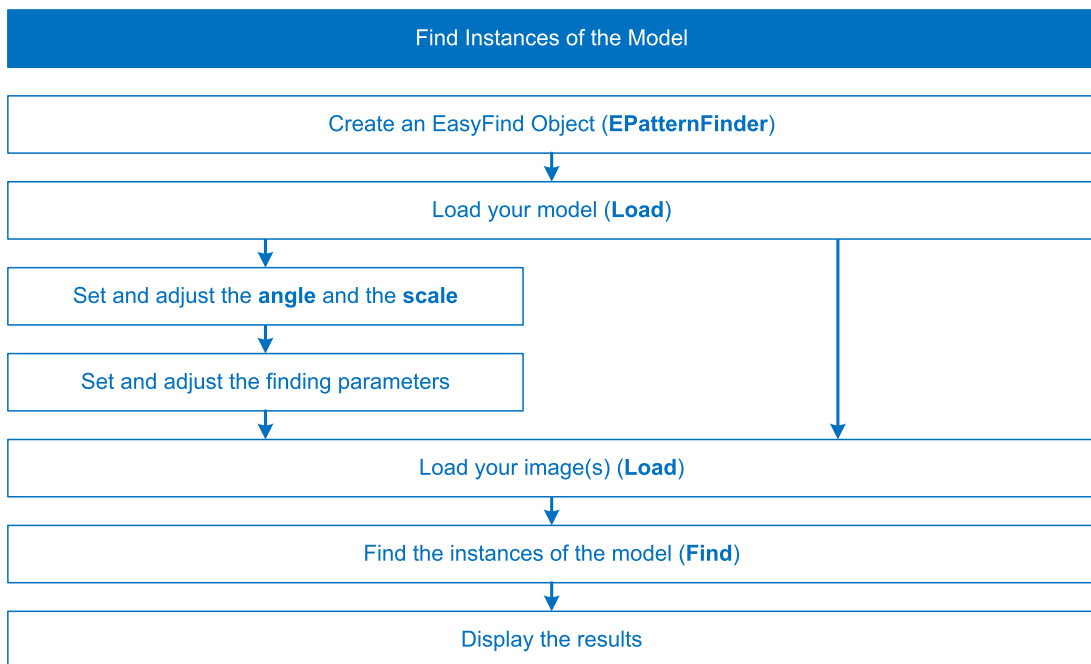
Learn the model from images



Learn the model from vectors



Find instances of the model



3.4. Using EasyFind

3.5. Learn the Model from Images

Learning

EasyFind detects in images the instances of a reference model.

- The reference model is a set of all the feature points that **EasyFind** computes during the learning process:
 - **EasyFind** can extract these points from a bitmap representation of the pattern as described below.
 - Or it can use the feature points that are directly given by geometrical shapes (see "[Learn the Model from Vectors](#)" on page 172).

Process

To learn a model in an image with **EasyFind**:

1. Create an EPatternFinder instance for the model and an EFoundPattern vector for the results.

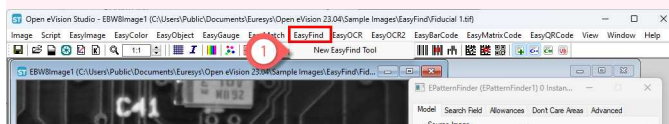
Code

```
EPatternFinder finder;  
vector<EFoundPattern> foundPattern;
```

Studio

In the main menu:

1. **EasyFind** > **New EasyFind Tool** > name your tool.



2. Load the image with your model.

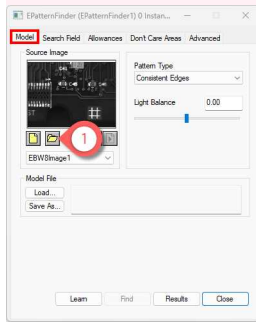
Code

```
EImageBW8 image;  
image.Load("Fiducial 1.tif");
```

Studio

In the tool window, in the **Model** tab:

1. Open your **Source Image**.
2. Select your image in the browse dialog (Fiducial 1.tif).



3. Attach an ROI to your image.

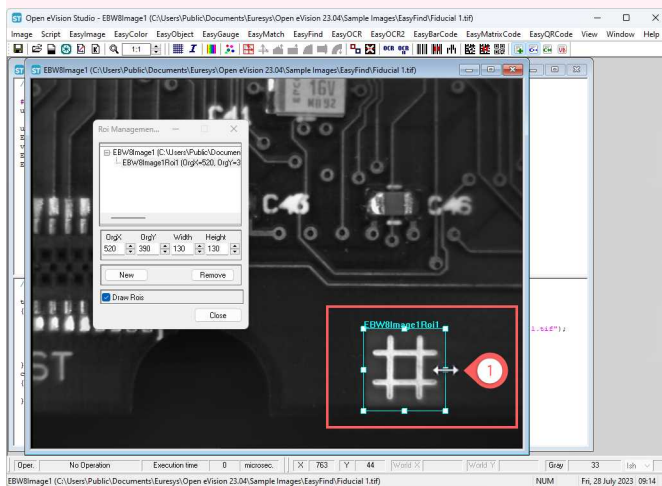
Code

```
EROIBW8 pattern;  
pattern.Attach(&image, 520, 390, 130, 120);
```

Studio

In the image window:

1. Right-click > **New ROI** > move and resize in the image > **Close**.



4. According to your application and needs, adjust the **"Learning Parameters"** on page 180.
5. To adjust the model shape and to improve your find results, you can add **"Don't Care Areas"**.

See **"Use "Don't Care Areas" in the Model"** on page 178.

6. Select the source image and learn the model.

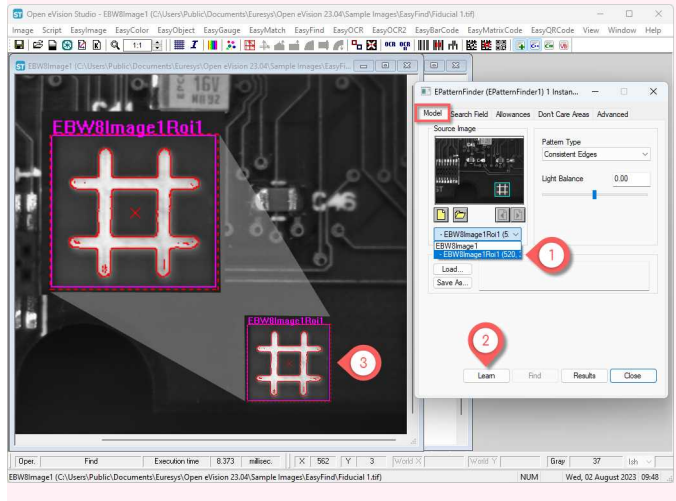
Code

```
finder.Learn(&pattern);
```

Studio

In the tool window, in the **Model** tab:

1. Select the ROI as your **Source Image**.
2. **Learn**
3. The result is displayed on the image.



7. If you want to reuse it later, save the new model (. fnd file).

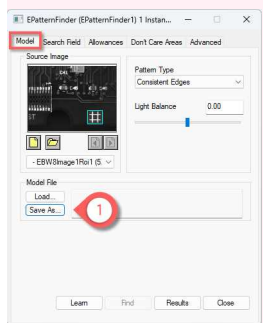
Code

```
finder.Save("myModel.fnd");
```

Studio

In the tool window, in the **Model** tab:

1. In the **Model File** area > **Save As...**



► **EasyFind** creates and save your model.

Use this model to perform:

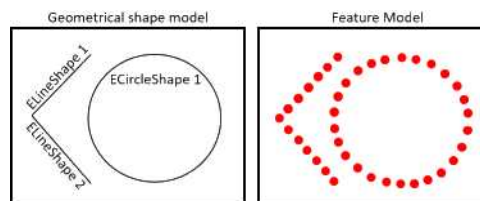
- "Find Instances of the Model" on page 174

3.6. Learn the Model from Vectors

Learning

EasyFind detects in images the instances of a reference model.

- The reference model is a set of all the feature points that **EasyFind** computes during the learning process:
 - **EasyFind** can extract these points from a bitmap representation of the pattern (see "[Learn the Model from Images](#)" on page 169).
 - Or it can use the feature points that are directly given by geometrical shapes as described below.
- To learn a model from geometrical shapes:
 - Use the class `EVectorModel` to hold a collection of shapes as a vector model.
 - Load the `EVectorModel` from an external DXF file or use the **Open eVision** API to define it.



Process using an EPolygonShape



NOTE

- This feature is not available in **Open eVision Studio**.
- You can also use the sample program [EasyFindVectorLearn](#).

To learn a model from geometrical shapes with **EasyFind**:

1. Create an `EPatternFinder` instance for the model.

Code

```
// EPatternFinder constructor
EPatternFinder finder;
```

2. Create your vector model and get its root shape.

Code

```
// EVectorModel constructor
EVectorModel myModel;
// Get the root EFrameShape of the model
EFrameShape& shapeMother = myModel.GetRoot();
```


3. Create your polygon.

Code

```
// EPolygonShape constructor
EPolygonShape polygon;
// Define the vertices of a polygon
std::vector<EPoint> vertices = { {0.f, 0.f}, {1.f, 0.f}, {1.f, 1.f}, {0.f, 1.f} };
// Define the EPolygonShape
polygon.SetPolygon(EPolygon(vertices, true));
```

4. Attach the polygon to the root shape and adjust its parameters according to your application and needs.

See ["Vector Model Parameters" on page 193.](#)

Code

```
// Attach the EPolygonShape to the root EFrameShape
polygon.Attach(&shapeMother);
// Sets the polarity of the EPolygonShape
polygon.SetProperty("polarity", "direct");
```

5. Learn the model.

Code

```
finder.Learn(&pattern);
```

6. If you want to reuse it later, save the new model (.fnd file).

Code

```
finder.Save("myModel.fnd");
```

► **EasyFind** creates and save your model.

Use this model to perform:

- ["Find Instances of the Model" on page 174](#)

Process using a DXF file



NOTE

This feature is not available in **Open eVision Studio**.

- Alternatively to define the geometrical shapes of your model, you can load an [EVectorModel](#) from an external DXF file.

Once loaded, the [EVectorModel](#):

- Holds a collection of shapes.
- Holds a center position depending on the method used for the definition of the DXF.

1. Create an EPatternFinder instance for the model.

Code

```
// EPatternFinder constructor
EPatternFinder finder;
```

2. Create your vector model.

Code

```
// EVectorModel constructor  
EVectorModel myModel;
```

3. Load the model from the DXF file.

Code

```
// Load the model from a dxf file  
myModel.LoadDXF("myModel.dxf");
```

4. The scale and polarity attributes are not part of the DXF file. If necessary, add them to the `EVectorModel`.

See "Vector Model Parameters" on page 193.

5. Learn the model.

Code

```
finder.Learn(&pattern);
```

6. If you want to reuse it later, save the new model (.fnd file).

Code

```
finder.Save("myModel.fnd");
```

► **EasyFind** creates and save your model.

Use this model to perform:

- "Find Instances of the Model" on page 174

3.7. Find Instances of the Model

Finding

EasyFind detects the instances of your reference model in your images.

These instances can be highly-degraded due to noise, blur, occlusion, missing parts or unstable illumination conditions. To optimize the finding process, adjust the search parameters and check what **EasyFind** has found in the result information.



TIP

To speed up the search process, you can also use [multicore processing](#). This is especially interesting when you use **EasyFind** with angle and scale tolerances.

Process

Create your model as described in "Learn the Model from Images" on page 169 or "Learn the Model from Vectors" on page 172.

- In **Open eVision Studio**, if you just created a model following "Learn the Model from Images" on page 169, go directly to **step 3**.

1. Create an EPatternFinder instance for the model and an EFoundPattern vector for the results.

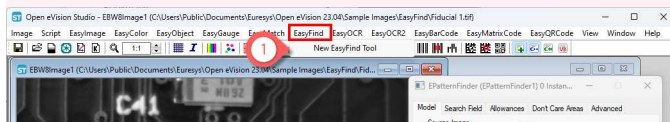
Code

```
EPatternFinder finder;  
std::vector<EFoundPattern> foundPattern;
```

Studio

In the main menu:

1. **EasyFind** > **New EasyFind Tool** > name your tool.



2. Open an existing model saved in a .fnd file:

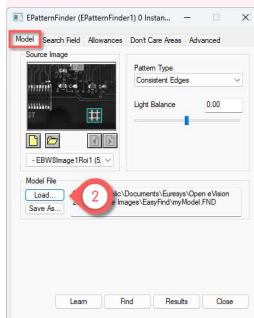
Code

```
finder.Load("myModel.fnd");
```

Studio

In the tool window, in the **Model** tab:

1. In the **Model File** area > **Load...**



3. According to your application and needs, adjust the following parameters:
 - **SetAngleBias** and **SetAngleTolerance**: search for instances in this angle range.
 - **SetScaleBias** and **SetScaleTolerance**: search for instances in this scale range.

see "Finding Parameters" on page 185 for more details.

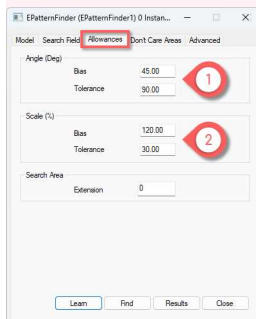
Code

```
finder.SetAngleBias(45.00f);
finder.SetAngleTolerance(90.00f);
finder.SetScaleBias(1.20f);
finder.SetScaleTolerance(0.30f);
```

Studio

In the **Allowances** tab:

1. In the **Angle (Deg)** area, set the **Bias** and the **Tolerance** on the angle.
2. In the **Scale (%)** area, set the **Bias** and the **Tolerance** on the scale.



4. Load your image(s).

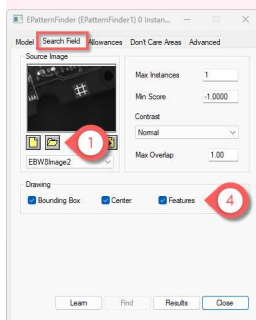
Code

```
EImageBW8 srcImage;
srcImage.Load("...\Sample Images\EasyFind\Fiducial 2 (Skewed).tif");
```

Studio

In the tool window, in the **Search Field** tab:

1. Open your **Source Image**.
2. Select one or several images in the browse dialog (Fiducial 2->8 xxx.tif).
3. Name your image set.
4. In the **Drawing** area, check all settings to display them on the found instances.



5. Find the instances in your image set.

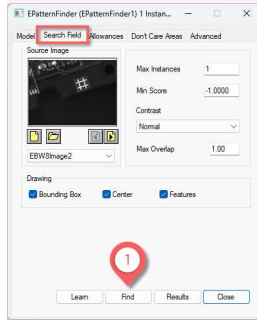
Code

```
foundPattern = finder.Find(&srcImage);
```

Studio

In the Search Field tab:

1. Find



6. Display the results.

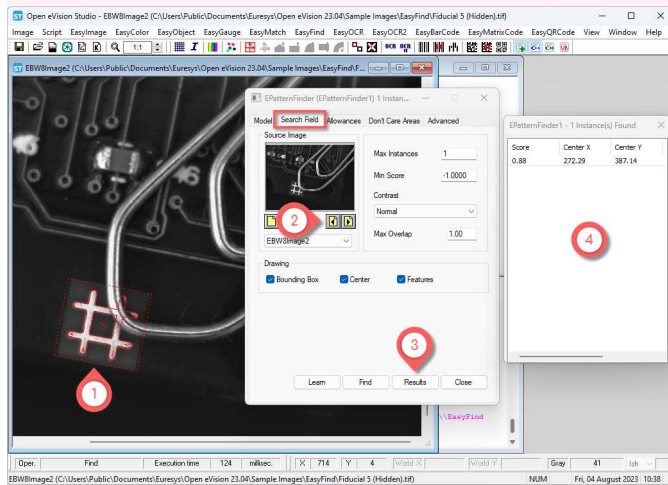
Code

```
float score= foundPattern[0].Score;
float centerX= foundPattern[0].Center.X;
float centerY= foundPattern[0].Center.Y;
```

Studio

In the tool window, in the Search Field tab:

1. The instance location and feature points are highlighted in the current source image.
2. Use the Load Previous File and Load Next File to browse the images of your set.
3. Click on Results to open the result windows.
4. The result windows displays the score and the center coordinates of the found instance(s).



3.8. Open eVision Studio Tools

3.9. Use "Don't Care Areas" in the Model




NOTE

The "don't care areas" are deprecated:

- Instead, use an ERegion (see "[Arbitrarily Shaped ROI \(ERegion\)](#)" on page 35).
- As ERegions are not supported in **Open eVision Studio**, you can still use "don't care areas" as described below to improve the finding results.

Don't care areas

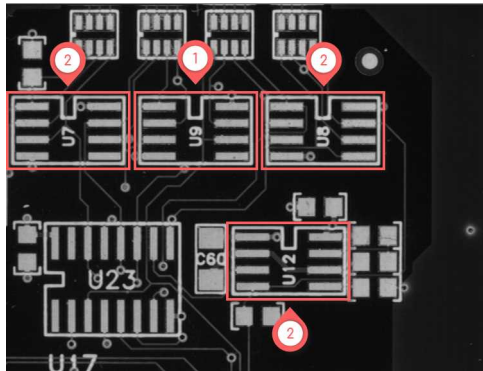
- Use "don't care areas" in geometric pattern matching to define in the image the meaningful features only.
 - Create a "don't care areas" mask image to remove from the search process the areas that may change from image to image, such as text and numbers:
 - "0" values indicate ignored areas.
 - "255" values indicate areas taken into account.
-  For more details about the mask, see "[Flexible Masks](#)" on page 57.

Process

Create your model as described in "Learn the Model from Images" on page 169.

This example is based on the file Solder Pad 1.tif with:

- An ROI positioned at (200, 130, 190, 130)
- Max Instances = 4



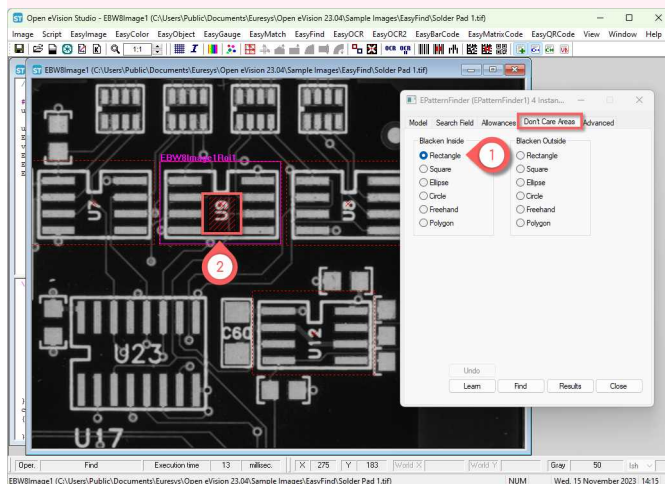
(1) Model - (2) Instances

1. Add a Don't Care Area to improve your model.

Studio

In the tool window, in the Don't Care Areas tab:

1. Blacken Inside > Rectangle.
2. Draw a rectangle over the central number to remove this part from the model.

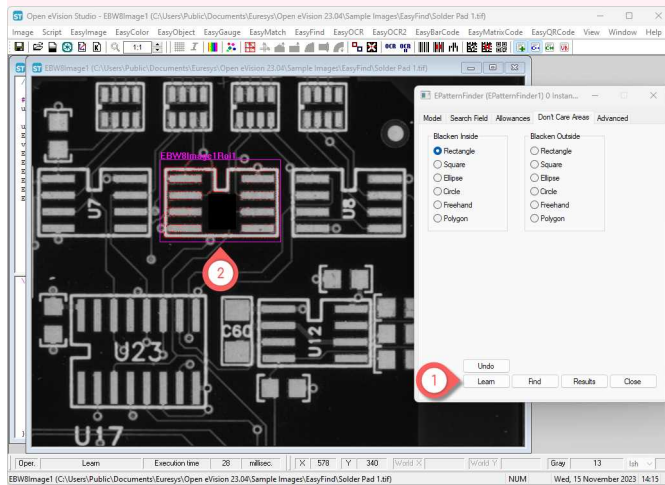


2. Learn the model.

Studio

In the tool window, in any tab:

1. Learn
2. The result is displayed on the image.



► In the example, the find results are better with the don't care area than without as follows:

Score	Center X	Center Y	Angle (Deg)	Scale(%)
1.00	295.00	195.00	0.00	100.00
0.95	493.51	195.28	0.00	100.00
0.94	96.84	194.84	0.00	100.00
0.92	440.32	397.89	0.00	100.00

Score	Center X	Center Y	Angle (Deg)	Scale(%)
1.00	295.00	195.00	0.00	100.00
0.96	96.89	194.83	0.00	100.00
0.96	493.53	195.30	0.00	100.00
0.95	440.25	397.89	0.00	100.00

Results without (left) and with (right) don't care areas

► EasyFind creates your model.

Use this model to perform:

- "Find Instances of the Model" on page 174

3.10. Setting the Parameters

3.11. Learning Parameters

📖 The following parameters are set and used in the process: "Learn the Model from Images" on page 169.



TIP

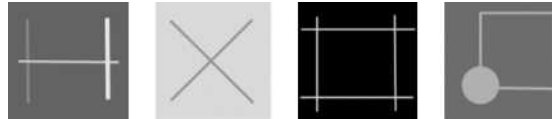
These parameter settings are saved together with your model in the model file.

Pattern type

- Use the parameter `SetPatternType` to set the pattern type as consistent edges (default) or thin structures.

EasyFind supports 2 pattern types:

- Use Consistent edges when:
 - Your model is well contrasted with sharp edges.
 - Your model is substantially different from the rest of the expected search fields.
- Use Thin structures when:
 - The edges of your model are consistent between thin elements and regions.
 - The contrast is the same for each thin element.



Examples of thin structures

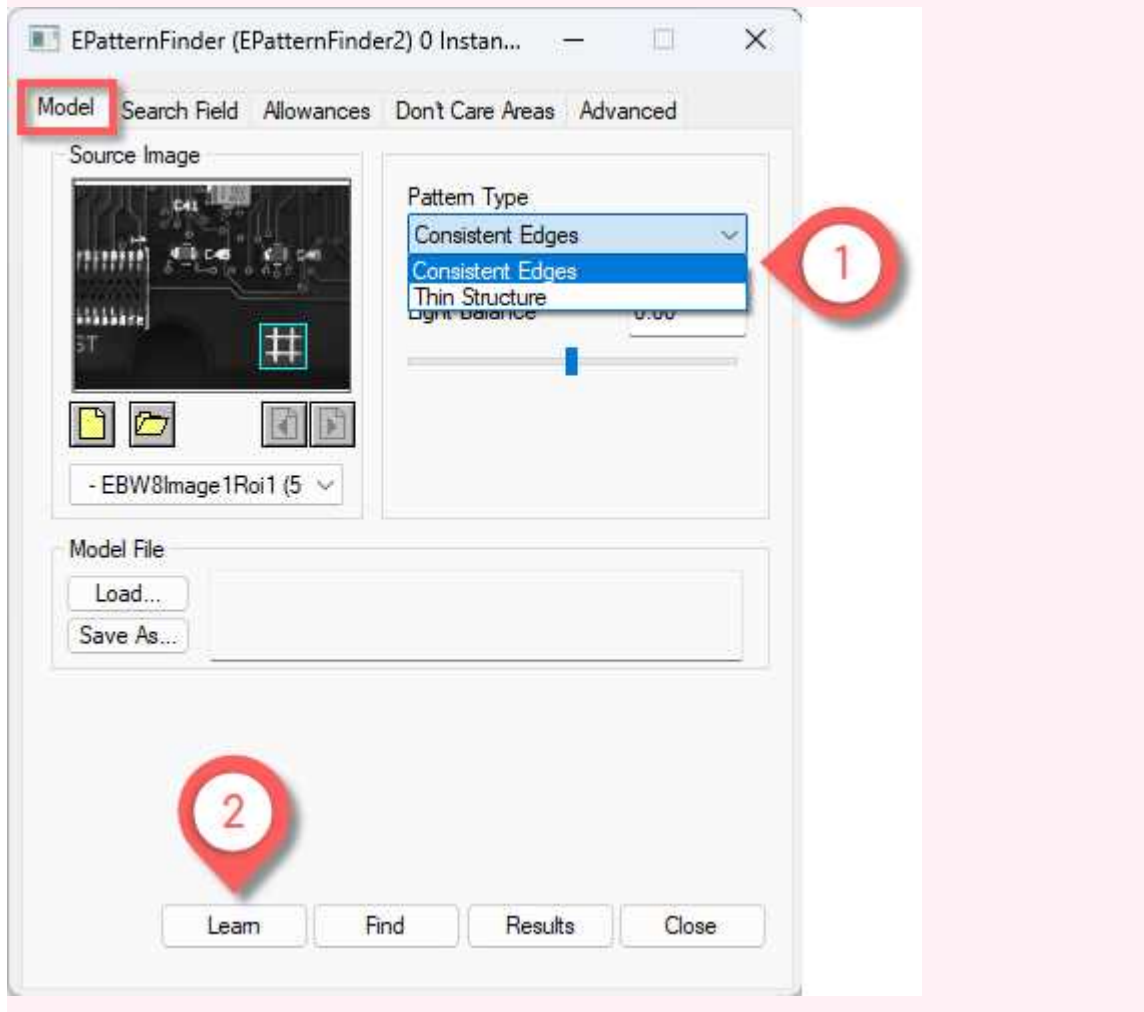
Code

```
finder.SetPatternType(PatternType_ConsistentEdges);
```

Studio

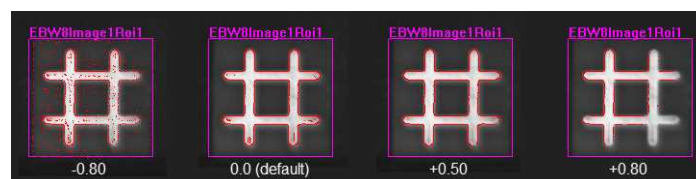
In the Model tab:

1. Select the Pattern Type.
2. Learn



Light Balance

- Use the parameter `SetLightBalance` to adjust the gray-level threshold of the model so that it fits the useful parts of the pattern.
 - In **Open eVision Studio**, the preview of the model is automatically updated when you adjust the light balance.
 - Once you have the correct light balance, learn your model again.
- ▶ Example of the computed feature points for different light balance values:



Code

```
finder.SetLightBalance(0.50f);
```

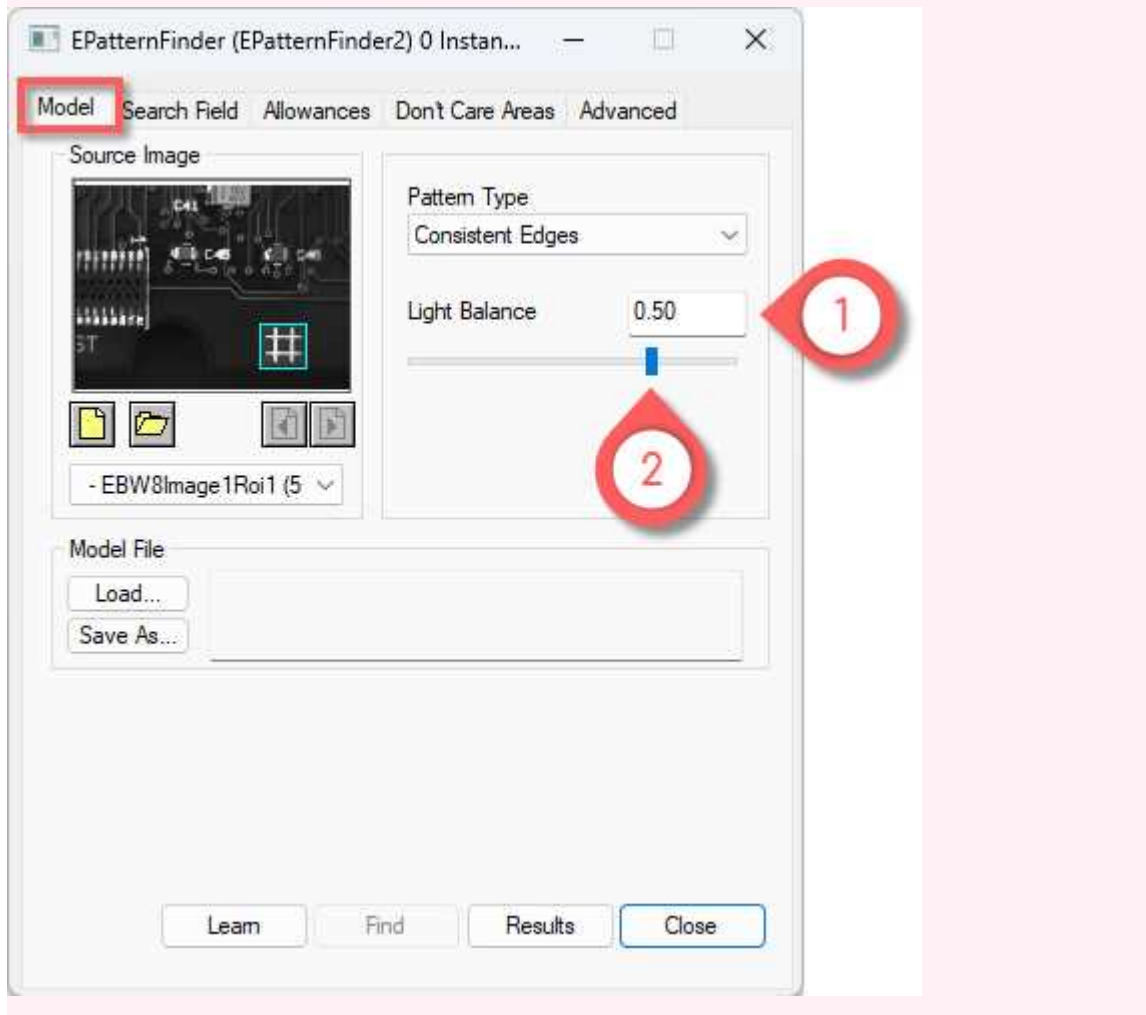
Studio

In the **Model** tab:

1. Set the **Light Balance** and click **Learn**.

Or

2. Move the **Light Balance** slider. The result is immediately displayed on your model.



Thin structure mode (advanced)

If the Pattern Type is Thin Structure:

- Use the parameter `SetThinStructureMode` to adjust the gray-level threshold of the model so that it fits the useful parts of the pattern:
 - **Auto:** **EasyFind** chooses automatically the best contrast for thin elements.
 - **Dark:** **EasyFind** selects thin elements darker than their neighborhood.
 - **Bright:** **EasyFind** selects thin elements brighter than their neighborhood.

Code

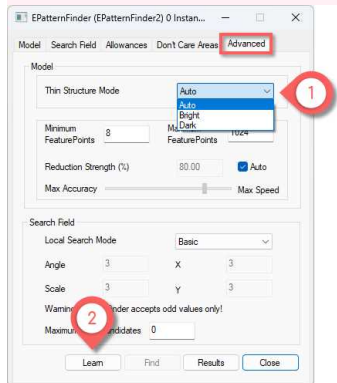
```
finder.SetThinStructureMode(ThinStructureMode_Auto);
```

Studio

If the **Pattern Type** is **Thin Structure** in the **Model** tab:

In the **Advanced** tab:

1. Select the **Thin Structure Mode**.
2. **Learn**



Number of feature points and reduction strength (advanced)



More feature points lead to a finer matching and a better positioning accuracy but also to a longer processing time.

- Use the parameters **SetMinFeaturePoints** and **SetMaxFeaturePoints** to adjust the number of feature points used to match the patterns.
 - By default, this number is computed automatically (between 8 and 1024).
- Use the parameter **SetReductionMode** to select the reduction mode:
 - **Auto** (default): the reduction strength is computed automatically.
 - **Manual**: use the parameter **SetReductionStrength** to set the reduction strength from more accuracy (0.0) to more speed (1.0).

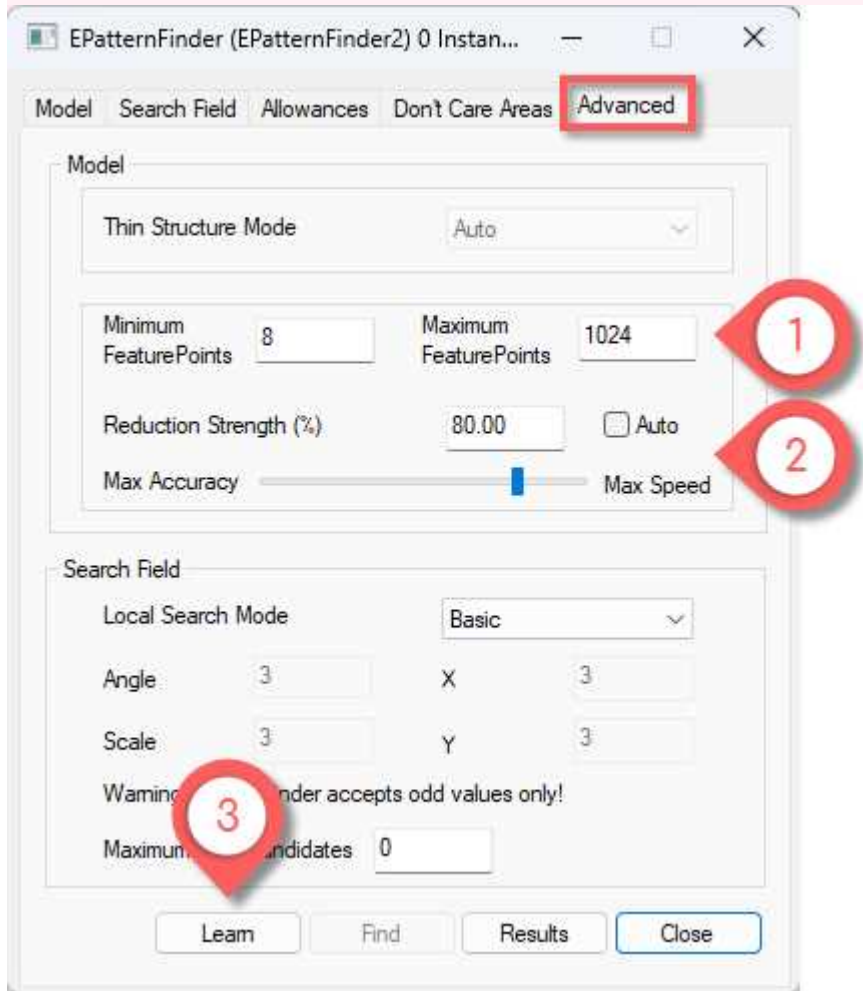
Code

```
finder.SetMinFeaturePoints(8);
finder.SetMaxFeaturePoints(1024);
finder.SetReductionMode(EReductionMode_Manual);
finder.SetReductionStrength(0.80f);
```

Studio

In the **Advanced** tab:

1. Set the **Minimum Feature Points** and the **Maximum Feature Points**.
2. **Reduction Strength**:
 - Check **Auto** for automatic computation.
 - Or
 - Uncheck **Auto**.
 - Set the **Reduction Strength** or move the slider between **Max Accuracy** and **Max Speed**.
3. **Learn**

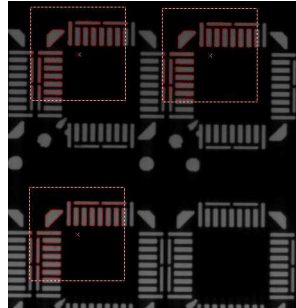


3.12. Finding Parameters

The following parameters are set and used in the process: **"Find Instances of the Model"** on page 174.

Maximum number of expected instances

- Use the parameter `SetMaxInstances` to set the maximum number of instances that **EasyFind** returns.
- ▶ In this example the number was 3:



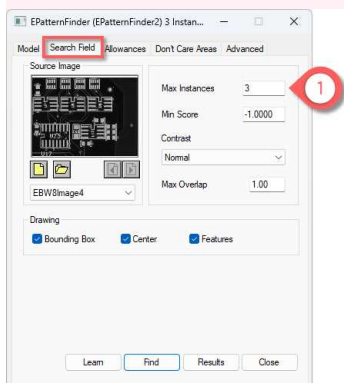
Code

```
finder.SetMaxInstances(3);
```

Studio

In the **Search Field** tab:

1. Set the **Max Instances**.



Minimum score and contrast

If the Pattern Type is Thin Structure:



The score depends on the Contrast:

- Normal: the score is normalized between -1 and 1:
 - 1 means a perfect match with the same contrast as the model.
 - 0 means no match.
 - -1 means a perfect match but with the inverted contrast compared to the model.
- Inverse: the score is normalized between -1 and 1:
 - 1 means a perfect match with the inverted contrast compared to the model.
 - 0 means no match.
 - -1 means a perfect match but with the same contrast as the model.
- Any: the score is normalized between 0 and 1:
 - 1 means a perfect match with the same contrast or the inverted contrast compared to the model.
 - 0 means no match.



Global vs point by point contrast:

- Global: the score is computed as the normalized sum of the correlation of each point.
 - ▶ The feature points with a high gradient have more weight.
 - ▶ Global gives higher score on incomplete instances.
- Point by point: the score is computed as the mean of the normalized correlation of each point.
 - ▶ Each feature point has the same weight.
 - ▶ Point by point gives higher score on images with irregular lighting and instances with variable contrast.
- Use the parameter [SetMinScore](#) to set the minimum score for an instance to be accepted and returned.
 - The default value is -1.0 (no score filtering).
- If the Pattern Type is Consistent Edges, use the parameter [SetContrastMode](#) to select the contrast mode.
 - The default value is Normal.

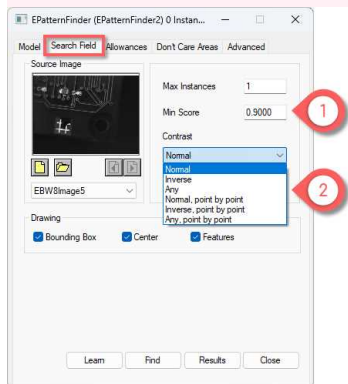
Code

```
finder.SetMinScore(0.90f);  
finder.SetContrastMode(EFindContrastMode_Normal);
```

Studio

In the **Search Field** tab:

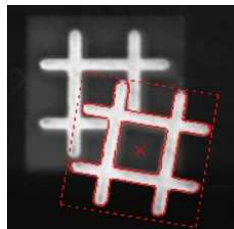
1. Set the **Min Score**.
2. Select the **Contrast**.



Maximum overlap

EasyFind can filter out instances that completely or partially overlap with each other.

- Use the parameter **SetMaxOverlap** to defined the allowed overlap.
 - The maximum overlap is defined as the area of both instances intersection divided by the area of the smallest instance.
 - Set the parameter between 0.0 (no overlap allowed) and 1.0 (complete overlap allowed).
 - The default value is 1.0 (complete overlap allowed).



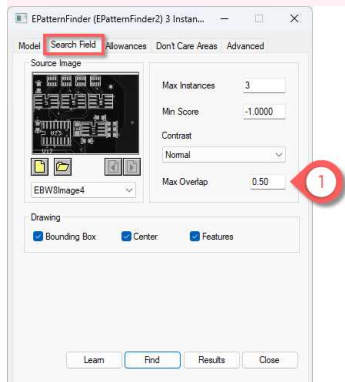
Code

```
finder.SetMaxOverlap(0.50f);
```


Studio

In the **Search Field** tab:

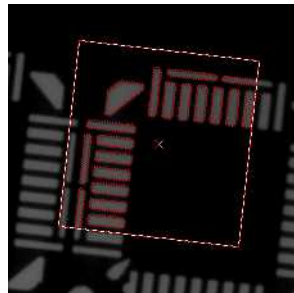
1. Set the **Max Overlap**.



Angle and scale

The angle and scale ranges are defined by a bias and a tolerance:

- For example, with an angle bias of 20° and an angle tolerance of 5°, **EasyFind** returns instances with an angle between 15° and 25° with respect to the learned model (20° ± 5°).
- The default values are 100.0 for the scale bias and 0.0 for the other parameters. That means **EasyFind** returns only patterns with no rotation nor scaling.
- Use the parameters [SetAngleBias](#) and [SetAngleTolerance](#) to set the angle range.
- Use the parameters [SetScaleBias](#) and [SetScaleTolerance](#) to set the scale range.



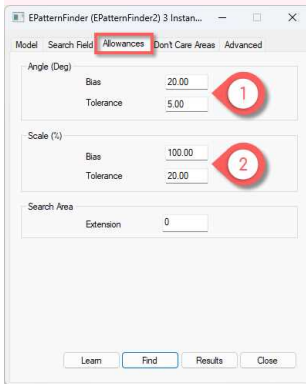
Code

```
finder.SetAngleBias(20.00f);  
finder.SetAngleTolerance(5.00f);  
finder.SetScaleBias(100.00f);  
finder.SetScaleTolerance(20.00f);
```

Studio

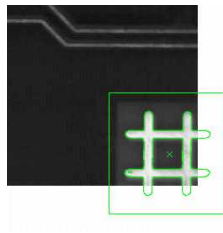
In the **Allowances** tab:

1. In the **Angle (Deg)** area, set the **Bias** and the **Tolerance**.
2. In the **Scale (%)** area, set the **Bias** and the **Tolerance**.



Find partial patterns

- Use the parameter `SetFindExtension` to extend the search area (in pixels) and locate instances of thin structures and consistent edges that are partially out of the search field.



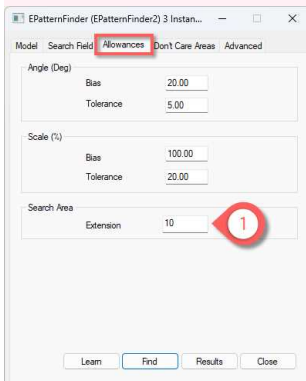
Code

```
finder.SetFindExtension(10);
```

Studio

In the **Allowances** tab:

1. In the **Search Area** area, set the **Extension**.



Local search mode (advanced)



In the multistage approach:

- At the coarsest stage, **EasyFind** finds the pattern occurrence candidates.
 - At each of the following stages, their position and score are refined until the last and finest one. This refining is achieved by searching for better candidates in the neighborhood of each of the ones found in the previous stage.
- Use the parameter [SetLocalSearchMode](#) to set the extent of the neighborhood in which the better candidates are searched.
 - **Basic**: the default local search neighborhood
Search extend: Angle and Scale = 3; X and Y = 3
 - **ExtendedTranslation**: the local search neighborhood is extended along the translation degrees of freedom.
Search extend: Angle and Scale = 3; X and Y = 5
 - **ExtendedAll**: the local search neighborhood is extended along all the degrees of freedom.
Search extend: Angle and Scale = 5; X and Y = 5
 - **ExtendedMore**: same as **ExtendedAll** but with a larger extension.
Search extend: Angle and Scale = 7; X and Y = 9
 - **Custom**: manually define the search extend:
Use [AngleSearchExtent](#) to set the Angle and [ScaleSearchExtent](#) to set the Scale
Use [XSearchExtent](#) and [YSearchExtent](#) to set the translation along X and Y

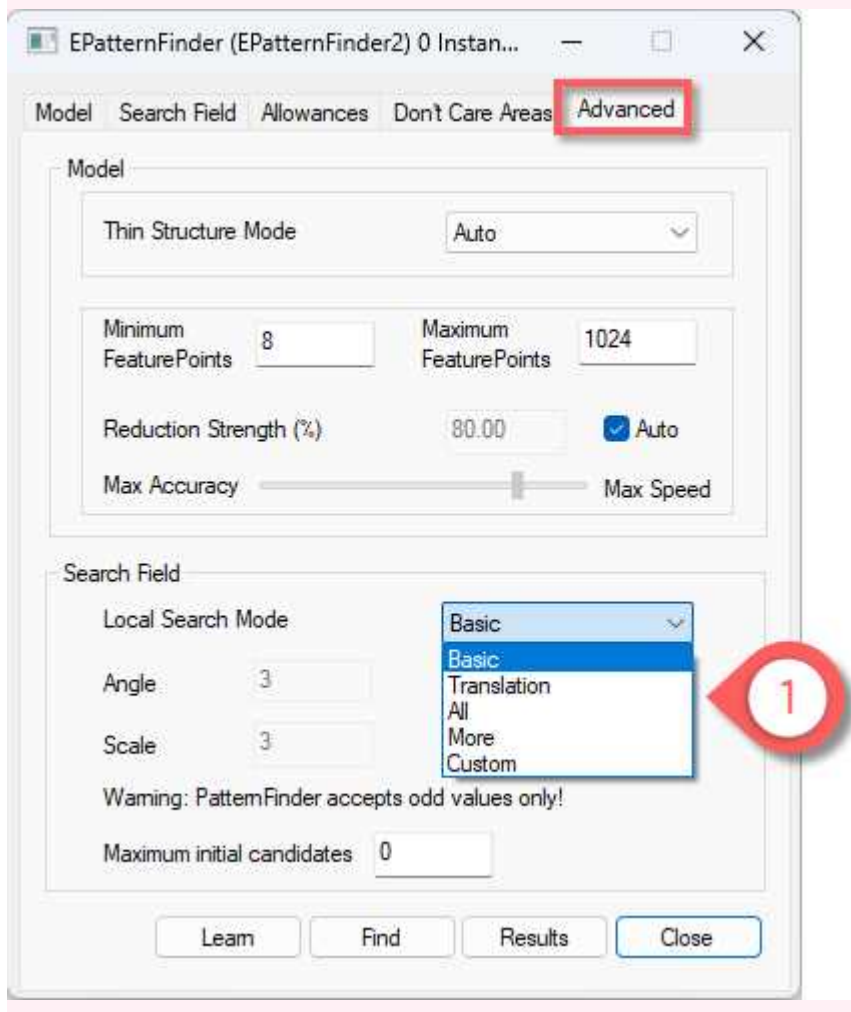
Code

```
finder.SetLocalSearchMode(LocalSearchMode_Basic);
```

Studio

In the **Advanced** tab:

1. In the **Search Field** area, select the **Local Search Mode**.
2. If you select the **Custom** mode, set the **Angle** and the **Scale**.



Maximum initial candidates (advanced)



During the search for matching patterns, **EasyFind** considers a set of candidates and progressively refines it:

- A large number of initial candidates can enable finding difficult or partial matches but at the cost of increasing the processing time.
- A small number of initial candidates can speed up the find process.
- Use the parameter `SetMaxInitialCandidates` to set the maximum number of initial candidates that **EasyFind** considers.
 - This number must be greater than or equal to the number of instances to be found.
 - By default, the value is chosen internally.

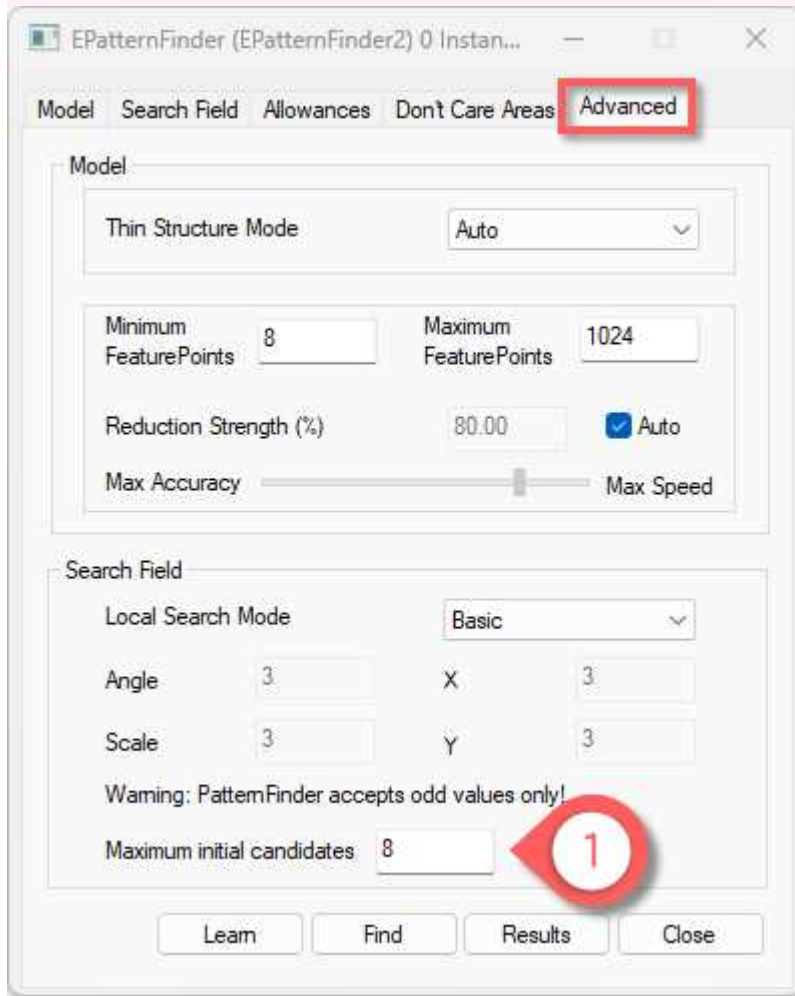
Code

```
finder.SetMaxInitialCandidates(8);
```

Studio

In the **Advanced** tab:

1. In the **Search Field** area, set the **Maximum initial candidates**.

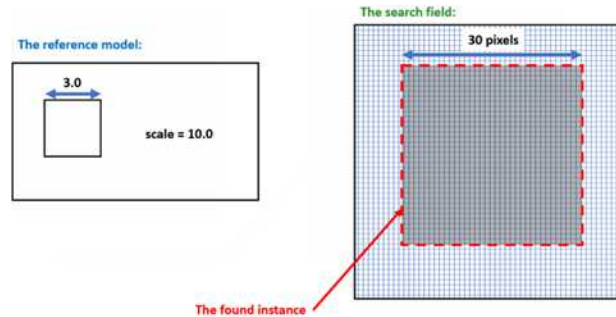


3.13. Vector Model Parameters

The following parameters are set and used in the process: "[Learn the Model from Vectors](#)" on page 172.

Scaling the vector model

- Use the parameter `Scale` to set the scale of your vector model.
 - The scale represents the size of the instances in the search field (in pixels) divided by their size in the `EVectorModel`.

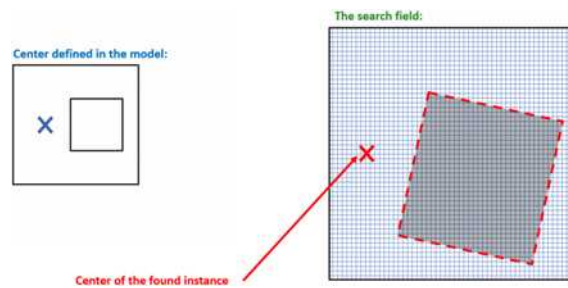


Code

```
myModel.SetScale(10.0f);
```

Centering the vector model

- Use the parameter `Center` to set or get the center of your vector model.
 - The center is the anchor point in the `EVectorModel` coordinate system.
 - After the Find, you can retrieve its position in the search field coordinates as illustrated below.

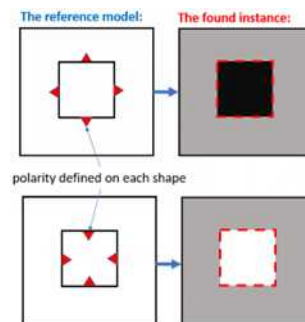


Code

```
myModel.SetCenter( {2.0f, 2.0f} );
```

Orienting the transitions in the vector model

- To search for light to dark or dark to light transitions, set the polarity attribute for the vector model in the [EVectorModel](#).



Code

```
// Sets the polarity of the EPolygonShape  
polygon.SetProperty("polarity", "direct");
```

4. EasyMatch - Matching Area Patterns

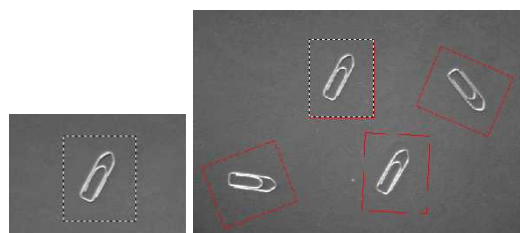
4.1. Workflow

EasyMatch

Reference

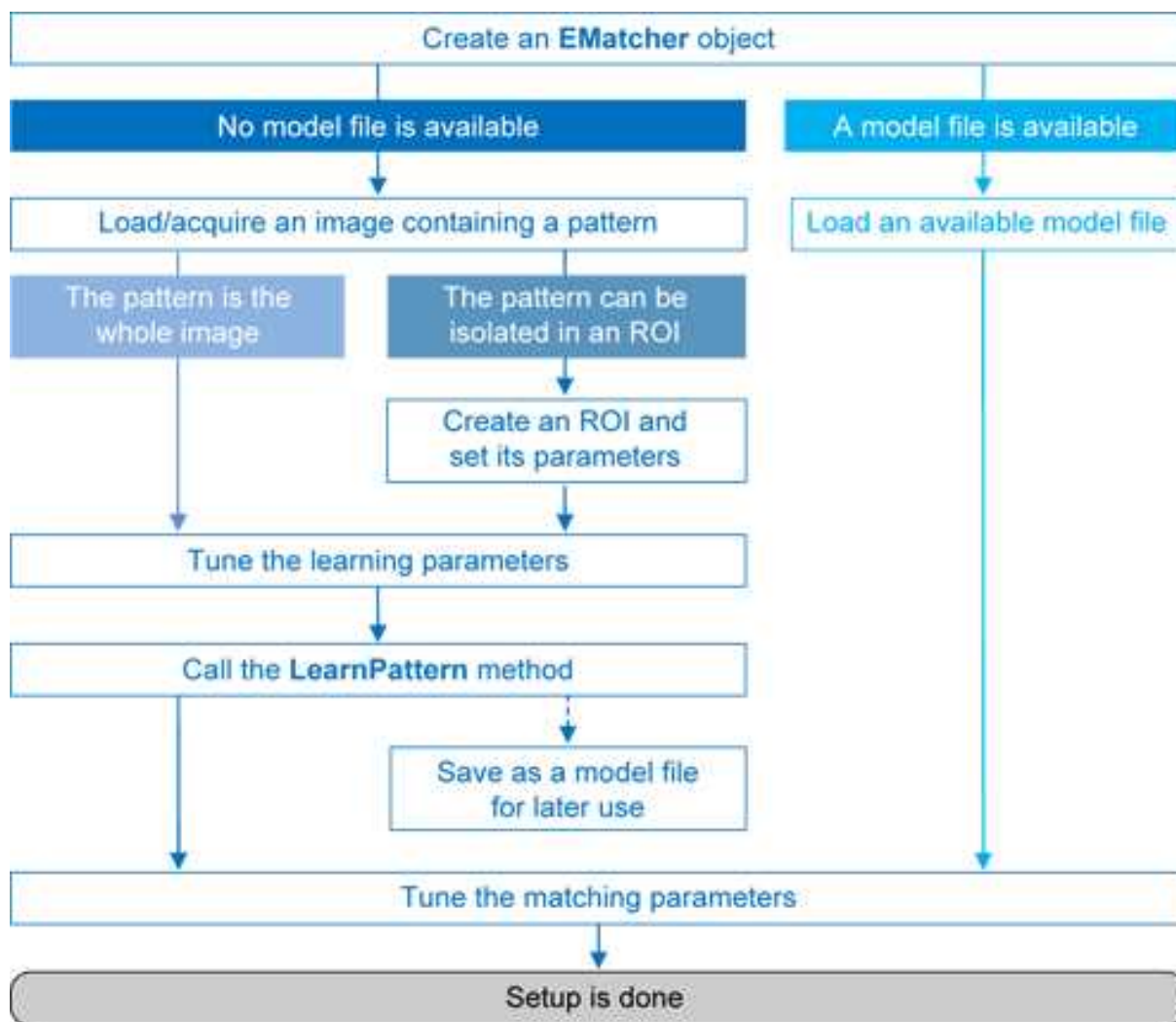
EasyMatch learns a pattern and finds exact matches:

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.

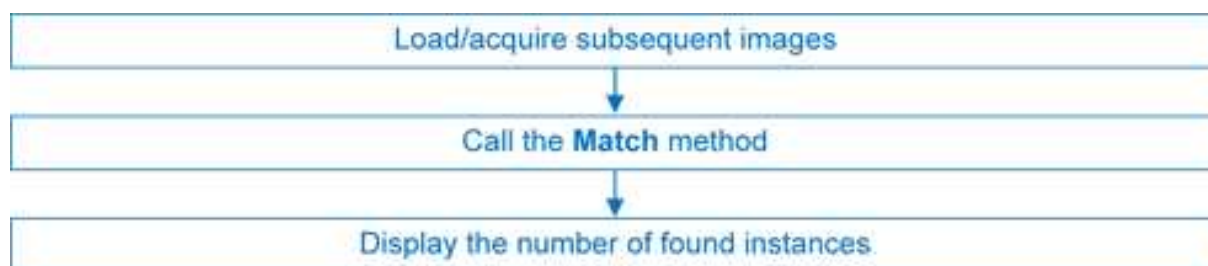


Learning and Matching a pattern

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.

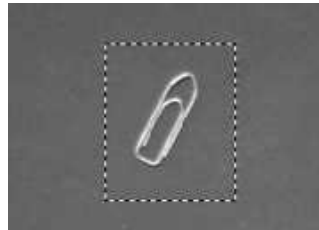
Customize Parameters

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.

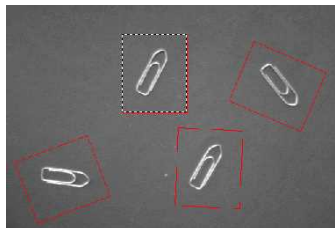
You can set these parameters:

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

Another way of specifying these ignored pixels is to use the region argument of the method [LearnPattern](#). The advantage of using the [ERegion](#) is that it is compatible with the [AdvancedLearning](#) feature, while [DontCareThreshold](#) isn't.

Our code snippet "Pattern Learning with ERegion" illustrates this.

- Overlapping:
 - **EasyMatch** can filter out instances that completely or partially overlap with each other in the search results if the parameter **MaxOverlap** is set to less than 1.0.
The overlap is defined as the area of the intersection of the two instances divided by the area of the smallest instance.
The maximum allowed overlap can take any value between 0 (no overlap allowed) and 1 (complete overlap allowed).
- Extension:
 - **EasyMatch** can match patterns that are partially outside of the matching ROI. While this feature is disabled by default, it can be tuned with the **Extension** parameter. Use this parameter to set the maximum number of pixels of a found pattern occurrence that may be outside the matching ROI.

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.



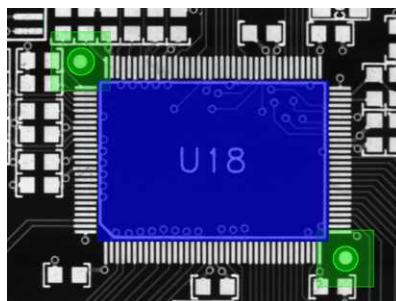
TIP

- 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)

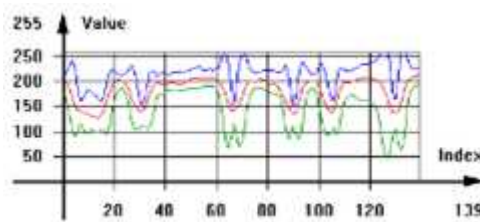
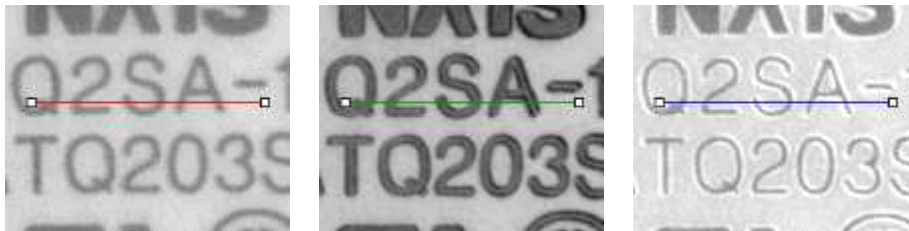
Training

- 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

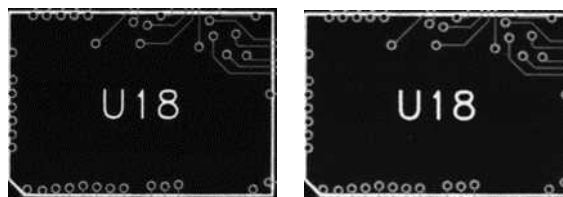
- After the training, use the methods `EChecker2::Save` and `EChecker2::Load` you can save the created model in a single file including all the relevant information and to retrieved it.

5.3. Inspecting an Image

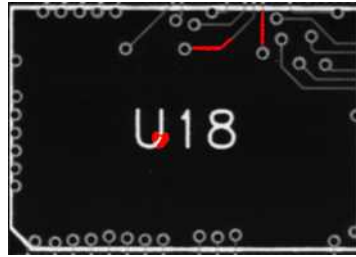
- Use the inspection on an image to assess it towards the trained model.
The process is straightforward:
 - a. The sample image is realigned with the model.
 - b. The gray-level is normalized.
 - c. This gray-level is combined with the high and low threshold images to populate an `ECodedImage2`.
 - d. 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 **EasyObject** processes.
- To perform the inspection, use the method `EChecker2::Inspect`.





PART IV
TEXT AND CODE READING
TOOLS

1. List of Supported Codes

Linear bar codes (1D)





- License: **EasyBarCode**
- Class: **EBarCodeReader**

Symbology	Variants	Checksum	Error correction	Multiple codes reader	Support of grading	Sample
Code 128	—	✓	—	✓	✓	
Ean 8	—	✓	—	✓	—	
Ean 13	—	✓	—	✓	✓	
GS1-128	—	✓	—	✓	✓	
Code 39	Extended and Reduced	Optional	—	✓	—	
Code 93	Extended	✓	—	✓	—	
GS1 DataBar Omnidirectional	RSS14	✓	✓	✓	—	
GS1 DataBar Limited	RSS14 Limited	✓	✓	✓	—	
GS1 DataBar Expanded	RSS14 Expanded	✓	✓	✓	—	
PharmaCode One Track	—	—	—	✓	—	
Codabar	—	Optional	—	✓	—	
Code 2 of 5 Interleaved	—	Optional	—	✓	—	
MSI	—	✓	—	✓	—	
UPC-A	—	✓	—	✓	—	
UPC-E	—	✓	—	✓	—	
ADS Anker	—	✓	—	✓	—	
BC 412	—	✓	—	✓	—	

Symbology	Variants	Checksum	Error correction	Multiple codes reader	Support of grading	Sample
Code 11	—	Optional	—	✓	—	
Code 13	—	✓	—	✓	—	—
Code 2 of 5	Datalogic, Matrix, IATA, Industry, Compressed and Inverted	Optional	—	✓	—	
Code 32	—	✓	—	✓	—	
Code BCD Matrix	—	✓	—	✓	—	
Code CIP	—	✓	—	✓	—	—
IBM Delta Distance A	—	✓	—	✓	—	—
Plessey	—	✓	—	✓	—	
Telepen	—	✓	—	✓	—	
STK Code	—	—	—	✓	—	—
Binary Code	—	—	—	✓	—	







Mail bar codes (1D)

- License: **EasyBarCode**
- Class: **E-MailBarcodeReader**

Symbology	Variants	Checksum	Error correction	Multiple codes reader	Support of grading	Sample
Intelligent Mail Barcode	—	✓	—	✓	—	
Japan Post 4-state Barcode	—	✓	—	✓	—	
POSTNET	POSTNET 5, 6, 9 and 11	✓	—	✓	—	
PLANET	—	✓	—	✓	—	




Matrix codes (2D)

- License: **EasyMatrixCode**
- Class: **EMatrixCodeReader**

Symbology	Variants	Error correction	Multiple codes reader	Support of grading	Sample
Datamatrix ECC000	—	—	✓	✓	
Datamatrix ECC050	—	✓	✓	✓	
Datamatrix ECC080	—	✓	✓	✓	
Datamatrix ECC100	—	✓	✓	✓	
Datamatrix ECC140	—	✓	✓	✓	
Datamatrix ECC200	DMRE	✓	✓	✓	

QR codes (2D)

- License: **EasyQRCode**
- Class: **EQRCodeReader**

Symbology	Variants	Error correction	Multiple codes reader	Support of grading	Sample
QRCode MicroQR	—	✓	✓	✓	
QRCode Model 1	—	✓	✓	✓	
QRCode Model 2	—	✓	✓	✓	

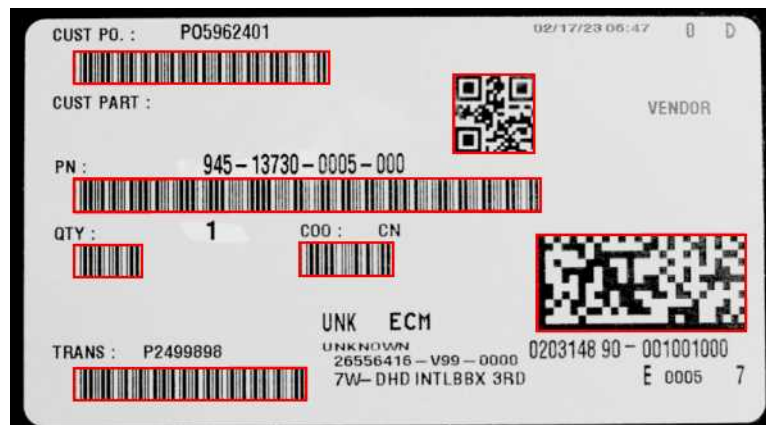
2. ECodeReader - Unified Interface

2.1. Reading Codes

ECodeReader integrates the functionality of **EasyBarCode2**, **EasyMatrixCode2** and **EasyQRCode** in a single unified interface.

In a single **Read** operation, **ECodeReader** locates and decodes the 3 kinds of codes supported by **Open eVision**, namely bar codes, QR codes and data matrix codes.

For more information about these codes and the associated support, please refer to the **EasyBarCode2**, **EasyMatrixCode2** and **EasyQRCode** user guides.



The 3 supported codes in a single image




Reading codes

Use the method **ECodeReader.Read** to locate and read all the codes in an image.

- By default, **ECodeReader** searches for all supported types of code in the image.
 - Use the property **ECodeReader.EnableCodeTypes** if you want to disable (or enable) specific code types.
- **Read** returns a vector of **ECode**, one for each code detected in the image.
 - Use the property **ECodeReader.MaxNumCodesPerType** to set the maximum number of codes to search for each type.
- Use the property **DecodedString** property of **ECode** to retrieve the contents of a code.
 - Use the property **CodeType** if you need to know the specific type of the code.

Code specific results

Use the following properties to retrieve the information specific to a given type of code from an `ECode` instance:

- `ECode.BarCode` returns an `EasyBarcode2.EBarcode` object.
 For more information, refer to the documentation of `EBarcode2`.
- `ECode.MatrixCode` returns a `EasyMatrixCode2.EMatrixCode` object.
 For more information, refer to the documentation of `EMatrixCode2`.
- `ECode.QRCode` returns an `EQRCode` object.
 For more information, refer to the documentation of `EQRCode`.




NOTE: Depending on the effective type of the code, only one of the properties returns a value. The others throw an exception. To know the property to call, use `ECode.CodeType`.

Drawing codes

Use the method `DrawPosition` of the `ECode` instance to draw the position of a code.

Code type specific settings

Use the following properties to set the settings specific to one of the supported types of code:

- `ECodeReader.BarCodeReader` retrieves the underlying instance `EasyBarcode2.EBarcodeReader`.
 For more information about the relevant settings, refer to the documentation of `EBarcode2`.
- `ECodeReader.MatrixCodeReader` retrieves the underlying instance `EasyMatrixCode2.EMatrixCodeReader`.
 For more information about the relevant settings, refer to the documentation of `EMatrixCode2`.
- `ECodeReader.QrCodeReader` retrieves the underlying instance `EQRCodeReader`.
 For more information about the relevant settings, refer to the documentation of `EQRCode`.

Multithreading

`ECodeReader` always parallelizes the location and the decoding of the different types of code.

- `Easy.MaxNumberOfProcessingThreads` has no effect on this behavior.
- However, `Easy.MaxNumberOfProcessingThreads` has the intended effect on the underlying code readers and speeds up the whole process.

Time-out

Use the property `Timeout` to set a time-out to the `ECodeReader.Read` process.

- The time-out is set for each of the underlying code readers that run in parallel. And so it also limits the total `Read` processing time.
- If one of the supported types of code is not present in the image, it is recommended to disable that type. Otherwise the corresponding reader runs and may reach its time-out, slowing the whole process.

2.2. Reading Using a Grid

If the codes in the images are arranged in a regular grid-like fashion:

- Use the dedicated method overloads `ECodeReader.Read` that return `ECodeGrid` objects to improve the reliability and the speed of the reading.
- Use the methods `ECodeGrid.SetEnableCell`, `ECodeGrid.SetEnableRow`, `ECodeGrid.SetEnableColumn` or `ECodeGrid.SetEnableAll` to disable the processing of cells that you know do not contain any code.
- Use the method `GetResults` of the returned `ECodeGrid` object to retrieve the codes read in each cell of the defined grid.



Reading of codes arranged in a grid

3. EasyBarCode - Reading Bar Codes

3.1. Reading Bar Codes

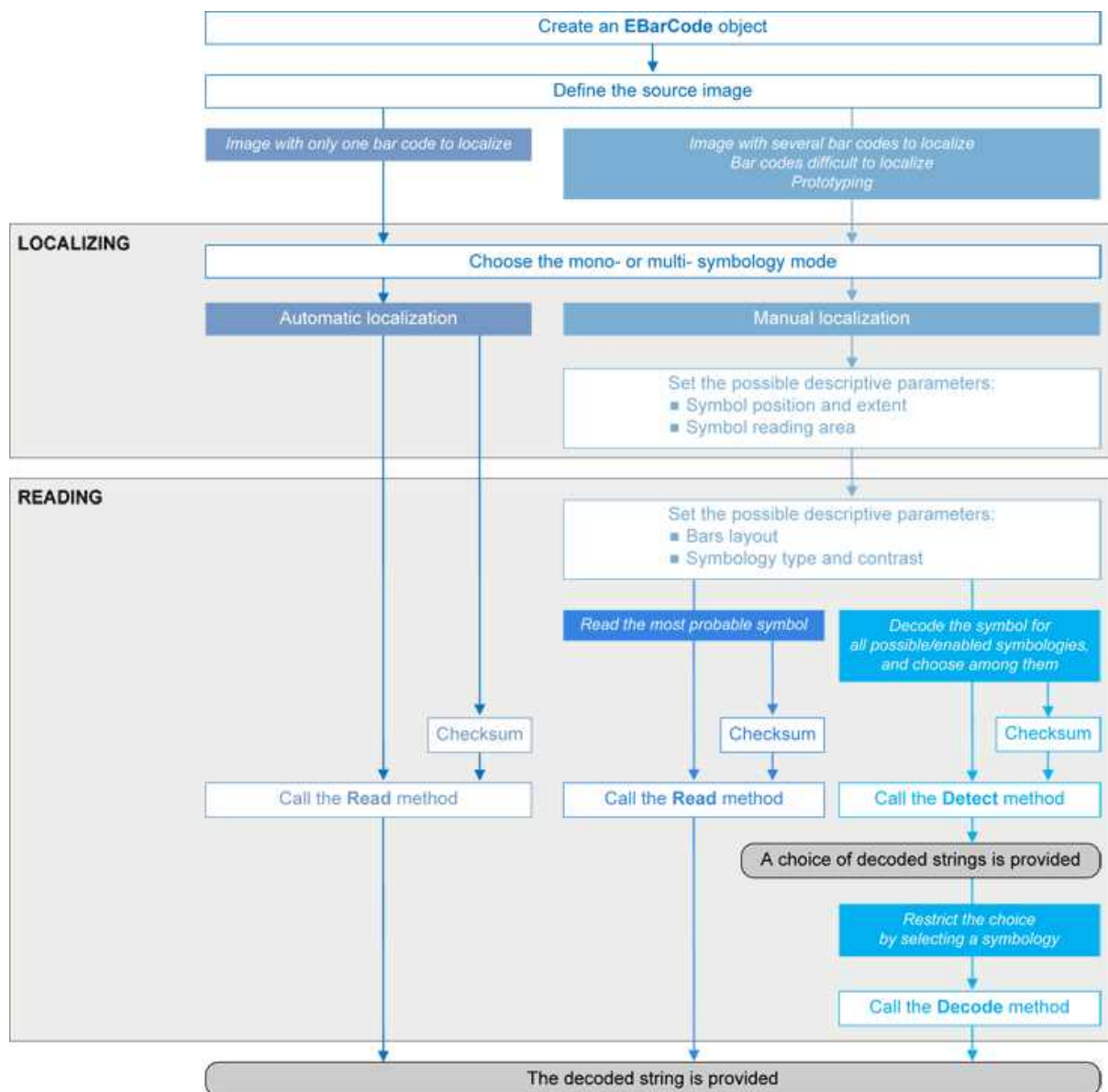
[Reference](#) | [Code Snippets](#)



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:

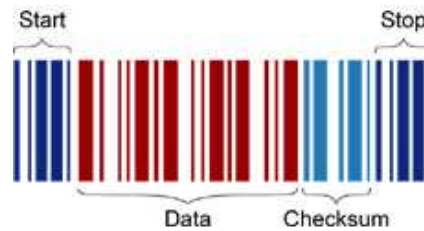
- ADS Anker
- Binary code
- Code 11
- Code 13
- Code 32
- Code 39 Extended (a super-set of Code 39)
- Code 39 Reduced (a subset of Code 39)
- Code 93
- Code 93 Extended
- Code 412 SEMI
- Code 2/5 3 Bars Datalogic
- Code 2/5 3 Bars Matrix
- Code 2/5 5 Bars IATA
- Code 2/5 5 Bars Industry
- Code 2/5 5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

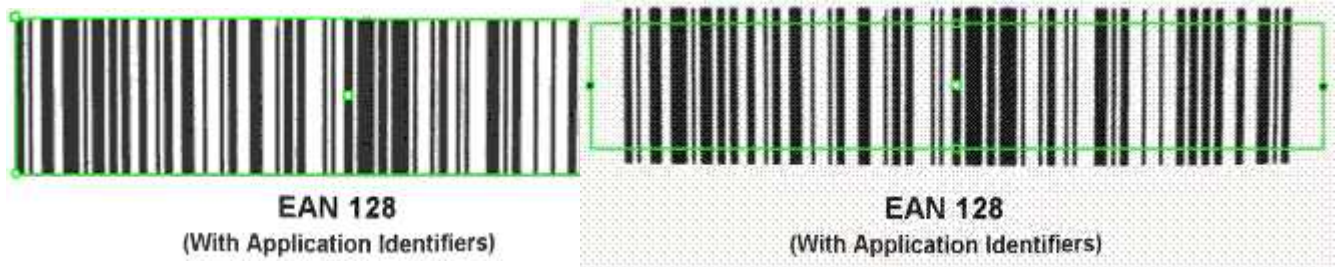
Note: When the bar code contains `\0x00` characters, the `std::string::c_str` method should not be used (since C-strings are terminated by the `\0x00` character). An iterator over the characters should be used instead of a C-string.

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 relative to the bar code bounding box, to take into account slanting bars (Advanced mode!).



Bounding box — graphical appearance
(manual location)

Reading area — graphical appearance (manual
location)

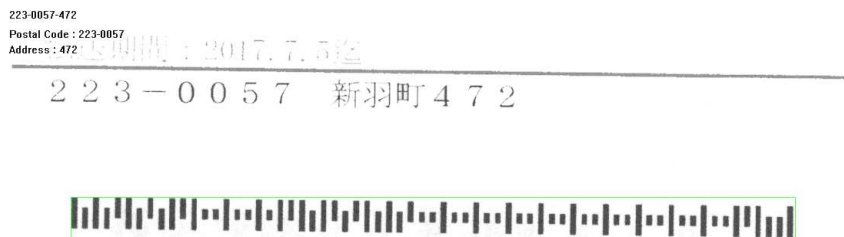
[Read all interpretations \(multi-symbology mode\)](#)

Use [Detect](#) to report the number of possible symbologies in the [NumEnabledSymbologies](#) property, and list the data contents by decreasing likeliness.

Then call the [Decode](#) method in a loop, using [GetDecodedSymbology](#) to walk through the list of successful symbologies in decreasing order of likelihood.

3.2. Reading Mail Bar Codes

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



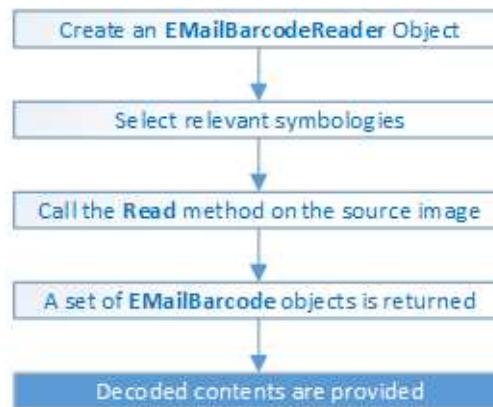
Mail bar code example

[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 bar code is horizontal and read from left to right
- Rotated 90° to the right: The mail bar code is vertical and read from top to bottom

- Rotated 90° to the left: The mail bar code is vertical and read from bottom to top
- Rotated 180°: The mail bar code 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.



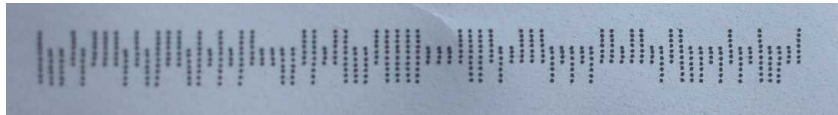
US Intelligent Mail bar code with highlighted position and decoded information

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

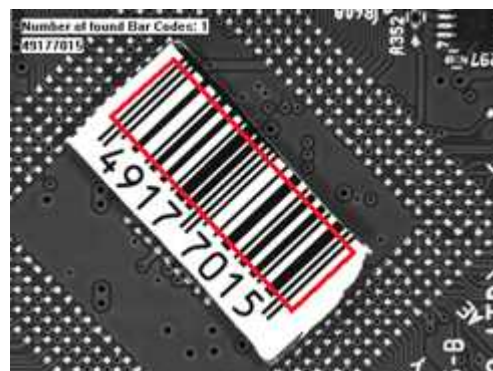
- ValidateChecksum activates the validation of the bar codes checksums, if present.

By default, this property is set to false.

4. EasyBarCode2 - Reading Bar Codes (Improved)

4.1. EasyBarCode2 vs EasyBarCode

- As **EasyBarCode**, **EasyBarCode2** locates and reads bar codes automatically.
- Compared to **EasyBarCode**, **EasyBarCode2** has the following advantages:
 - A faster and more reliable detection of bar codes.
 - The ability to read multiple bar codes at once (with or without grid).
 - The ability to use an ERegion to restrict the search domain.
 - The ability to set a timeout.
 - A better support for many symbologies.
 - A more flexible support of checksums.
- The **EasyBarCode2** classes and methods are defined into the **EasyBarCode2** namespace.
- The following examples illustrate bar codes read by **EasyBarCode2** that cannot be read by **EasyBarCode**.



Occluded bar code (left) and a quiet zone that is too small at the top (right)



Blurry bar codes

4.2. Reading Bar Codes

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

Bar code definition

A bar code is a 2D pattern of parallel bars and spaces of varying thickness that represents a character string.

These bars and spaces are arranged according to a convention named symbology (see below), that specifies the character set and the encoding rules.

Reading bar codes

- Use the method `EBarCode2Reader.Read` to locate the bar codes as `EBarCode` objects in an image and decode them.
- This method returns a vector of `EBarCode` objects for each bar code in the field of view, up to the maximum set with `EBarCode2Reader.MaxNumCodes`. By default, this maximum is set to 1 to optimize the process in the prevalent single-code cases.
- Use `GetDecodedString` to retrieve the contents of an `EBarCode` object.

 Refer to the following code snippet as an example: "[Reading a Bar Code](#)" on page 1

Symbologies

As stated above, a symbology defines how a bar code is encoded using its bars and spaces.

- In **EasyBarCode2**, the following symbologies are enabled by default:

<input type="checkbox"/> Code 128 (incl. Latin-1 chars)	<input type="checkbox"/> Code 39 Reduced
<input type="checkbox"/> Ean 8	<input type="checkbox"/> Code 93
<input type="checkbox"/> Ean 13	<input type="checkbox"/> Code 93 Extended
<input type="checkbox"/> GS1 128	<input type="checkbox"/> GS1 DataBar Omnidirectional
<input type="checkbox"/> Code 39	<input type="checkbox"/> GS1 DataBar Limited
<input type="checkbox"/> Code 39 Extended	<input type="checkbox"/> GS1 DataBar Expanded

- The following symbologies are also supported:
 - Codabar
 - MSI
 - UPC A
 - UPC E
 - ADS Anker
 - BC412
 - Code 11
 - Code 13
 - Code 25 Datalogic
 - Code 25 Matrix
 - Code 25 Iata
 - Code 25 Industry
 - Code 25 Compressed
 - Code 25 Inverted
 - Code 25 Interleaved
 - Code 32
 - Code BCD Matrix
 - Code CIP
 - Code STK
 - IBM Delta Distance A
 - Plessey
 - Telepen
 - Binary Code
 - Pharmacode (one track)
 - RSS14
 - RSS14 Limited
 - RSS14 Expanded
- To setup the symbologies to consider during the reading operation, use the following methods of the [EBarCode2Reader](#) class:
 - [EnableSymbology](#)
 - [EnableSymbologies](#)
 - [EnableAllSymbologies](#)
 - [EnableDefaultSymbologies](#)
 - [DisableAllSymbologies](#)
- Use [GetEnabledSymbologies](#) to check the enabled symbologies.

**NOTE**

[EnableAllSymbologies](#) does not enable the [Code STK](#), the [Binary Code](#) and the [Pharmacode](#) symbologies as these very permissive symbologies can generate a large amount of false positives.

You must use [EnableSymbology](#) explicitly to enable the [Code STK](#), the [Binary Code](#) and the [Pharmacode](#) symbologies.

- Because some symbologies are similar, a given bar code can be detected as corresponding to more than one. Use [EBarCode2.Symbologies](#) to retrieve all compatible symbologies, in order of decreasing confidence.
- Use [GetDecodedString](#) with its optional symbology parameter to retrieve the decoded string as decoded given the symbology passed as parameter. If you omit this parameter, the decoded string will be decoded using the most likely symbology.
- For the GS1-128 and GS1 DataBar Omnidirectional / Limited / Expanded symbologies:
 - Use [GetDecodedString](#) to get the machine-readable code (for ex.:]C11118011215190101).
 - Use [EGs1Translator.GetHumanReadableCode](#) to get the human-readable version (for ex.: (11)180112(15)190101).

Checksum and validation


The checksum of a bar code enables the reader to validate the bar code contents.

- By default, **EasyBarCode2** checks the checksum validity when required by the symbology specifications.
 - The symbologies Code39 (and variants), Codabar, Code 11 and Code 25 (and variants) define but do not enforce a checksum. We call their checksum optional. By opposition, the checksums that are always present are called mandatory. When the checksum is optional, there is no way to know if the barcode contains one from the image alone and no checksum validation is performed by default.
 - For the Pharmacode (one track) symbology, by default, no checksum is computed but additional validations are performed on the bar code vertical uniformity to rule out false positives.
- To reject bar codes if they fail mandatory or optional checksum validation or Pharmacode validation, respectively use the properties [EBarCode2Reader.ValidateMandatoryChecksum](#), [EBarCode2Reader.ValidateOptionalChecksum](#) and [EBarCode2Reader.ValidatePharmaCode](#).
 - Set the property to true, to not return bar codes failing the corresponding validation.
 - By default, [ValidateMandatoryChecksum](#) and [ValidatePharmaCode](#) are set to true and [ValidateOptionalChecksum](#) is set to false.
- Read the property [GetChecksumOK](#) to check the checksum status of the returned [EBarCode2](#) object.
- By default, the checksum characters are included into the string returned by [GetDecodedString](#).
 - Set the optional parameter `includeChecksum` to false to remove these checksum characters from the returned string,



NOTE

As mentioned above, some symbologies, such as [Code 39](#), have optional checksum characters. In those cases, determining their presence automatically is usually not possible. Thus, setting `includeChecksum` to false might crop parts of the decoded string.

 Refer to the following code snippet as an example: "[Reading a Bar Code of a Specific Symbology](#)" on page 1

4.3. Grading Bar Codes


Reference | Code Snippets

To compute the print quality indicators as defined by the **ISO 15416** standard:

1. Set the parameter [ComputeGrading](#) to True.
 2. The print quality of the bar codes is computed during the [Read](#) operation.
 3. Retrieve the grades with the accessor [GetGradingParameters](#) of the class [EBarCode](#).
- This will retrieve an object [EBarCodeGradingParameters](#).

4. Retrieve the numeric grades (0 to 40) directly from the object.
5. Use the method `ConvertToAlphabeticGrade` to convert the numeric grades to alphabetic grades (F to A).

NOTE: At the moment, the grading is only supported for the Ean 13, Code 128 and GS1 128 symbologies. Don't hesitate to contact the support if you need the grading of other symbologies.

 Refer to the following code snippet as an example: "[Grading a Bar Code](#)" on page 1

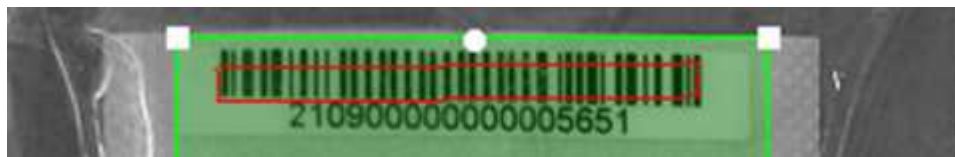
4.4. Advanced Features

[Retrieving the position of a bar code](#)

- Use `EBarCode2::GetPosition` to retrieve the position of a detected bar code.
- This method returns an `EQuadrangle` object representing the detected position of the bar code in the image.

[Reading using an ERegion](#)

- Use `Read(EROIBW8 field, ERegion region)`, the `Read` overload with an `ERegion`, to restrict the domain on which the `EBarCode2Reader` detects and reads the bar codes .



Reading using a grid

If the codes in the images are arranged in a regular grid-like fashion:

- Use the dedicated method `EBarCodeReader.Read` overloads that return `EBarCodeGrid` objects to improve the reliability and the speed of the reading.
- Use the methods `EBarCodeGrid.SetEnableCell`, `EBarCodeGrid.SetEnableRow`, `EBarCodeGrid.SetEnableColumn` or `EBarCodeGrid.SetEnableAll` to disable the processing of cells that you know do not contain any code.
- Use the method `GetResults` of the returned `EBarCodeGrid` object to retrieve the codes read in each cell of the defined grid.

 For an example, see "[Reading a Grid of Bar Codes](#)" on page 1.



Reading of bar codes arranged in a grid

Using a timeout

If you have a defined time constraint:

- Use `SetTimeout` to define a timeout for the `EBarCode2Reader` object. This timeout limits the amount of time available to the `Read` method and forces it to return early if needed.

Detecting small codes in large images

EasyBarCode2 can fail to detect small codes in large images, because it resizes the image before trying to detect the bar code.

- To solve this problem:
 - You can specify the approximate size of the smallest module in your bar code to prevent a too aggressive resizing.
 - This parameter can cause a large increase of the computation times when set to a small value.

If you have small codes in large images that you cannot read:

- a. Set this parameter to 1.
 - b. If the reading is successful with a value of 1, try larger values, for example: 1.5, 2, 2.5, 3.
 - c. The ideal value should be small enough to detect your codes and large enough to not slow the processing too much.
- In the API:
 - Use the method `SetMinModuleSize` to specify the size of the smallest module in your bar code. The input is a floating-point value greater than or equal to 1.
 - Use `SetUseMinModuleSize` to enable or disable this additional treatment.
- NOTE:** The value of this parameter can also be set by using the learning feature (see below).

Specifying the orientation of your codes

For most symbologies, a pattern identifies whether the codes must be read from left to right or right to left.

This is however not the case for the Code STK, Binary Code and Pharmacode symbologies.

For these symbologies, you must know whether to decode the barcode from left to right or from right to left. Use the method `SetReadingOrientation` to specify it.

Learning from given images to improve the detection performances

- **EasyBarCode2** can fine-tune itself to detect more images.

For example, when

$$\max(\text{barWidth}, \text{spaceWidth}) \geq 0.08 \times \min(\text{imageWidth}, \text{imageHeight})$$

or (with `minModuleSize` set to 1) when

$$\min(\text{barWidth}, \text{spaceWidth}) \leq 0.004 \times \min(\text{imageWidth}, \text{imageHeight})$$

bar codes can normally not be detected and a learning is necessary.

- The learning automatically tunes 2 parameters to best fit the set of images you gave it:
 - The `minModuleSize` is set to the highest value that enables reading the maximum number of codes. It is set to the highest value because the higher the `minModuleSize`, the faster the reading.
 - The scales at which you try to find a bar code (an internal parameter not accessible to the user) are modified to read the maximum number of codes depending on the parameters of the learn method (see below).
- `EBarCode2Reader.Learn` has 2 Boolean parameters:
 - `keepDefaultScales` indicates that the scales used by default should not be disabled by the learning (True by default).
 - `addAllScales` indicates that all the scales that allow to detect one code should be enabled (True by default).
- Tips:
 - If you perform a learning on a few problematic images, set both parameters to True.

- If you are only interested in tuning `minModuleSize` or if you perform a learning on a representative set of problematic images, set `keepDefaultScales` to `True` and `addAllScales` to `False`.
- If you perform a learning on a representative set of all the images you want to read, set both parameters to `False`.
- The only benefit to set either of these parameters to `False` is the reading speed. In most cases, the impact should not be important, so you can leave them to `True`.

 Refer to the following code snippet as an example: "[Learning a Bar Code](#)" on page 1

5. EasyMatrixCode - Reading Matrix Codes

5.1. 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.

5.2. EasyMatrixCode

Specifications

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



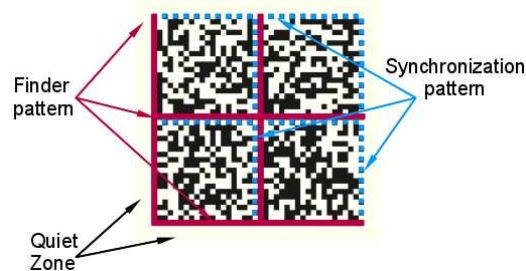
ECC 200, 26x26 cells data matrix code (left) and finder pattern (right)

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.

Supported Symbols

ECC other than the ECC200

NOTE: See the **ISO/IEC 16022** standard for more information.

- Error correction table

	Correctable errors %
ECC000	0
ECC050	2.8
ECC080	5.5
ECC100	12.6
ECC140	25

- Symbol table

Size	Capacity (numerical alphanumerical byte)														
	ECC000			ECC050			ECC080			ECC100			ECC140		
	num	alph	byte	num	alph	byte	num	alph	byte	num	alph	byte	num	alph	byte
9 × 9	6	3	1	n/a			n/a			n/a			n/a		
11 × 11	12	8	5	1	1	n/a	n/a			n/a			n/a		
13 × 13	24	16	10	10	6	4	4	3	2	1	1	n/a	n/a		
15 × 15	37	25	16	20	13	9	13	9	6	8	5	3	n/a		
17 × 17	53	35	23	32	21	14	24	16	10	16	11	7	2	1	1
19 × 19	72	48	31	46	30	20	36	24	16	25	17	11	6	4	3
21 × 21	92	61	40	61	41	27	50	33	22	36	24	15	12	8	5
23 × 23	115	76	50	78	52	34	65	43	28	47	31	20	17	11	7
25 × 25	140	93	61	97	65	42	82	54	36	60	40	26	24	16	10
27 × 27	168	112	73	118	78	51	100	67	44	73	49	32	30	20	13
29 × 29	197	131	86	140	93	61	120	80	52	88	59	38	38	25	16
31 × 31	229	153	100	164	109	72	141	94	62	104	69	45	46	30	20
33 × 33	264	176	115	190	126	83	164	109	72	121	81	53	54	36	24
35 × 35	300	200	131	217	145	95	188	125	82	140	93	61	64	42	28
37 × 37	339	226	148	246	164	108	214	143	94	159	106	69	73	49	32
39 × 39	380	253	166	277	185	121	242	161	106	180	120	78	84	56	36
41 × 41	424	282	185	310	206	135	270	180	118	201	134	88	94	63	41
43 × 43	469	313	205	344	229	150	301	201	132	224	149	98	106	70	46
45 × 45	500	345	226	380	253	166	333	222	146	248	165	108	118	78	51
47 × 47	560	378	248	418	278	183	366	244	160	273	182	119	130	87	57
49 × 49	596	413	271	457	305	200	402	268	176	300	200	131	144	96	63

ECC200

NOTE: See the ISO/IEC 16022 standard for more information.

Square

- Symbol table

Size	Capacity			Correctable errors %	Size	Capacity			Correctable errors %
	num	alpha	byte			num	alpha	byte	
10 × 10	6	3	1	25	44 × 44	288	214	142	14 to 27
12 × 12	10	6	3	25	48 × 48	348	259	172	14 to 27
14 × 14	16	10	6	28 to 39	52 × 52	408	304	202	15 to 27
16 × 16	24	16	10	25 to 38	64 × 64	560	418	277	14 to 27
18 × 18	36	25	16	22 to 34	72 × 72	736	550	365	14 to 26
20 × 20	44	31	20	23 to 38	80 × 80	912	682	453	15 to 28
22 × 22	60	43	28	20 to 34	88 × 88	1152	862	573	14 to 27
24 × 24	72	52	34	20 to 35	96 × 96	1392	1042	693	14 to 27
26 × 26	88	64	42	19 to 35	104 × 104	1632	1222	813	15 to 28
32 × 32	124	91	60	18 to 34	120 × 120	2100	1573	1047	14 to 27
36 × 36	172	127	84	16 to 30	132 × 132	2608	1954	1301	14 to 26
40 × 40	228	169	112	15 to 28	144 × 144	3116	2335	1555	14 to 27

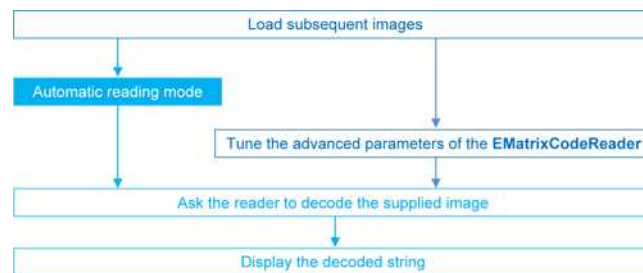
Rectangular

- Symbol table

Size	Capacity			Correctable errors %
	num	alpha	byte	
8 × 18	10	6	3	25
8 × 32	20	13	8	24
12 × 26	32	22	14	23 to 37
12 × 36	44	31	20	23 to 38
16 × 36	64	46	30	21 to 38
16 × 48	98	72	47	18 to 33

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:

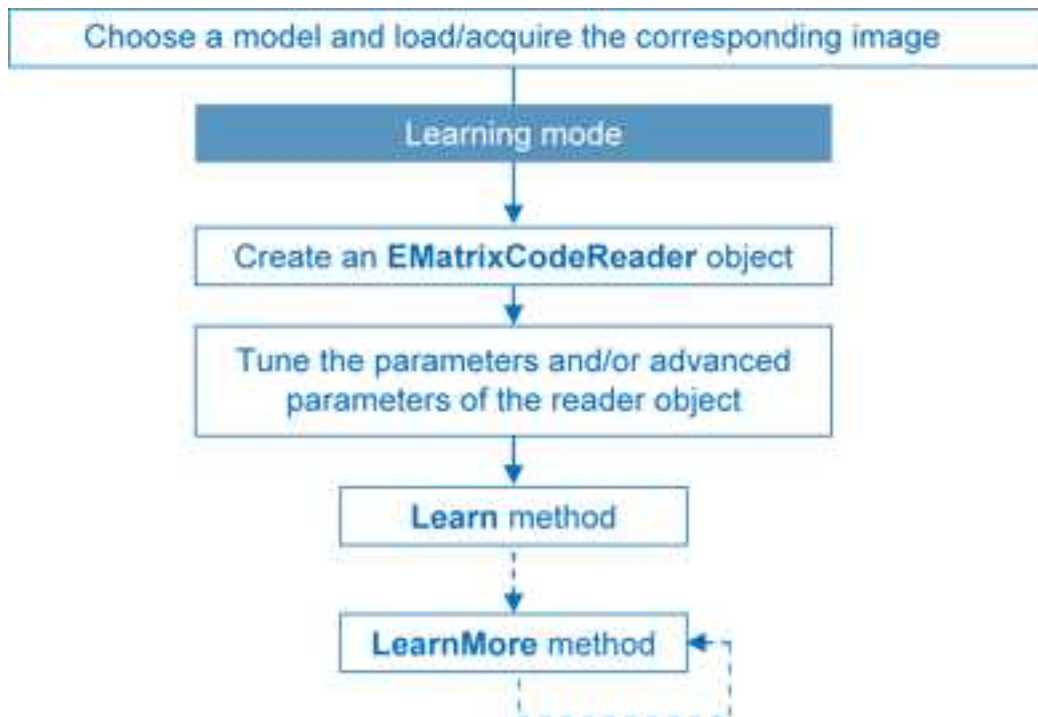
- Its decoded string,
- Its position in the image,
- Its logical size,
- Its encoding type,
- Its grading results,
- Methods to draw the data matrix code on the source image.

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 [GetIso15415GradingParameters](#), [GetIso29158GradingParameters](#) and [GetSemiT10GradingParameters](#) accessors of the [EMatrixCode](#) class.



NOTE

The print quality of the matrix codes is computed during the [Read](#) operation, only if the [ComputeGrading](#) parameter is set to true.

Using GS1 Data Matrix Codes

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

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

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

The structure of GS1-compliant content is as follows:

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

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.



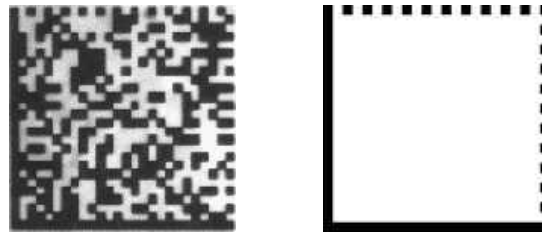
TIP

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

5.3. 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).

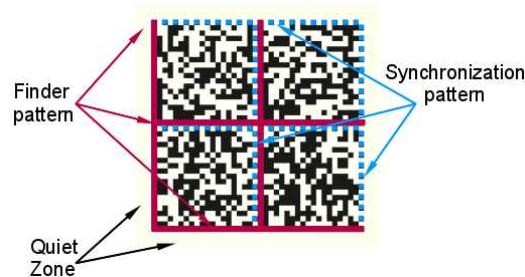


NOTE

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.

Supported Symbols

ECC other than the ECC200

NOTE: See the **ISO/IEC 16022** standard for more information.

- Error correction table

	Correctable errors %
ECC000	0
ECC050	2.8
ECC080	5.5
ECC100	12.6
ECC140	25

- Symbol table

Size	Capacity (numerical alphanumerical byte)														
	ECC000			ECC050			ECC080			ECC100			ECC140		
	num	alph	byte	num	alph	byte	num	alph	byte	num	alph	byte	num	alph	byte
9 × 9	6	3	1	n/a			n/a			n/a			n/a		
11 × 11	12	8	5	1	1	n/a	n/a			n/a			n/a		
13 × 13	24	16	10	10	6	4	4	3	2	1	1	n/a	n/a		
15 × 15	37	25	16	20	13	9	13	9	6	8	5	3	n/a		
17 × 17	53	35	23	32	21	14	24	16	10	16	11	7	2	1	1
19 × 19	72	48	31	46	30	20	36	24	16	25	17	11	6	4	3
21 × 21	92	61	40	61	41	27	50	33	22	36	24	15	12	8	5
23 × 23	115	76	50	78	52	34	65	43	28	47	31	20	17	11	7
25 × 25	140	93	61	97	65	42	82	54	36	60	40	26	24	16	10
27 × 27	168	112	73	118	78	51	100	67	44	73	49	32	30	20	13
29 × 29	197	131	86	140	93	61	120	80	52	88	59	38	38	25	16
31 × 31	229	153	100	164	109	72	141	94	62	104	69	45	46	30	20
33 × 33	264	176	115	190	126	83	164	109	72	121	81	53	54	36	24
35 × 35	300	200	131	217	145	95	188	125	82	140	93	61	64	42	28
37 × 37	339	226	148	246	164	108	214	143	94	159	106	69	73	49	32
39 × 39	380	253	166	277	185	121	242	161	106	180	120	78	84	56	36
41 × 41	424	282	185	310	206	135	270	180	118	201	134	88	94	63	41
43 × 43	469	313	205	344	229	150	301	201	132	224	149	98	106	70	46
45 × 45	500	345	226	380	253	166	333	222	146	248	165	108	118	78	51
47 × 47	560	378	248	418	278	183	366	244	160	273	182	119	130	87	57
49 × 49	596	413	271	457	305	200	402	268	176	300	200	131	144	96	63

ECC200

NOTE: See the ISO/IEC 16022 standard for more information.

Square

- Symbol table

Size	Capacity			Correctable errors %	Size	Capacity			Correctable errors %
	num	alpha	byte			num	alpha	byte	
10 × 10	6	3	1	25	44 × 44	288	214	142	14 to 27
12 × 12	10	6	3	25	48 × 48	348	259	172	14 to 27
14 × 14	16	10	6	28 to 39	52 × 52	408	304	202	15 to 27
16 × 16	24	16	10	25 to 38	64 × 64	560	418	277	14 to 27
18 × 18	36	25	16	22 to 34	72 × 72	736	550	365	14 to 26
20 × 20	44	31	20	23 to 38	80 × 80	912	682	453	15 to 28
22 × 22	60	43	28	20 to 34	88 × 88	1152	862	573	14 to 27
24 × 24	72	52	34	20 to 35	96 × 96	1392	1042	693	14 to 27
26 × 26	88	64	42	19 to 35	104 × 104	1632	1222	813	15 to 28
32 × 32	124	91	60	18 to 34	120 × 120	2100	1573	1047	14 to 27
36 × 36	172	127	84	16 to 30	132 × 132	2608	1954	1301	14 to 26
40 × 40	228	169	112	15 to 28	144 × 144	3116	2335	1555	14 to 27

Rectangular

- Symbol table

Size	Capacity			Correctable errors %
	num	alpha	byte	
8 × 18	10	6	3	25
8 × 32	20	13	8	24
12 × 26	32	22	14	23 to 37
12 × 36	44	31	20	23 to 38
16 × 36	64	46	30	21 to 38
16 × 48	98	72	47	18 to 33

Extended rectangular data matrix DMRE

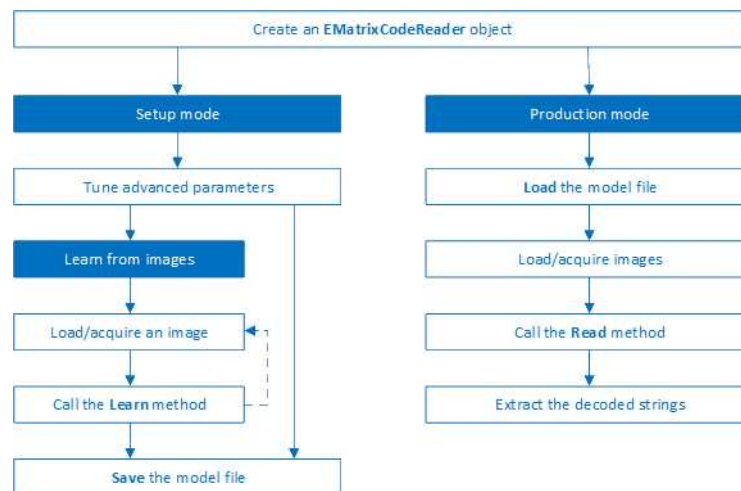
NOTE: See the **ISO/IEC 21471** standard for more information.

- Symbol table

Size	Capacity			Correctable errors %	Size	Capacity			Correctable errors %
	num	alpha	byte			num	alpha	byte	
8 x 48	36	25	16	21 to 36	20 x 36	88	64	42	19 to 35
8 x 64	48	34	22	21 to 36	20 x 44	112	82	54	19 to 34
8 x 80	64	46	30	20 to 35	20 x 64	186	124	82	17 to 31
8 x 96	76	55	36	21 to 38	22 x 48	144	106	70	17 to 32
8 x 120	98	72	47	20 to 36	24 x 48	160	118	78	17 to 31
8 x 144	126	93	61	18 to 33	24 x 64	216	160	106	15 to 28
12 x 64	86	63	41	19 to 34	26 x 40	140	103	68	18 to 32
12 x 88	128	94	62	18 to 33	26 x 48	180	133	88	16 to 30
16 x 64	124	91	60	18 to 34	26 x 64	236	175	116	15 to 28

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 return a vector containing the detected and decoded matrix codes in the image.

The `Read` method provides the following overloads:

- One overload that takes an `ERegion` object as an additional parameter to specify more precisely the search area.
- One overload to specify a search grid when your matrix codes are placed in a regular fashion.

The `EMatrixCode2.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 and to read small codes, 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.

By default, the `Learn` method explores the full range of the `EMatrixCodeReader` internal parameters to search for data matrices in any type of context.

- If the learning process finds valid data matrices, the internal parameters that were necessary to find them are enabled, set and ordered in a way meant to improve both reading speed and capability in similar contexts.
- The user-defined advanced parameters (`MaxNumCodes`, `Timeout`, `ReadMode` and `ComputeGrading`) are not affected by the `Learn` method.
- After setting `MatrixCodeDimensionsRange`, a call to `Read` or `Learn` restricts the processing around that range. This can dramatically improve the speed of the process.

NOTE: However, if you set the property after the learning, the learning is reset and must be redone.

- If the [Learn](#) method is not able to detect any code in the image, it throws an exception.
NOTE: The internal processing structure is not affected in this situation.
- If the [Learn](#) method has been called, all the subsequent images read or learned must have the same dimensions as the one used in the first learn call, otherwise an exception is thrown.

[Restoring the state of an EMatrixCodeReader](#)

- To restore a previously saved [EMatrixCodeReader](#) state, call the [Load](#) method.
- To restore the default state of an [EMatrixCodeReader](#) instance, call the [ResetLearning](#) method.

Computing the Print Quality

[Reference](#) | [Code Snippets](#) | dedicated code snippet: [Inspecting Print Quality Grades](#)

To compute the print quality indicators as defined by BC11, ISO 15415, ISO/IEC TR 29158 (formerly known as AIM DPM-1-2006) and SEMI T10-0701 standards, retrieve the grades with the [GetIso15415GradingParameters](#), [GetIso29158GradingParameters](#) and [GetSemiT10GradingParameters](#) accessors of the [EMatrixCode2.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),
- [GS1?] means that the [GS1] escape character is present there if the previous application identifier has a variable size value,
- {Id} is an application identifier,
- {Value} is the value associated with that identifier.

Example

The string:

```
]d2 [GS1]10GR-1-GNU[GS1]11180112151901012112345
```

is interpreted as follows:

]d2	[GS1]	10	GR-1-GNU	[GS1]	11	180112	15	190101	21	12345
		Id1	Value1		Id2	Value2	Id3	Value3	Id4	Value4

- The first Id is 10, this means the first value is the batch / lot number whose size is at most 20 chars.
 - Thus, the product has a batch / lot number of GR-1-GNU.
 - As batch / lot numbers are of a variable size, a separator is required after GR-1-GNU.
- The second Id is 11, this means the second value is the production date which always consists of 6 digits.
 - The product has a production date of 180112 (12 of January 2018).
 - As production dates have a fixed size, no group separator is required after it.
- The third Id is 15, this means the third value is the “best before date” which always consists of 6 digits.
 - The product has a “best before date” of 190101 (1st of January 2019).
 - As “best before dates” have a fixed size, no group separator is required after it.
- The fourth Id is 21, this means the fourth value is the serial number whose size is at most 20 chars.
 - The product has a serial number of 12345.
 - Although serial numbers are of variable size, 12345 is the last part of the decoded string, so there is no group separator after it.

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

- Use `EGs1Translator::GetHumanReadableCode` to get the human-readable version:

```
(10)GR-1-GNU(11)180112(15)190101(21)12345
```

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.

Reading Using a Grid

If the codes in the images are arranged in a regular grid-like fashion:

- Use the dedicated method `EMatrixCodeReader.Read` overloads that return `EMatrixCodeGrid` objects to improve the reliability and the speed of the reading.
- Use the methods `EMatrixCodeGrid.SetEnableCell`, `EMatrixCodeGrid.SetEnableRow`, `EMatrixCodeGrid.SetEnableColumn` or `EMatrixCodeGrid.SetEnableAll` to disable the processing of cells that you know do not contain any code.
- Use the method `GetResults` of the returned `EMatrixCodeGrid` object to retrieve the codes read in each cell of the defined grid.

 For an example, see ["Reading a Grid of Matrix Codes" on page 1](#).



Reading of data matrices arranged in a grid

Returning Unreadable Codes

Some codes may have their data part so damaged that they cannot be decoded.

- However, you can set `EnableReturnUnreliableCodes` to `True` to return them all the same.
 - These codes are placed at the end of the return vector.
 - Calling `EMatrixCode::GetIsReliable` on them returns `False`.
- Use the usual methods to retrieve the estimated logical size as well as the position of the unreliable codes detected.



Top: input images

Bottom: read results with unreliable detections depicted in red

- Since the data part is no longer a rejection criterion, the likelihood that a false positive unreliable code is returned increases.
 - To filter out these false positive detections, use the value returned by `EMatrixCode::GetReliabilityScore` as a filtering threshold. This score translates as a confidence that the detected object is an actual data matrix code.

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.

- 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 `EMatrixCode2.EMatrixCode` object.
 - Is set to `False` by default.
- The `MatrixCodeDimensionsRange` parameter:
 - Sets the range of valid lengths in pixels that the sides of the data matrices must satisfy to be detected.
 - You can also use this parameter to extend the reading capability to codes relatively small compared to the input image.

After the tuning:

- Use the `Save` method to store the state of the `EMatrixCode2Reader` 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`.

6. EasyQRCode - Reading QR Codes

6.1. Workflow

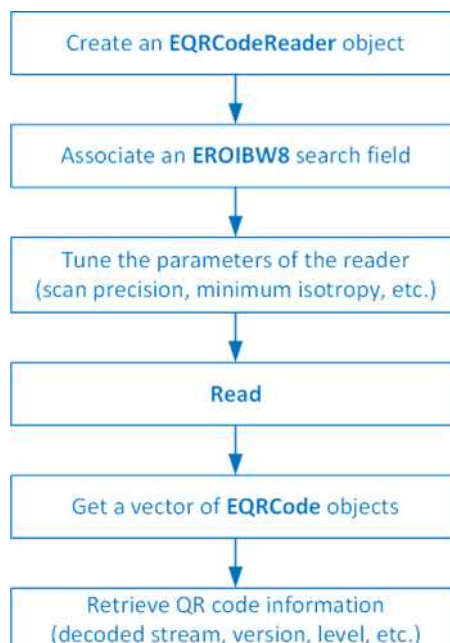
EasyQRCode



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



6.2. 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).

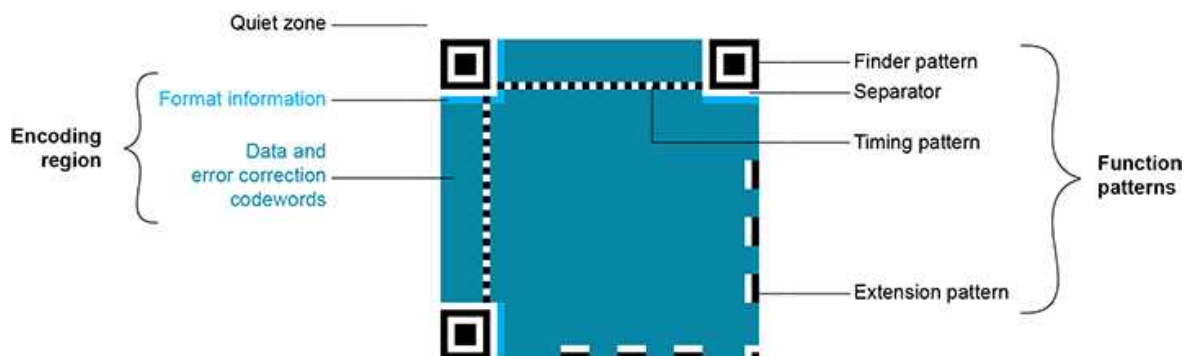
QR code structure

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.

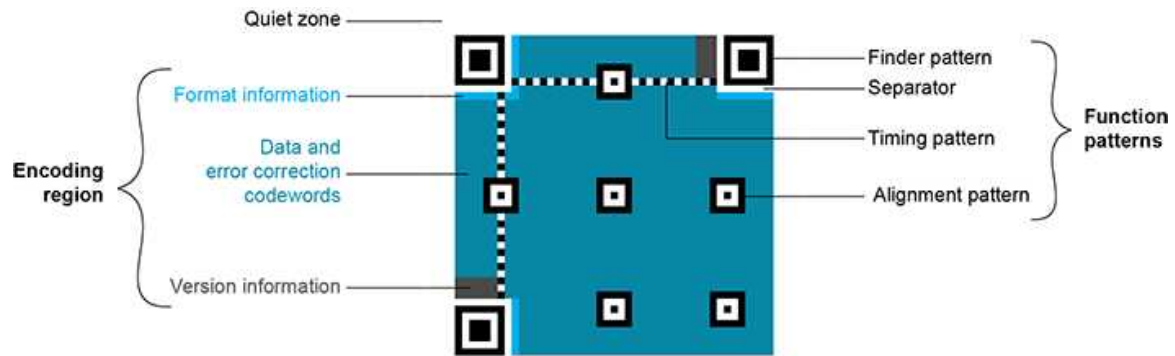
A QR code must be structured with the following elements:

- *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

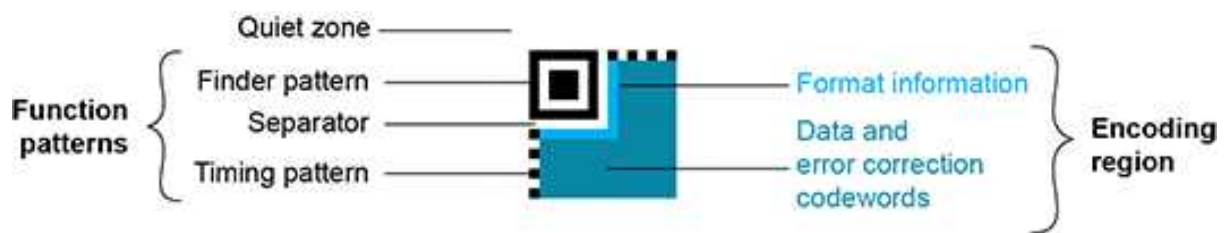
Variants of this structure exist, according to the model, format, or version of the QR code. For instance, model 1 QR codes do not feature alignment patterns but extension patterns. Micro QR codes include only one finder pattern, and no alignment pattern.



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 252 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 252).

6.3. Reading QR Codes

Read a QR code

1. Set a **search field** on an **EROIBW8** image.
 - To restrict the field further, use an optional **ERegion**.
 - To read multiple QR codes placed in a regular fashion, use a specific overload of **Read** to use a grid.
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.
 - For the GS1 QR codes:
 - Use **GetDecodedString** to get the machine-readable code (for ex.: JQ31118011215190101).
 - Use **EGs1Translator::GetHumanReadableCode** to get the human-readable version (for ex.: (11)180112(15)190101).
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 `0xEFBFBD` escape character.
- See the code snippet [Retrieving Information of a QR Code](#).

The `decoded stream` class consists of:

- A coding mode (basic, ECI, FNC1/GS1 or FNC1/AIM).
- An application indicator (if the coding mode is FNC1/AIM, otherwise `0`).

The `decoded data`:

- Is accessible from each part of the decoded stream.
- Is interpreted according to its encoding (numeric, alphanumeric, byte or Kanji) and the `ECI table indicator` (if the coding mode is ECI, otherwise `-1`).
- Can be the raw bit stream (the bit data after unmasking and error correction, but before decoding as a vector of bytes).
- Can be the corresponding `decoded string` (specify a `byte interpretation mode` if the encoding is byte without ECI coding mode or if the ECI table is not supported).
- See also the code snippet [Retrieving the Decoded Data \(Advanced\)](#).

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::GetIso15415GradingParameters` and `EQRCode::GetIso29158GradingParameters` 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 -1. This result affects the overall symbol grade.
 - The Version Additional Parameter is returned as -1 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.

Reading Using a Grid

If the codes in the images are arranged in a regular grid-like fashion:

- Use the dedicated method `EQRCodeReader.Read` overloads that return `EQRCodeGrid` objects to improve the reliability and the speed of the reading.
- Use the methods `EQRCodeGrid.SetEnableCell`, `EQRCodeGrid.SetEnableRow`, `EQRCodeGrid.SetEnableColumn` or `EQRCodeGrid.SetEnableAll` to disable the processing of cells that you know do not contain any code.
- Use the method `GetResults` of the returned `EQRCodeGrid` object to retrieve the codes read in each cell of the defined grid.

 For an example, see ["Reading a Grid of QR Codes" on page 1](#).



Reading of QR codes arranged in a grid

7. EasyOCR - Reading Texts

7.1. Workflow

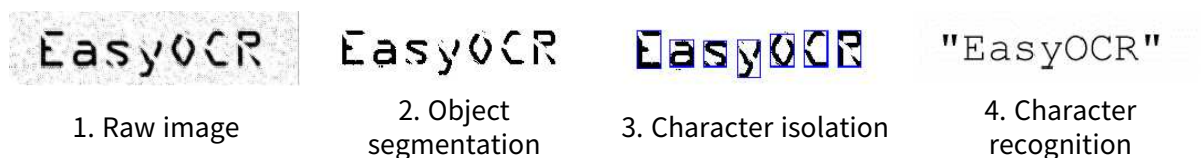
[Reference](#) | [Code Snippets](#)

EasyOCR

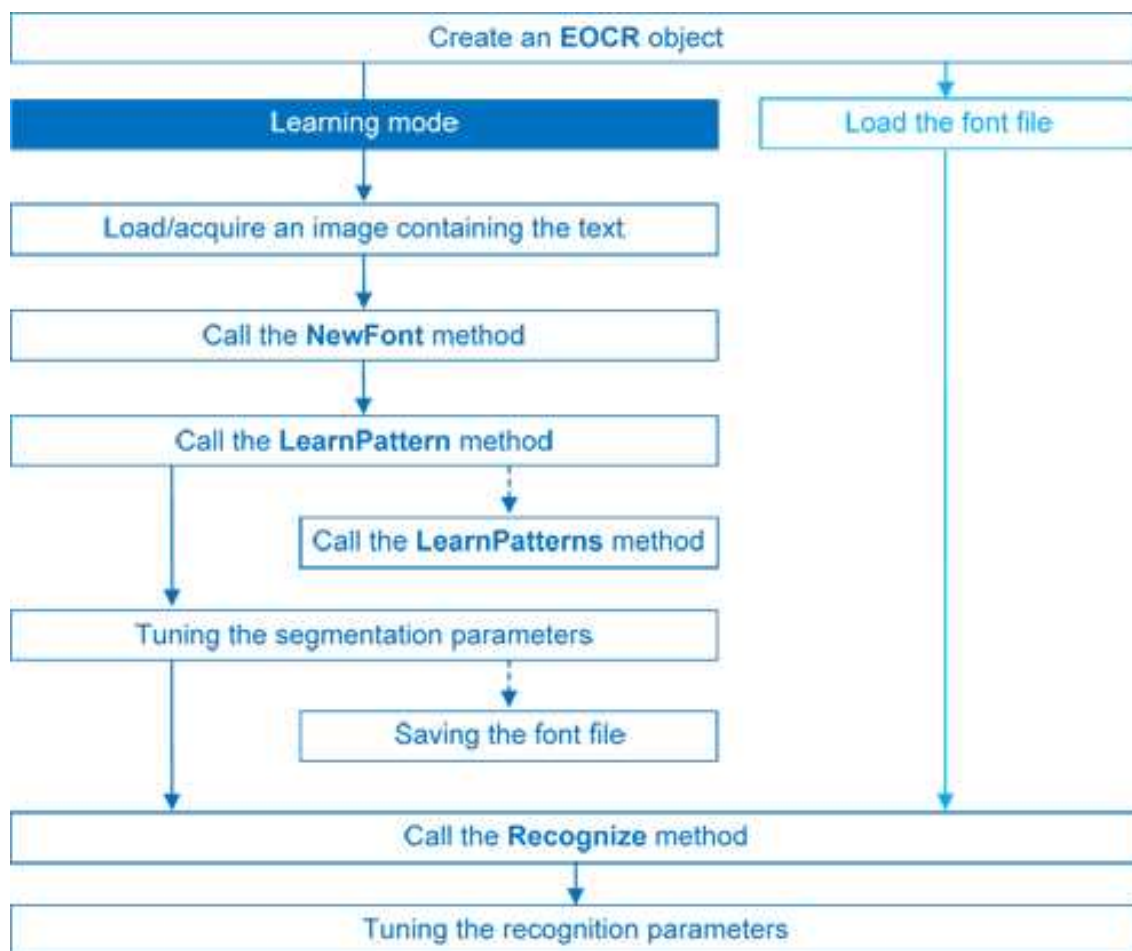
EasyOCR optical character recognition library reads short texts (such as serial numbers, part numbers and dates).

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



7.2. 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).

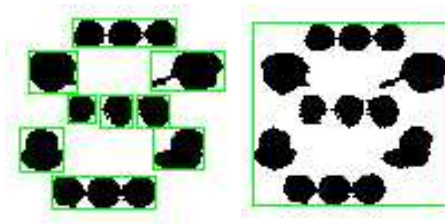
Font files are created as follows:

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](#).

7.3. Segmenting

Segmenting

1. **EasyOCR** analyses the blobs to locate the characters and their bounding box, using one of two **segmentation modes**:
 - **keep objects** mode: one blob corresponds to one character.
 - **repaste objects** mode: the blobs are grouped into characters of a nominal size. This is useful when characters are broken or made up of several parts. When a blob is too large to be considered a single character, it can be split automatically using **CutLargeChars**.



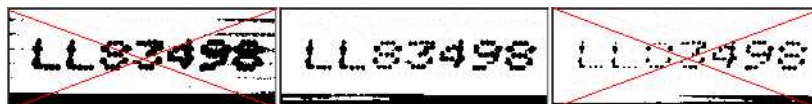
Character segmentation by blob grouping

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

Segmentation parameters

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

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



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

- **NoiseArea**: Blob areas smaller than this value are discarded. Make sure small character features are preserved (i.e., the dot over an "i" letter).
- **MaxCharWidth**, **MaxCharHeight**: Maximum character size. If a blob does not fit in a rectangle with these dimensions, it is discarded or split into several parts using vertical cutting lines. If several blobs fit in a rectangle with these dimensions, they are grouped together.
- **MinCharWidth**, **MinCharHeight**: Minimum character size. If a blob or a group of blobs fits in a rectangle with these dimensions, it is discarded.

- **CharSpacing**: The width of the smallest gap between adjacent letters. If it is larger than **MaxCharWidth** it has no effect. If the gap between two characters is wider than this, they are treated as different characters. This stops thin characters being incorrectly grouped together.
- **RemoveBorder**: Blobs near image/ROI edges cannot normally be exploited for character recognition. By default, they are discarded.

7.4. 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.

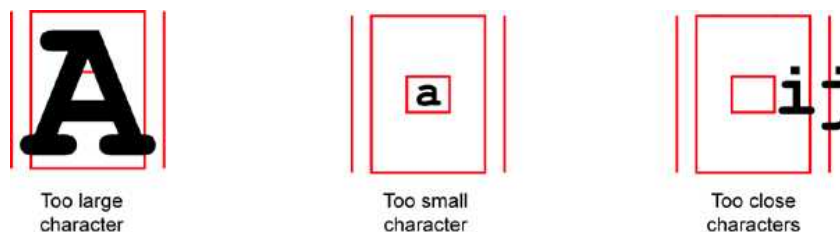
Steps 2 to 4 can be repeated at will to process other images or ROIs. The **Recognize** method can be used as well.

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:

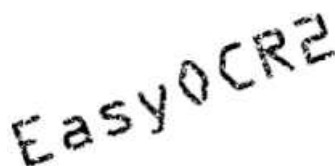
- **Keep objects** mode: a character is a blob; no attempt is made to group blobs, thus damaged characters cannot be handled and small features such as accents and dots may be discarded by the minimum character size criterion.
- **Repaste objects** mode: blobs are grouped to form distinct **characters** if they fit in the maximum character size and are not separated by a vertical gap, thus preserving accents and dots.
- By default, **EasyOCR** considers two characters to be on a different line if their bottom lines are too different (around 30% of the character height). Use the property [LineSpacingMode](#) to change that behavior.

8. EasyOCR2 - Reading Texts (Improved)

8.1. Introduction

8.2. Purpose and Principles

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



Source image

It uses an innovative segmentation method to detect the blobs in an image, then it places textboxes over the detected blobs following a user-defined topology.



Image segmentation (left) and textboxes fitting (right)

The topology specifies the number of lines, words and characters in the text. You can also specify a character type (letter, digit and/or symbol) for each character in the text, to improve the recognition rate and speed.

EasyOCR2 supports the rotation of the text up to 360 degrees, handles non-uniform illumination and textured backgrounds as well as dot-printed or fragmented characters.

To recognize characters, **EasyOCR2** uses a pretrained classifier, powered by Deep Learning technologies, or a classifier that you trained on your character database.

For each input character:

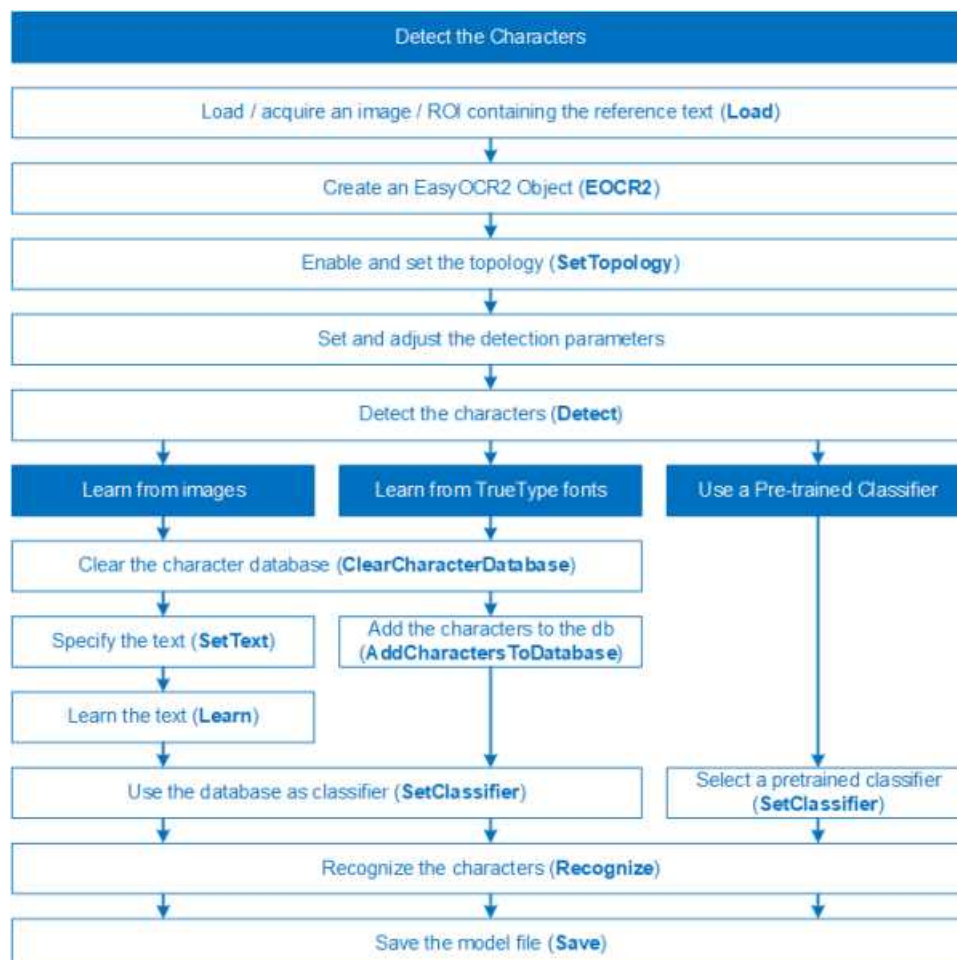
- The classifier calculates a score for all candidate outputs.
- It returns the candidate with the highest score as the recognition result.



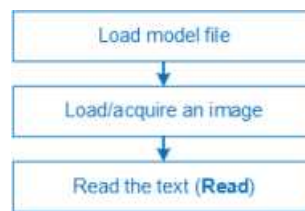
Text recognition

8.3. Workflow

Learning mode



Production mode



8.4. EasyOCR2 vs EasyOCR

EasyOCR2 gives better results than **EasyOCR** when dealing with:

- Unknown text rotation.
- Dotted or fragmented characters.
- Non-uniform illumination or textured backgrounds.
- Complex text topologies.
- When you have the TrueType font files that match the text, you can use these font files directly with **EasyOCR2** for the recognition, while you cannot do it with **EasyOCR**.
- Using the provided pretrained classifiers, **EasyOCR2** can perform the recognition without any preliminary training.

8.5. Using EasyOCR2

8.6. Detect the Characters

See also: code snippet: [Detecting Characters](#)

Detection

EasyOCR2 uses an innovative segmentation method to detect the blobs in an image. Then it places textboxes over the detected blobs following a user-defined topology specifying the number of lines, words and characters in the text (see "[Set the Topology](#)" on page 269).

You can specify a character type (letter, digit and/or symbol) for each character in the text, to improve the recognition rate and speed.

Process

To find characters in an image with EasyOCR2:

1. Load your image.

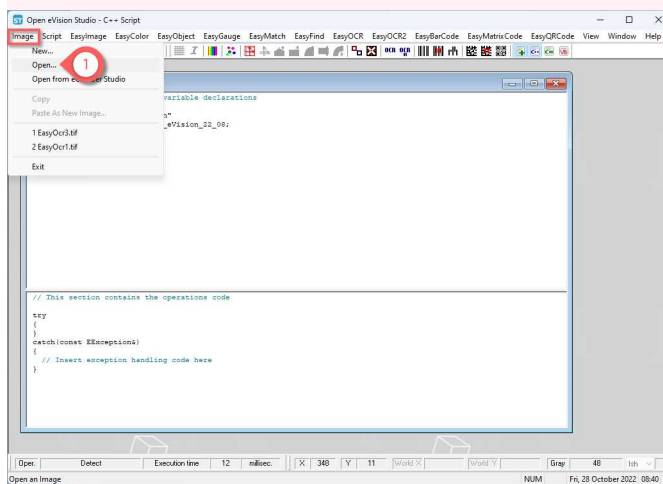
Code

```
EImageBW8 image;
image.Load("image.tif");
```

Studio

In the main menu:

1. Image > Open > select your image (EasyOcr3.tif).



2. Attach an ROI to your image.

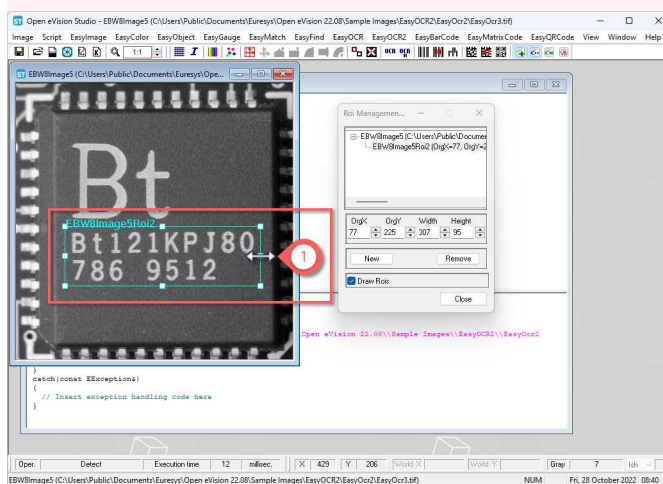
Code

```
EROIBW8 roi;
roi.Attach(&image, 50, 224, 340, 96);
```

Studio

In the image window:

1. Right-click > New ROI > move and resize in the image > Close.



3. Create an EOCR2 instance.

Code

```
EOCR2 ocr2;
```

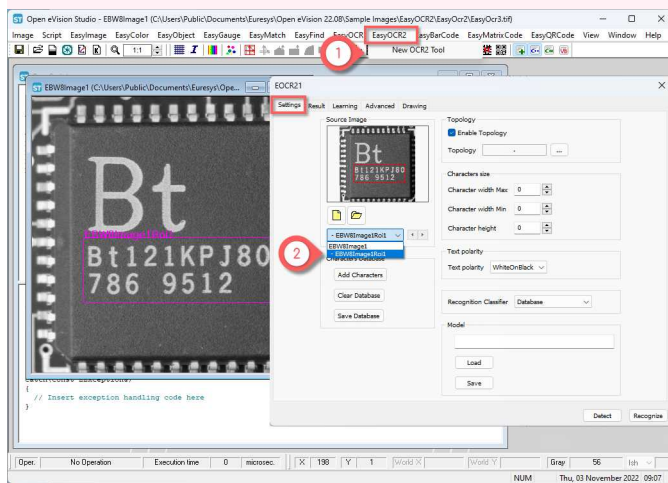
Studio

In the main menu:

1. EasyOCR2 > New OCR2 Tool > name your tool.

In the tool window, in the Settings tab:

2. Select your Source Image.



4. Enable or disable the topology.

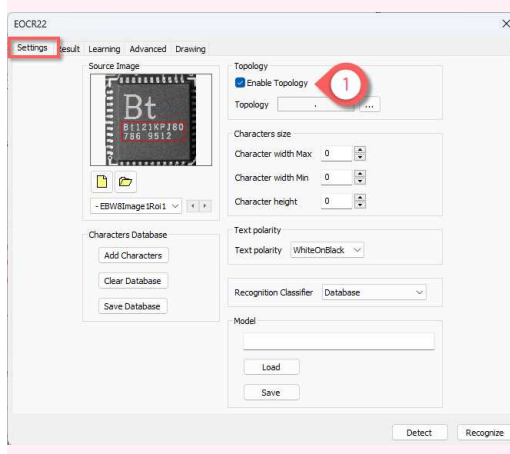
Code

```
ocr2.SetEnabledTopology(true);
```

Studio

In the Settings tab:

1. Check or uncheck Enable Topology.



5. If you are using a topology, configure it as detailed in "Set the Topology" on page 269.

6. Set the following mandatory parameters:

- **CharsWidthRange**: search for characters with a width in this range.
- **CharsHeight**: search for characters with this height.
- **TextPolarity**: search for light characters on a dark background or vice versa.

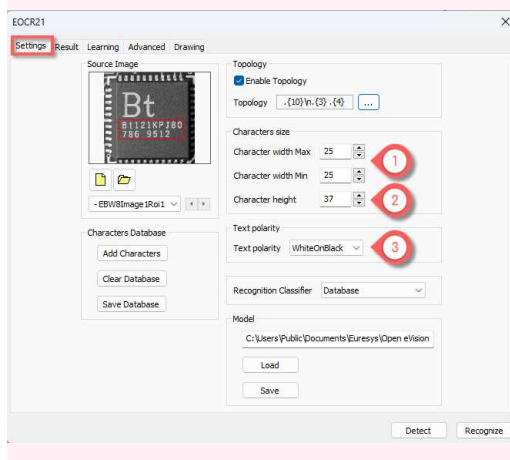
Code

```
ocr2.SetCharsWidthRange(EIntegerRange(25, 25));
ocr2.SetCharsHeight(37);
ocr2.SetTextPolarity(EasyOCR2TextPolarity_WhiteOnBlack);
```

Studio

In the **Settings** tab:

1. Set the **Character width Max** and the **Character width Min**.
2. Set the **Character height**.
3. Select the **Text polarity**.



7. According to your application and needs, adjust the additional parameters:

- **"Segmentation Parameters"** on page 281.
- **"Detection Parameters"** on page 284.
- If you do not use a topology: **"No Topology Parameters"** on page 288

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

- ▶ Blobs that are too large or too small to be considered as parts of a character are filtered out.
- ▶ **EasyOCR2** fits character boxes to the detected blobs according to a given **topology** and **detectionMethod**.
- ▶ The detection returns an **EOCR2Text** structure that contains a textbox and a bitmap image for each character, hierarchically stored in **EOCR2Line** -> **EOCR2Word** -> **EOCR2Char** structures.

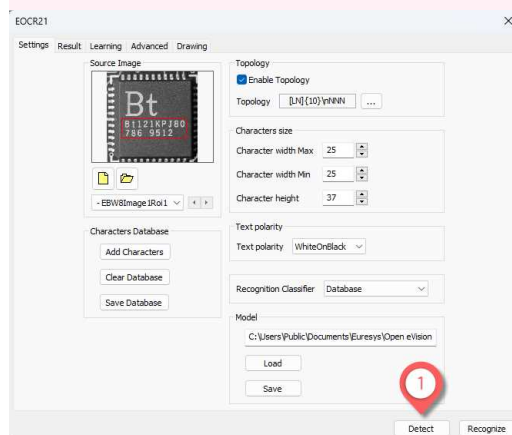
Code

```
EOCR2Text text = ocr2.Detect(roi);
```

Studio

In any tab:

1. Detect



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

Use the resulting character-images to perform:

- ["Learn the Characters" on page 272](#)
- ["Recognize the Characters" on page 276](#)

8.7. Set the Topology

 The following parameters are set and used in the process: "Detect the Characters" on page 264.

Topology



The parameter **Topology** specifies the structure of the text (number of lines, words and characters) as well as the type of characters in the text.

The box-fitting method uses this parameter to structure the textboxes it fits to the detected blobs. And the recognition method limits the number of candidates for each character based on the information of the topology.

- The topology uses the following modified regular expression (Regex) 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 a punctuation character among: ! " # % & ' () * , - . / : ; < > ? @ [\] _ { | } ~
 - "S" represents a symbol among: \$ + - < = > | ~
 - "\n" represents a line break.
 - " " (space) represents a space between two words.
- You can combine these wildcards.
 - For example: [LN] represents an alphanumeric character.
- To specify multiple characters:
 - 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.

Pretrained classifiers

- Currently, when you use pretrained classifiers, not all types of character are recognized:
 - Only "Lu", "N" and "P" are supported.
 - Using "." in your topology only results in a character that is either a uppercase letter, a number or a punctuation character.
 - Using "L" only results in an uppercase letter.
 - Using another character type throws an exception.

Examples

- `[LuN]{3,5}PN{4} \n .{5} LL` represents a text comprised of 2 lines:
 - The first line has 1 word:
 - The word has 3 to 5 uppercase alphanumeric characters followed by a punctuation character and 4 digits.
 - The second line has 2 words:
 - The first word has 5 characters (of any type).
 - The second word has 2 letters (upper- or lowercase).
- `L{3}P N{6} \n L{3}P NPN{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 digits.
- `.{10} \n .{7} \n .{5} .{5} \n .{5} .{7}` represents a text with 4 lines:
 - The first line contains 1 word of 10 characters (of any type).
 - The second line contains 1 word of 7 characters
 - The third line contains 2 words, each of 5 characters.
 - The fourth line contains 2 words of 5 and 7 characters respectively.

Process

 Detect the characters in your image as described in "[Detect the Characters](#)" on page 264.

1. Set the topology either as text.

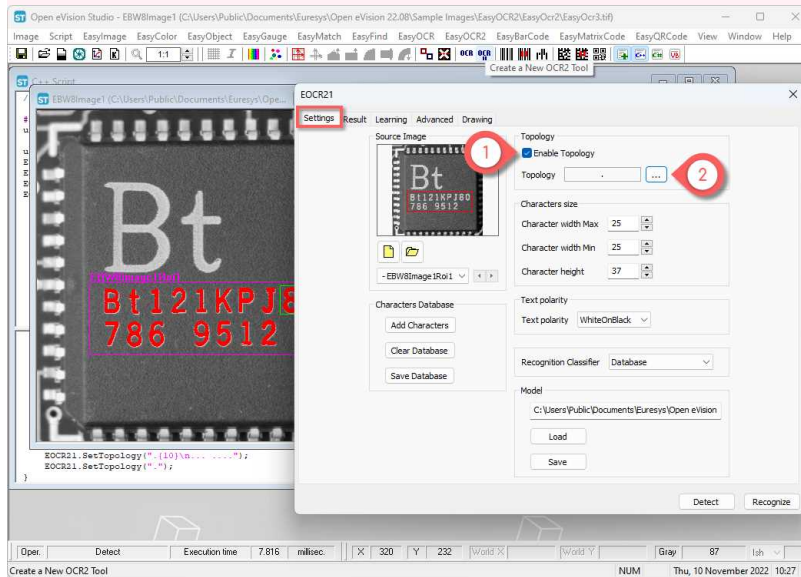
Code

```
ocr2.SetEnabledTopology(true);
ocr2.SetTopology(".{10}\n.{3} .{4}");
```

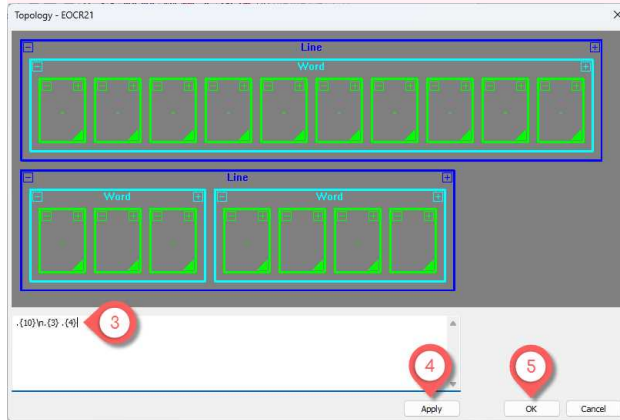
Studio

In the **Settings** tab:

1. Check **Enable Topology**.
2. ... to open the topology editor



3. Enter the topology.
4. **Apply** to update the wizard view.
5. **OK**

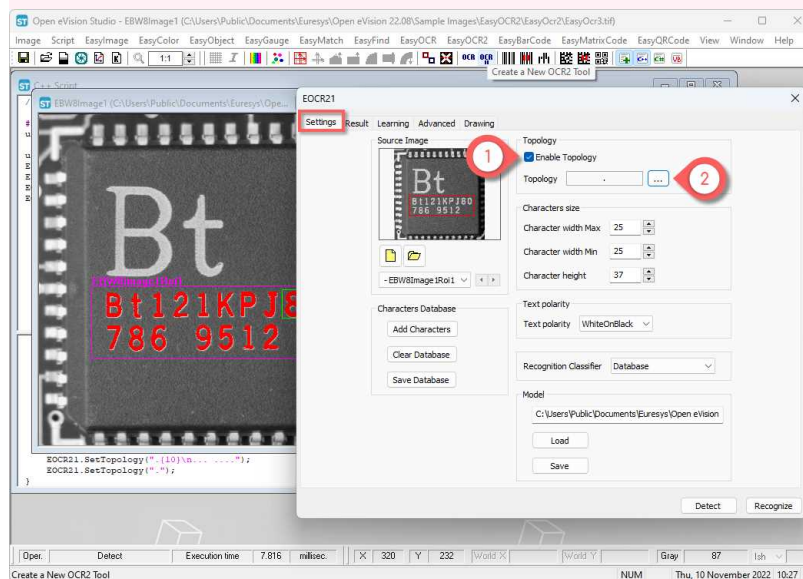


2. Or using the wizard.

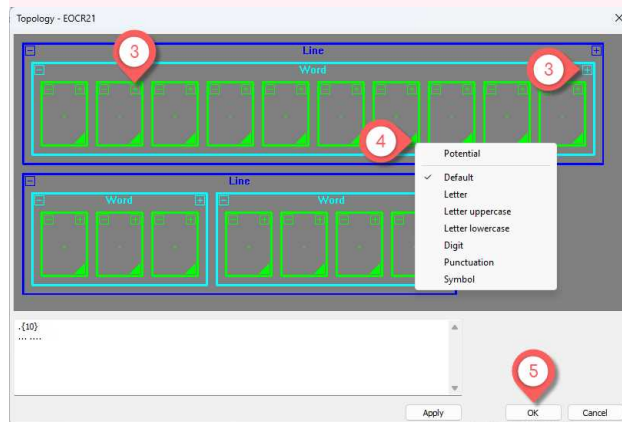
Studio

In the **Settings** tab:

1. Check **Enable Topology**.
2. ... to open the topology editor



3. Use [+] and [-] to add or remove a line, a word or a character.
4. Click the corner to select the character type(s).
5. **OK**



8.8. Learn the Characters

See also: [Learning Characters](#)

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.

Process

📄 Detect the characters in your image as described in "Detect the Characters" on page 264.

1. Set the correct values of the text.

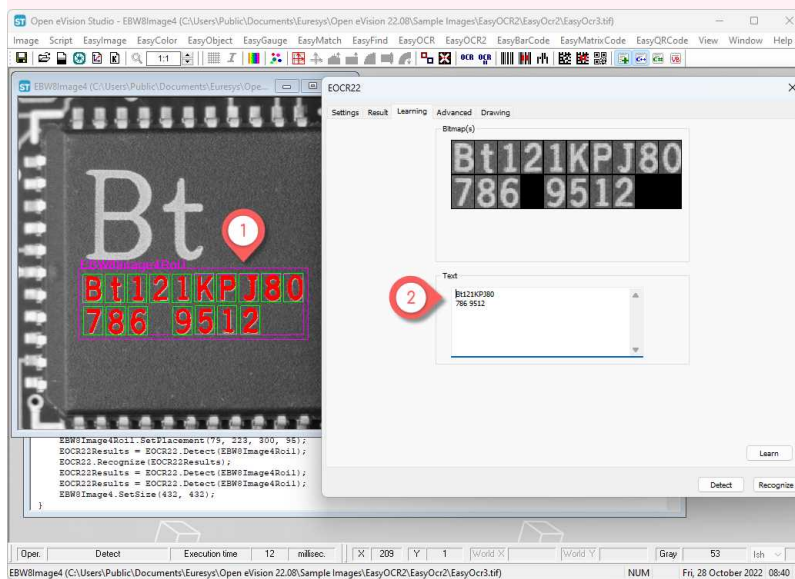
Code

```
text.SetText("Bt121KPJ80\n786 9512");
```

Studio

In the **Learning** tab:

1. Select an element (character, word, line or text) in the image (see "View Elements in Open eVision Studio" on page 279).
2. Enter the correct corresponding text.



2. Add the detected characters and their correct value to the current character database.

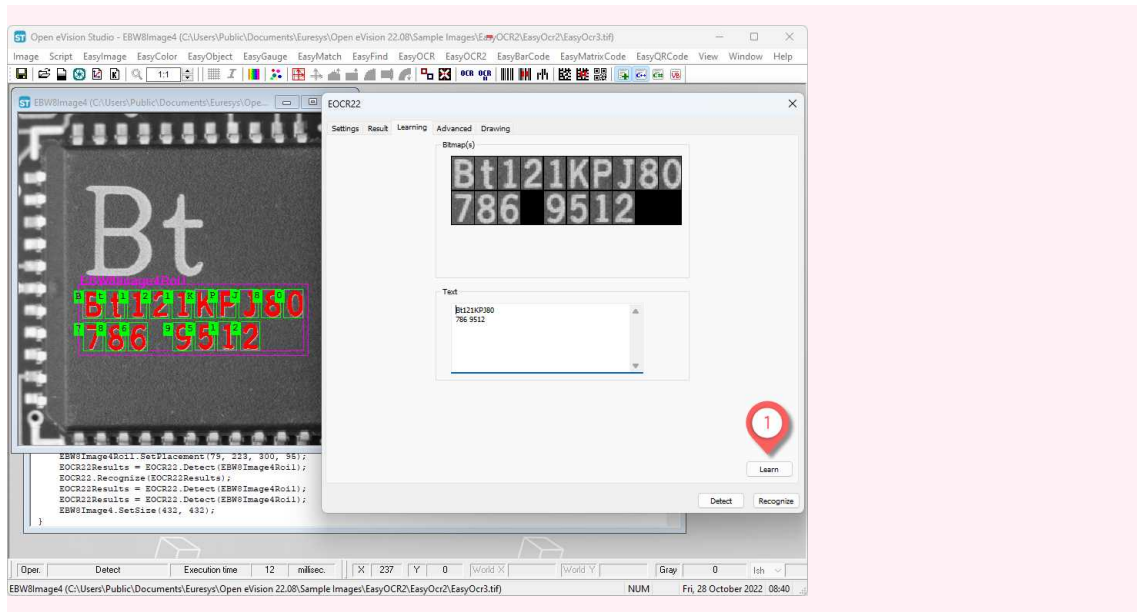
Code

```
ocr2.Learn(text);
```

Studio

In the **Learning** tab:

1. **Learn**



3. Save the current character database.

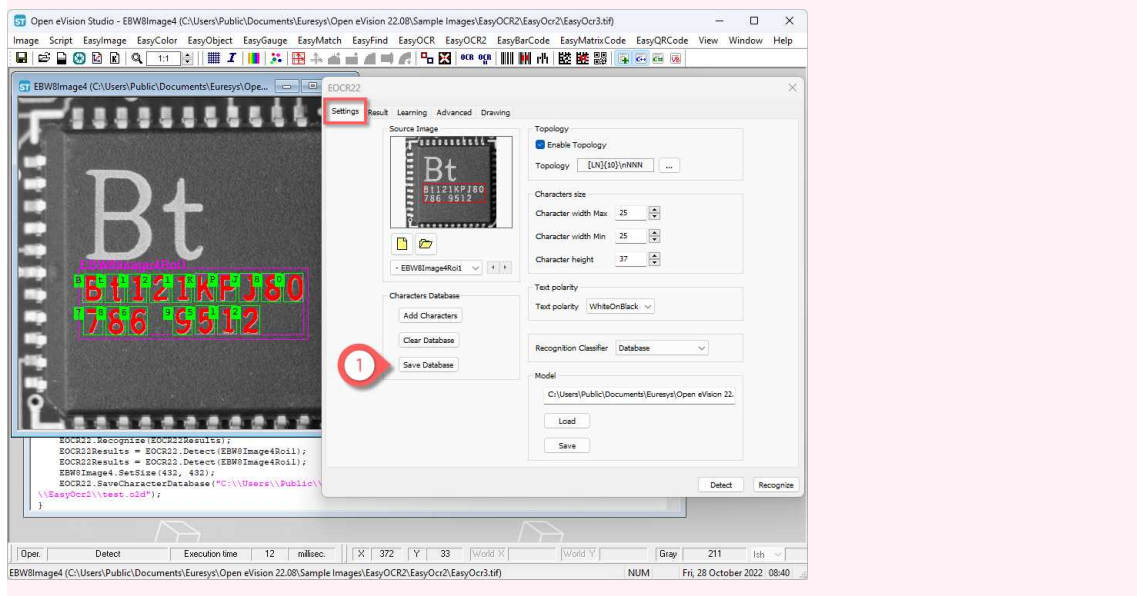
Code

```
ocr2.SaveCharacterDatabase("myDB.o2d");
```

Studio

In the **Settings** tab:

1. **Save Database** > enter the name of your database (.o2d file).



- Alternatively, save the model file, including the detection parameters and the created character database.

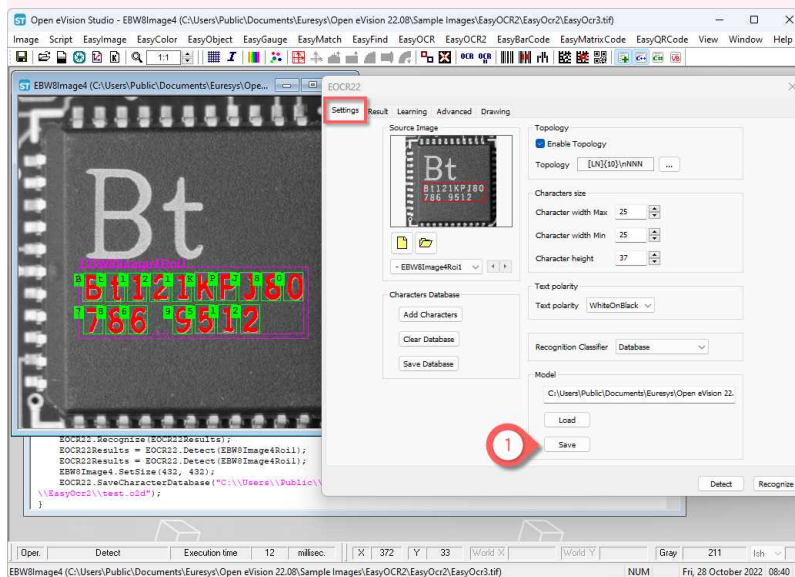
Code

```
ocr2.Save("myModel.o2m");
```

Studio

In the **Settings** tab:

- Save** > enter the name of your model (.o2m file).



Learn characters from a True Type font

- Use **AddCharactersToDatabase** with the path to the True Type font to learn its characters.

Code

```
ocr2.AddCharactersToDatabase("C:\Windows\Fonts\Arial.ttf");
```

Clear the database

- Use `ClearCharacterDatabase` to clear the current character database of any existing learning.

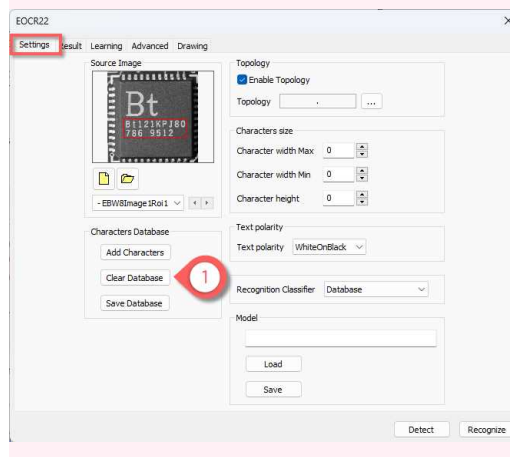
Code

```
ocr2.ClearCharacterDatabase();
```

Studio

In the **Settings** tab:

1. **Clear Database**



8.9. Recognize the Characters

See also: [code snippets: Reading Using TrueType Fonts](#), [Reading Using EOCR2 Character Database](#), [Reading Using EOCR2 Model File](#)

Recognition



To recognize characters, **EasyOCR2** uses a pretrained classifier or a classifier that you trained on your character database.

For each input character:

- The classifier calculates a score for all candidate outputs.
- It returns the candidate with the highest score as the recognition result.

Use the topology to pass information to the classifier about each character. This reduces the number of candidates and improves the recognition rate (see "[Set the Topology](#)" on page 269).



- Use the method [Read](#) or [Recognize](#) to retrieve a string with the recognition results.
 - Call [Read](#) to detect and recognize the characters in one step.
 - Call [Detect](#) to extract the text from the image then [Recognize](#) to recognize the extracted text. This allows you to modify elements of the detected text before the recognition.
- To access more information about the results, use the method [ReadText](#) that returns an [EOCR2Text](#) structure with:
 - The coordinates and the size of each textbox,
 - A bitmap image of each textbox,
 - A list of the recognition scores for each character.

Process

 Detect the characters in your image as described in "[Detect the Characters](#)" on page 264.

1. Select the [Classifier](#) used by [EOCR2](#) for recognition.

By default:

- [EOCR2Classifier_DatabaseClassifier](#): [EOCR2](#) uses the current character database.

Pretrained classifiers used in different contexts:

- [EOCR2Classifier_Industrial_A_Z_0_9_P](#) for characters used in an industrial context without a specific font.
- [EOCR2Classifier_OCRA_A_Z_0_9_P](#) for characters using the OCR-A font.
- [EOCR2Classifier_SEMI_A_Z_0_9_P](#) for characters using the SEMI-OCR font.

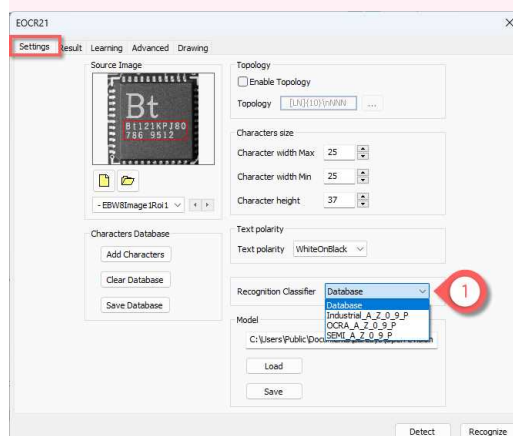
Code

```
ocr2.SetClassifier(EOCR2Classifier_Database);
```

Studio

In the [Settings](#) tab:

1. Select a [Recognition Classifier](#).



2. Recognize or read the decoded text.

Code

```
EOCR2Text text = ocr2.Detect(roi);
std::string result = ocr2.Recognize(EOCR2Text);
```

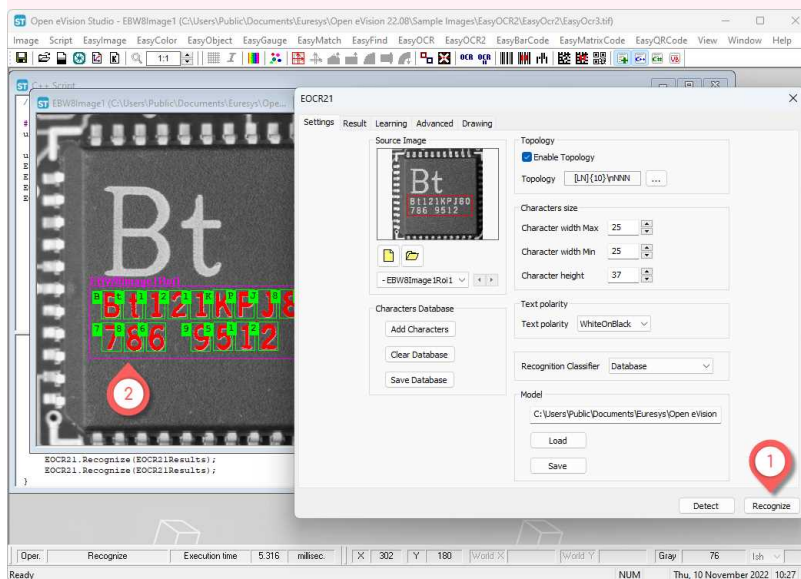
or

```
std::string result = ocr2.Read(roi);
```

Studio

In any tab:

1. Recognize
2. The results are displayed on the image.



Read a model file

- Use [Load](#) to read the model (.o2m) file from your disk.

The model file contains:

- All the detection parameters,
- The topology,
- The reference character database.

Overriding the classifier

- Use [SetClassifier](#) with a symbol and a classifier to override the current classifier and assign another classifier ([EOCR2Classifier](#)) to a specific symbol or a combination.

NOTE: Currently you can override only the [EOCR2Classifier_DatabaseClassifier](#).

Accelerate the recognition

A character is expected to be recognized in 4 ms, half of this with multithreading and even less with a GPU.

Multithreading

- With pretrained classifiers, use `Easy::GetMaxNumberOfProcessingThreads` to multithread the recognition and to determine how many threads you can use for the recognition.

GPU acceleration

- You can also use a compatible GPU to accelerate even more the recognition.
 - Use `Easy::IsGPUAvailable` to determine if there is a compatible GPU available.
 - Use `EOCR2::SetEnableGPU(true)` to enable the GPU-recognition.

8.10. Open eVision Studio Tools

8.11. View Elements in Open eVision Studio

Drawing

In **Open eVision Studio**, you can display the elements you want on your image (blobs, characters, words, lines, text and results). You can also choose a color for each element and configure the display of the results.



TIP

Enable the drawing of the elements that you need to select during the learning process ("[Learn the Characters](#)" on page 272).

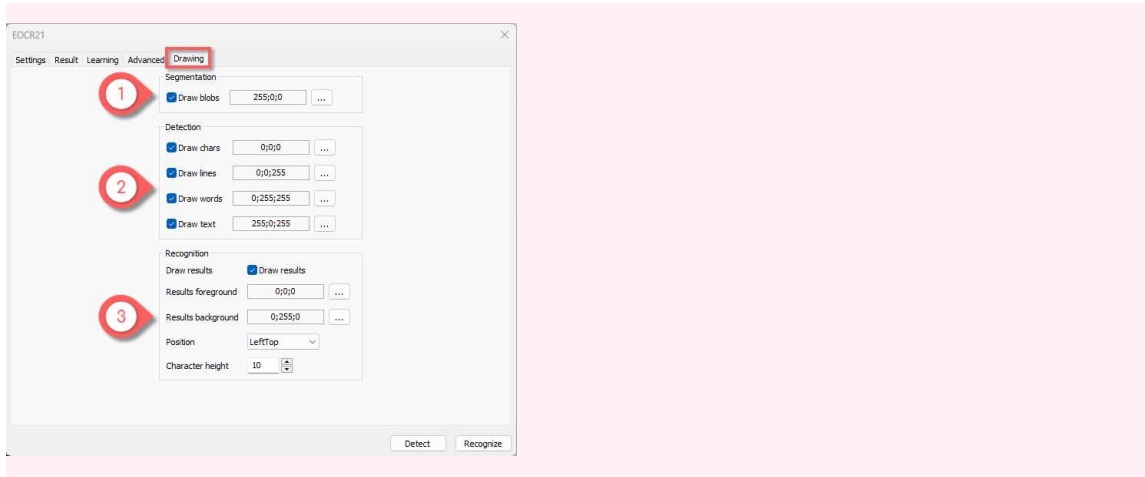
Process

1. In **Open eVision Studio**, display the elements you want in the chosen drawing color.

Studio

In the **Drawing** tab:

1. In the **Segmentation** area > check to display the blobs > to choose the color.
2. In the **Detection** area > check to display the elements > to choose the color.
3. In the **Recognition** area > check to display the results > configure the display.



8.12. View Results in Open eVision Studio

Drawing

In **Open eVision Studio**, you can display the results of a character reading including the matching score for all possible candidates. The list of candidates depends on the topology.

Process

1. In **Open eVision Studio**, display the results for the selected character.

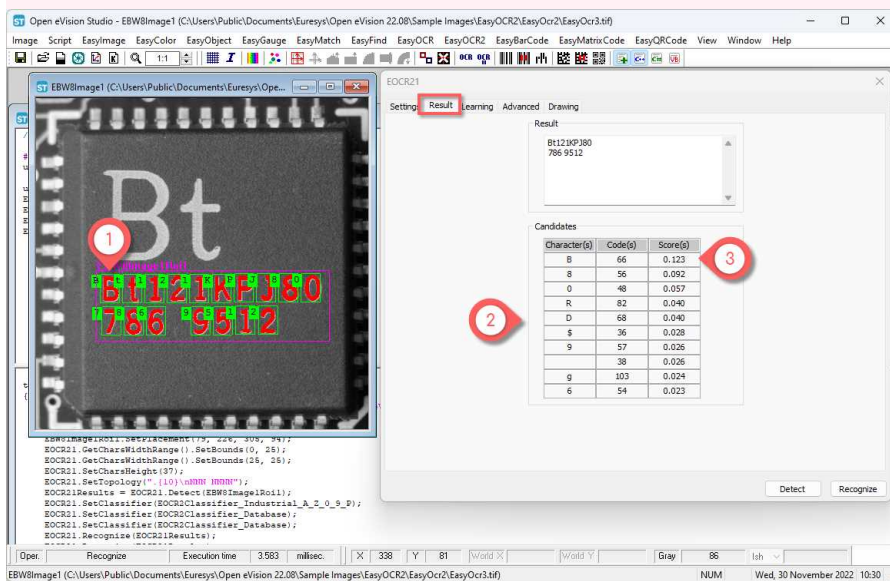
Studio

In the image:

1. Select a character in your image.


In the **Result** tab:

2. The table lists the score for each candidate.
3. The character with the higher score is selected as recognized.



8.13. Setting the Parameters

8.14. Segmentation Parameters

 The following parameters are set and used in the process: "[Detect the Characters](#)" on page 264.

Select the segmentation method

- Use the parameter `SegmentationMethod` to select the algorithm used for segmentation:
 - `EOCR2SegmentationMethod_Global`: the global segmentation uses a simple threshold. It is faster and best suited for a clear background.
 - `EOCR2SegmentationMethod_Local` (default): the local segmentation is more complex and is best suited for a non uniform background.

Configure the global segmentation

- Use the parameter `GlobalSegmentationThresholdMode` to set the threshold computation method for the segmentation. This parameter is a `EThresholdMode`.
- Use the parameter `EnableSecondPassGlobalSegmentation` to perform the segmentation twice during the first pass to have more accurate results when the text background is not just plain.
 - This functionality is only available when `GlobalSegmentationThresholdMode` is set to the minimum residue (`MinResidue`).

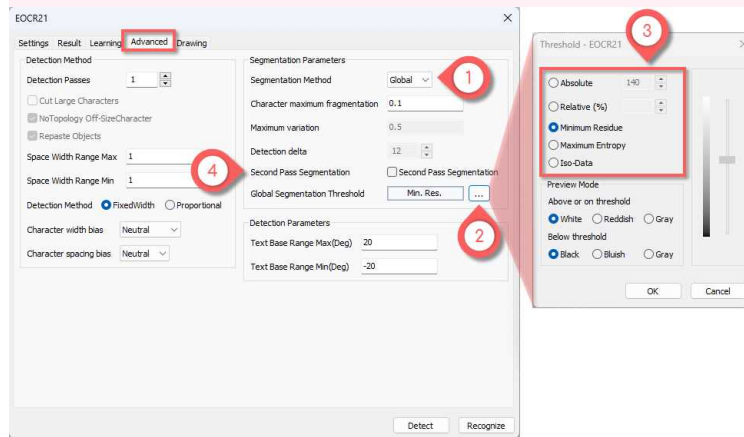
Code

```
ocr2.SetSegmentationMethod(EOCR2SegmentationMethod_Global);  
ocr2.SetGlobalSegmentationThresholdMode(MinResidue);  
ocr2.SetEnableSecondPassGlobalSegmentation(True);
```

Studio

In the **Advanced** tab:

1. Set the **Segmentation Method** to **Global**.
2. **Global Segmentation Method** > ... to select the threshold.
3. Select a threshold mode.
4. Check **Second Pass Segmentation**.



Configure the local segmentation

- Use the parameter **MaxVariation** to define 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.
 - Set this parameter between 0 and 1 (the default setting is 0.25).
 - A low setting allows only very stable blobs.
 - A high setting allows detection of blobs that are more unstable.
- Use the parameter **DetectionDelta** to set the range of grayscale values used to determine the stability of a blob.
 - Set this parameter between 1 and 127 (the default setting is 12).
 - With a low setting, the algorithm is more sensitive to noise.
 - With a high setting, the algorithm is insensitive to blobs that have a low contrast with the background.

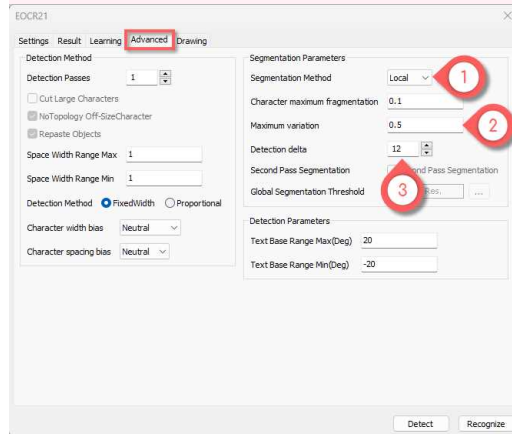
Code

```
ocr2.SetSegmentationMethod(EOCR2SegmentationMethod_Local);
ocr2.SetMaxVariation(0.5);
ocr2.SetDetectionDelta(12);
```

Studio

In the **Advanced** tab:

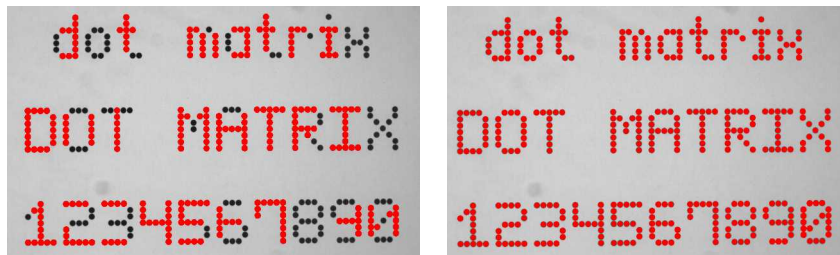
1. Set the **Segmentation Method** to **Local**.
2. Set the **Maximum variation**.
3. Select the **Detection delta**.



Adjust the fragmentation

- Use the parameter `CharsMaxFragmentation` to set how small a blob can be for the segmentation algorithm to consider it as (part of) a character.
 - Set this parameter between 0 and 1 (the default setting is 0.1).
 - The minimum allowed area of a blob is given by:

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



Left: `CharsMaxFragmentation` = 0.1 (default) leads to incomplete segmentation results
 Right: `CharsMaxFragmentation` = 0.01 gives better results

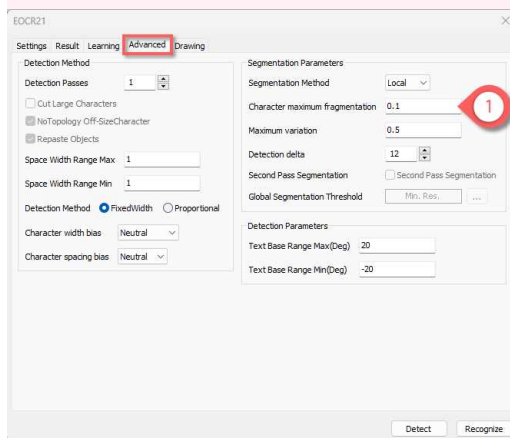
Code

```
ocr2.SetCharsMaxFragmentation(0.1);
```

Studio

In the **Advanced** tab:

1. Set the **Character maximum fragmentation**.



8.15. Detection Parameters

The following parameters are set and used in the process: "Detect the Characters" on page 264.

Select the detection method

- If the topology is required, use the parameter `DetectionMethod` to select the algorithm used for fitting.
 - `EOCR2DetectionMethod_FixedWidth` (default) is optimized for texts using a fixed-width font (including dotted text).

FIXED WIDTH

A fixed-width font, processed with `EOCR2DetectionMethod_FixedWidth`

- `EOCR2DetectionMethod_Proportional` is optimized for texts using a proportional font.

PROPORTIONAL

A proportional font, processed with the `EOCR2DetectionMethod_Proportional`



- With the fixed-width method, all the character boxes have the same width and they do not necessarily fit tightly around the characters.
- With the proportional method, the character boxes fit tightly around the characters. If any character falls outside the range of allowed widths, the detection fails.

Configure the fixed-width font detection

- Use the parameter `RelativeSpacesWidthRange` to specify how wide the spaces between the words may be.

The box-fitting method tests the following range of spaces:

$$\begin{aligned} \min(\text{RelativeSpacesWidthRange}) \times \min(\text{charsWidthRange}) \\ \leq \text{space} \leq \\ \max(\text{RelativeSpacesWidthRange}) \times \max(\text{charsWidthRange}) \end{aligned}$$

NOTE: 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 character boxes.

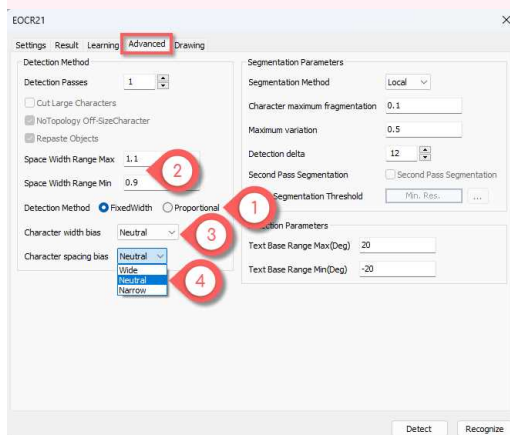
Code

```
ocr2.SetDetectionMethod(EOCR2DetectionMethod_FixedWidth);
ocr2.SetRelativeSpacesWidthRange(0.90f, 1.10f);
ocr2.SetCharsWidthBias(Neutral);
ocr2.SetCharsSpacingBias(Neutral);
```

Studio

In the **Advanced** tab:

- In the **Detection Method**, check **FixedWidth**.
- Enter **Space Width Range Max** and **Space Width Range Min** as float.
- Select the **Character width bias**.
- Select the **Character spacing bias**.



Configure the proportional font detection

- Set the parameter `EnableCutLargeCharacter` if you want that the detection tries to split segmented blobs that are too wide into different characters.

NOTE: This parameter is also used when the topology is not required.

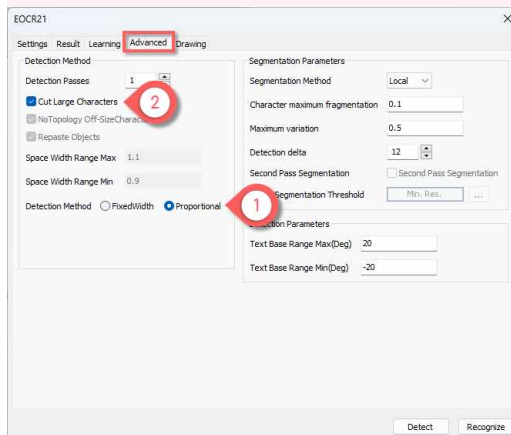
Code

```
ocr2.SetDetectionMethod(EOCR2DetectionMethod_Proportional);
ocr2.SetEnableCutLargeCharacter(true);
```

Studio

In the **Advanced** tab:

1. In the **Detection Method**, check **Proportional**.
2. Check **Cut Large Characters**.



Set the text rotation angle

- Use the parameter `TextAngleRange` to specify how the text in the image may be oriented. The box-fitting method tests the following range of rotation angles:

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

- The angles are defined with respect to the horizontal.
- To set the unit for the angles (degrees, radians, revolutions or grades), use `easy.SetAngleUnit`.
- The default setting is [-20, 20] degrees.

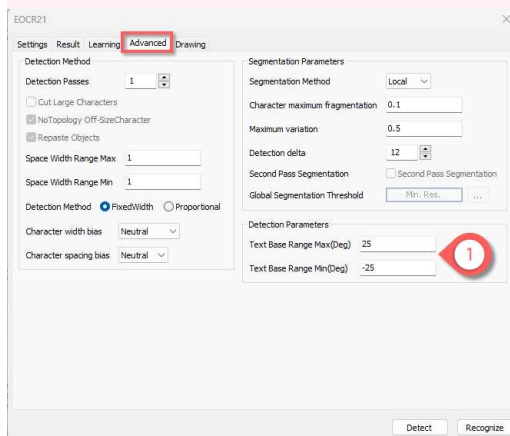
Code

```
ocr2.SetTextAngleRange(-25.00f, 25.00f);
```

Studio

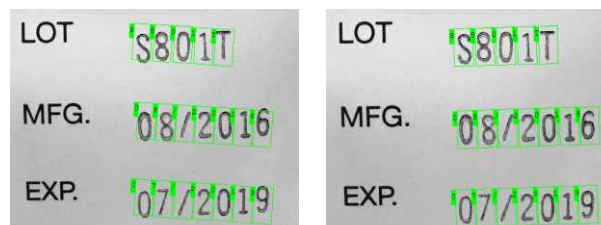
In the **Advanced** tab:

1. Enter **Text Base Range Max(Deg)** and **Text Base Range Min(Deg)** as float.



Set a second detection pass

- Use the parameter `NumDetectionPasses` to configure a second pass for the fitting of textboxes.
 - The first pass fits textboxes to all detected blobs.
 - The second pass selects only the blobs that are covered by the textboxes from the first pass. It fits textboxes to that subset of blobs, possibly resulting in a more optimal fit.
 - Set this parameter to either 1 or 2.
 - The default setting is 1.



Left: `NumDetectionPasses` = 1 and the text angle estimate is slightly off
 Right: `NumDetectionPasses` = 2 and the text angle estimate is better

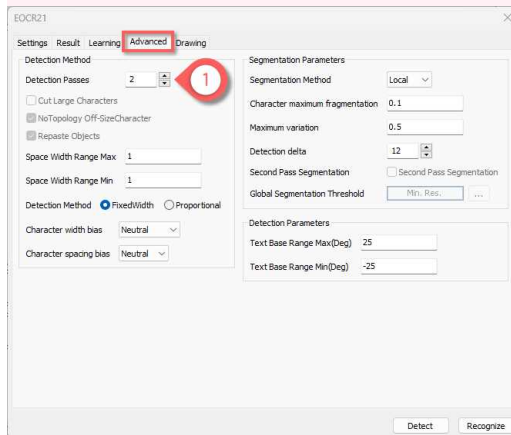
Code

```
ocr2.SetDetectionPasses(2);
```

Studio

In the **Advanced** tab:

1. Set the **Detection Passes** to 1 or 2.



8.16. No Topology Parameters

The following parameters are set and used in the process: "[Detect the Characters](#)" on page 264.

If you do not use a topology, **EasyOCR2** fits boxes to the detected blobs as best as it can.

Detection parameters

You can use the following parameters that are described in "[Detection Parameters](#)" on page 284:

- In the section "Configure the proportional font detection": [EnableCutLargeCharacter](#).
- In the section "Configure the fixed-width font detection": [RelativeSpacesWidthRange](#).
- In the section "Set the text rotation angle": [TextAngleRange](#).

Detect characters that are out of the size range

- Use the parameter [EnableOffSizeCharacter](#) to allow the detection of characters whose size (width and height) is out of the size range but that are in the vicinity of characters in the valid size range.

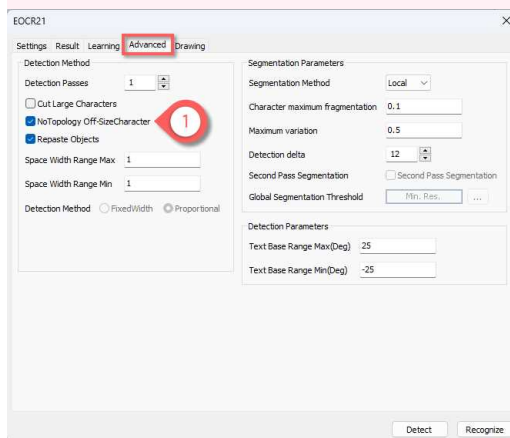
Code

```
ocr2.SetEnableOffSizeCharacter(true);
```

Studio

In the **Advanced** tab:

1. Check **NoTopology Off-SizeCharacter**.



Group blobs of a same character

- Use the parameter **RepasteObjects** to group the blobs that the detection considers as belonging to the same character.
 - This can improve the detection when you have a clean segmentation and the characters are close to each other.
 - The default setting is True and the detection tries to merge the blobs.

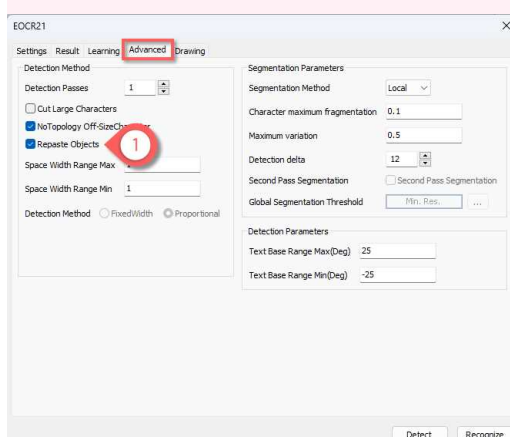
Code

```
ocr2.SetRepasteObjects(true);
```

Studio

In the **Advanced** tab:

1. Check **Repaste Objects**.



9. Code Grading

9.1. What Is Grading?

The "code grading", or "code quality verification", is a process that assesses the quality of a printed or engraved code.

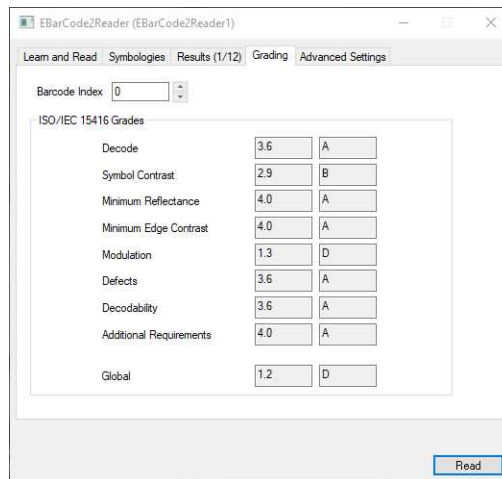
- Different grading standards exist, each pertaining to one or more code types, that specify several grading parameters and how to compute them. In addition to these parameters, a global grade is often computed to summarize the overall quality of the code.
- When a code receives a failing grade, identifying which grade(s) caused the failure can help to locate the physical problem.
- The grades are generally returned:
 - As letters, from A (best) to F (worst).
 - As numbers: from 4.0 (best) to 0.0 (worst).
- While grading is meant to assess the print quality of a code, it also requires specific capture conditions to give accurate results. However, if these conditions are not met, the grading results might give insight on how you can improve your acquisition setup for better results.
- The grading standards currently implemented within **Open eVision** are:
 - **ISO/IEC 15416** for 1D bar codes
 - **ISO/IEC 15415** for data matrix codes and QR codes
 - **ISO/IEC 29158** for data matrix codes and QR codes
 - **SEMI T10-0701** for data matrix codes

9.2. How to Compute the Grading with Open eVision

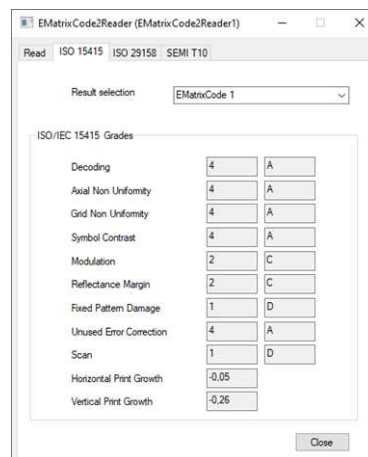
[Read the grades in Open eVision Studio](#)

In **Open eVision Studio**:

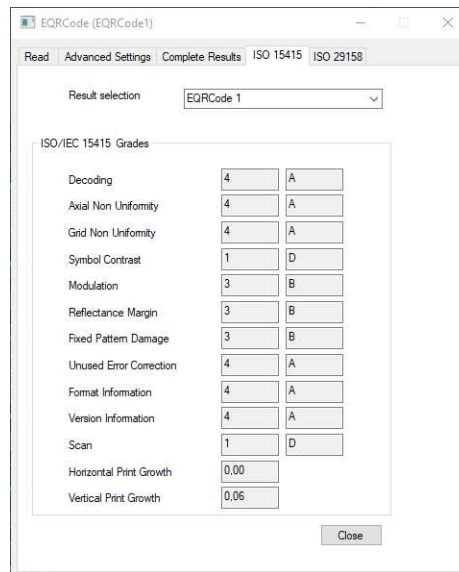
1. Open an image.
 2. Check **Compute Grading** in the first tab of the tool.
 3. Read a bar code, a matrix code or a QR code.
 4. Open the tab(s) named **Grading** or after the standard name.
- The tables list the grades (both as numbers and as letters) and the measurement values.



The ISO 15416 grades in EasyBarCode2



The ISO 15415 grades and values in EasyMatrixCode2



The ISO 15415 grades and values in EasyQRCode

[Compute the grades in the API](#)

In **Open eVision**, each type of code is handled by a specific reader class:

- 1D Barcode: EasyBarCode2::EBarCodeReader
- QR Codes: EQRCODEReader
- Data Matrix Codes: EasyMatrixCode2::EMatrixCodeReader

While the reader classes are different, the way to compute and retrieve the grading results is the same:

1. Instantiate the relevant reader class.
2. Use `SetComputeGrading(true)` to enable the grading computation.
3. For **ISO 29158** only, use `SetIso29158CalibrationParameters` to set up the calibration parameters (see "[ISO/IEC 29158 for Data Matrix and QR Codes](#)" on page 302).
4. Use `Read` on an image containing a code to read and grade the code.
5. Use the corresponding accessor(s) to retrieve the calibration results for the relevant grade(s):
 - For **ISO 15415**: `GetIso15415GradingParameters`
 - For **ISO 15416**: `GetIso15415GradingParameters`
 - For **ISO 29158**: `GetIso29158GradingParameters`
 - For **SEMI T10-0701**: `GetSemiT10GradingParameters`

9.3. ISO/IEC 15416 for 1D Bar Codes

ISO 15416 is a standard that establishes the guidelines on how to assess the print quality of linear (1D) bar code symbols.

It provides a set of quality indicators that give insight on specific areas of the bar code quality. You can use these indicators to compute an overall grade for the inspected bar code.

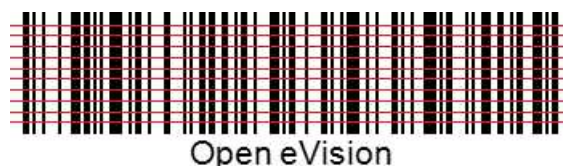


NOTE

In **Open eVision** 23.12, **ISO 15416** grading is only available for the **Ean13**, **Code128** and **Gs1-128** symbologies.

You can find more information on these symbologies on the [ISO website](#) (must be purchased) and on the [GS1 website](#) (for free).

The scan lines



Each grade is computed on 10 horizontal scan lines:

1. A margin of 10% is removed at the top and at the bottom of the bar code.
2. The 10 scan lines are distributed evenly on the remaining bar code.
3. A grade between 4.0 and 0.0 is computed for each line.
4. The global grade is the average of the grades for each line.

The decoding

For each scan line, a global threshold is computed:

1. All pixels above the threshold belong to a space and all pixels below to a bar.
2. **Open eVision** computes the exact bar / space widths with sub-pixel interpolation.
3. **Open eVision** decodes the scan line with the reference decoding algorithm of the symbology.



NOTE

This decoding process is imposed by **ISO 15416**.

Open eVision uses a different and more robust algorithm (based on gradients instead of thresholds) to locate the bars and the spaces of a bar code. Thus, **Open eVision** can recognize and read bar codes that are otherwise not decoded by this simpler **ISO 15416** process (for example when the illumination is not uniform along the bar code).

The grade values

Each grade is represented by a number from 4.0 to 0.0 with 0.1 steps or by letters from A to F.

- In this document, we use the numbers notation.
- **Open eVision** returns integers from 40 to 0 instead of float values from 4.0 to 0.0 to avoid any rounding issue.

Numeric grade	Alphabetic grade
4.0 to 3.5	A
3.4 to 2.5	B
2.4 to 1.5	C
1.4 to 0.5	D
0.4 to 0.0	F

The ISO 15416 quality indicators

The decode grade

- For each scan line, the *decode grade* is set to 4.0 if the decoding succeeds and 0.0 otherwise.

Symbol Contrast grade

- The *symbol contrast grade* indicates the fraction of the total image contrast used by the bar code for each scan line.
- ▶ If the printing contrast is too weak, if the lighting is not bright enough or if the camera exposure time or gain is too small, the grade is low.



The bars and the spaces are too close on the gray scale

The minimum reflectance grade

- For each scan line, the *minimum reflectance grade* is set to 4.0 if the lowest gray value is less than half of the highest gray value and 0.0 otherwise.
- ▶ If the printed barcode or lighting is too bright or if the camera exposure or gain is too high, the grade is low.



The minimum edge contrast grade

- The *minimum edge contrast grade* indicates the smallest contrast between two adjacent bar and space (including the quiet zones). For each scan line, this grade is 4.0 if the minimum edge contrast is at least 15% (38 gray values) and 0.0 otherwise.
- ▶ A dirty background can result in a low grade.



One space is too dark

The modulation grade

- The *modulation grade* indicates the importance of the minimum edge contrast relative to the symbol contrast.
- ▶ A dirty background can result in a low grade.



There is a light defect in the bar

The defects grade

- The *defects grade* indicates the importance of the irregularities found within the elements and the quiet zones.
 - Damaged or dirty bars or spaces can result in a low grade.

The decodability grade

- The *decodability grade* indicates the importance of the differences between the distances measured and expected.
- ▶ If the distances measured are far from those expected, the grade is low.
- Bad measured distances can have several causes:
 - The code is badly printed.
 - The image acquired by the camera is blurry / noisy.
 - The image acquired by the camera has a resolution that is too small.



The bar is 1 px too large and the space is 1 px too thin

The additional requirements grade

- The *additional requirements grade* indicates a requirement that is specific to a symbology.
- ▶ **Code128**, **Gs1_128** and **Ean13** grade the size of the quiet zone of the bar code. If the quiet zone is too small for a line, the associated grade is 0.0.

The global grade

- For each line, the *global grade* is the smallest of all the other grades.
- ▶ If two scan lines yield different decoded strings, the *global grade* is set to 0.0, irrespective of the other scan lines.

9.4. ISO/IEC 15415 for Data Matrix and QR Codes

ISO 15415 is a standard that establishes the guidelines on how to assess the print quality of 2D bar code symbols, either multirow bar codes or two-dimensional matrix symbologies such as data matrix and QR codes.

It provides a set of quality indicators that give insight on specific areas of the bar code quality. You can use these indicators to compute an overall grade for the inspected bar code.

The ISO 15415 as a print quality assessment tool

To be used as a print quality assessment tool as intended, the **ISO 15415** standard requires very specific acquisition conditions.

These conditions might include, but are not limited to:

- A camera perpendicular to the plane of the code to be assessed.
- 4 light sources, placed at the 4 cardinal points around the code and providing specific illumination at a 45° angle.
- An 8-bit gray-scale digitization.
- A 1:1 magnification lens.
- At least 5 pixels per module on the produced image.
- A symbol centered in the image.

For more information about those conditions, please refer to chapter 7 of the standard.

The ISO 15415 as a symbol and/or setup quality assessment tool

Obviously, the conditions above are difficult to meet. However, the **ISO 15415** standard is still useful as an assessment tool of your symbol and/or setup quality even if all those conditions are not met.

In the following paragraphs, we will give you insights on how to use the returned grades to that effect.



NOTE

As **Open eVision** assesses one image at a time, it provides, according to the standard, a scan grade.

To compute an **ISO 15415** overall grade, you need to make a total of 5 acquisitions with different symbol rotations, grade each of these acquisitions using **Open eVision** and then average the results.

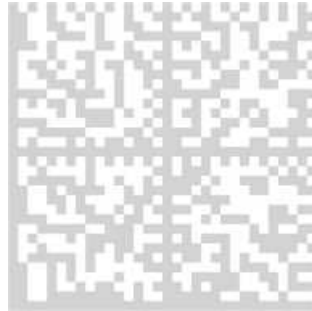
The ISO 15415 quality indicators

The decode grade

- The *decode grade* indicates if the symbol is readable (if it can be correctly decoded) by the symbology reference decoding algorithm.

The symbol contrast grade

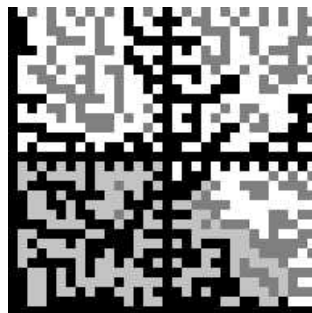
- The *symbol contrast grade* is a measure of the relative difference of reflectance between the brightest and the darkest module in the symbol.
- ▶ If this grade is low, the detection and the digitization are more difficult as it is difficult to separate the code from the background.



A data matrix with a low contrast

The modulation grade

- The *modulation grade* is a measure of the uniformity of the reflectance of the dark and light modules.
- ▶ If this grade is low, the digitization is more difficult, as the variations prevent finding an easy way to separate white from black.



A data matrix with a low modulation grade

The reflectance margin grade

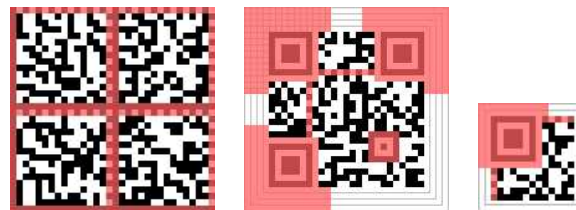
- The *reflectance margin grade* is a measure of how the modules colors are distinguishable relative to the global threshold. This global threshold is the mean reflectance of the brightest and the darkest module.
- ▶ If this grade is low, the digitization is less reliable. As the cells are too close to the separation limit, the light and dark cells can be confused.



A data matrix with a low modulation grade

The fixed pattern damage grade

- The fixed patterns of the symbols characterize the symbols as such. The *fixed pattern damage grade* indicates the likelihood that the symbol is correctly located and identified in the image.



The areas relevant for this grade

The axial nonuniformity grade

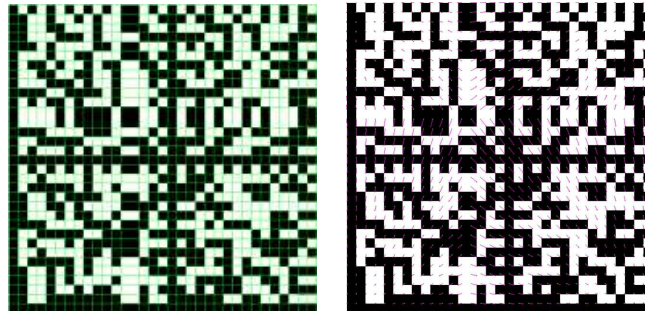
- The *axial nonuniformity grade* is a measure of the isotropy of the principal axis of the symbol relative to its corresponding ideal symbol geometry.
- ▶ Usually, a low grade here only causes issues if a column or a row becomes too small to be accurately sized during the gridding.



A data matrix stretched along its horizontal axis

The grid nonuniformity grade

- The *nonuniformity grade* is a measure of the maximum relative deviation of the intersection of the lines of the estimated grid from to its corresponding intersection in an ideal grid.
- ▶ The lower the grade, the less uniform the shapes of the modules of the symbol are. However, this grade relies on a maximum deviation. It is thus quite unstable and its practical usage is limited.

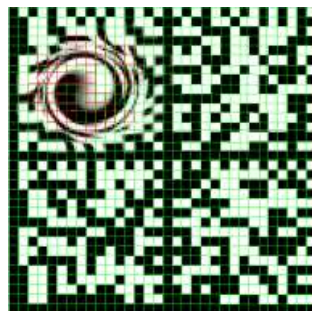


On the left, the grid is computed from a data matrix in which the vertical and horizontal lines comprise one specific module and it had been stretched

On the right, the green dots correspond to the expected positions of the corners of the modules if the grid had been uniform

The unused error correction grade

- The *unused error correction grade* indicates the extent to which the error correction algorithm capacity was used to decode the symbol.
- ▶ A low grade is a sign that any further degradation of the acquisition conditions will lead to a decoding failure.



A data matrix with damaged modules that requires the use of error correction

The format information grade (QR codes and micro QR codes)

- The *format information grade* indicates the readability of the format information blocks.
- ▶ The lower the grade, the more errors must be corrected to recover the error correction level and the masking pattern that are necessary to reliably decode the symbol.



The areas relevant for this grade

The version information grade (QR codes)

- The *version information grade* indicates the readability of the version information blocks.
- ▶ The lower the grade, the more errors must be corrected to recover the version of the symbol.

NOTE: This is relevant only for versions ≥ 7 .



The areas relevant for this grade

The scan grade

- The *scan grade* is the lowest grade value of all the grade indicators. It is a measure of the overall quality of the code in the image.

9.5. ISO/IEC 29158 for Data Matrix and QR Codes

The **ISO 29158** standard is a modification and an extension to the **ISO 15415** standard designed to be more adapted to the grading of DPM (Direct Part Mark) codes.

- It specifies a new acquisition methodology as well as new quality indicators specifically aimed at DPM symbols.
- Compared to **ISO 15415**, the new acquisition methodology simplifies the acquisition setup at the cost of calibrating of this setup and providing the resulting calibration parameters to the grading process.

Calibrating an ISO 29158 setup

To perform the calibration, you must record the image of an "ideal" (perfect) code on the setup.

To have a successful calibration:

1. Set up the Illumination and acquisition parameters (gain, exposure...) so that the *Mean Light* (ML, the mean value of the pixels of the center of the light cells) is in the range considered valid by the standard (70%~86% of the maximum gray level).

NOTE: To compute the *Mean Light* with **Open eVision**, grade the symbol with the default calibration parameters and retrieve it from the structure returned by `GetIso29158CalibrationParameters`.

2. Record the ML as `MLCa1`.

Compute the other calibration parameters:

3. `RCa1`, the calibration reflectance, is the maximum reflectance of the calibration symbol, that is the gray level of the lightest cell.
 - This gray level is computed as the mean gray level on the aperture.
 - The aperture is a circle of a radius 50% or 80% of the cell width and centered on the middle of the cell.
4. `SRCa1`, the calibration system response, represents the parameters used to set the brightness of the image at the calibration time.
 - It can be the gain, the exposure, the global illumination due to the lighting angle or even a combination of those.
 - Its nature does not matter as long as the same measure is used for `SRCa1` and `SRTarget`.

NOTE: `SRTarget` is the same measure as `SRCa1` but taken at the grading time. Since it is part of the calibration parameters, it should be computed at grading time and passed along the other parameters using `SetIso29158CalibrationParameters` just before calling `Read`.

ISO 291528 Quality Indicators

The cell contrast grade

- The *cell contrast grade* is a measure of the relative difference between the mean reflectance of the brightest and of the darkest modules in the symbol.
- ▶ If the grade is low, the digitization is more difficult as it is difficult to separate the dark from the light cells.

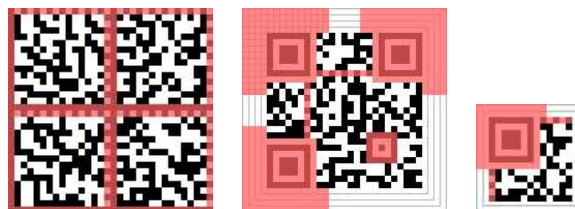
The cell modulation grade

- The *cell modulation grade* is a measure of the variability and the reliability of cell colors. In essence, it is a combination of the *modulation grade* and *reflectance margin grade* of **ISO 15415**. It is thus a composite grade.
- ▶ If this grade is low, the digitization is more difficult, as the variations prevent finding an easy way to separate the white from the black. It is also less reliable, as the cells might be too close from the threshold for a clear dark/light classification.

The fixed pattern damage grade

The fixed patterns of the symbols are what characterize the symbols as such.

- The *fixed pattern damage grade* is a measure of the likelihood that the symbol is correctly located and identified in the image.
- ▶ If this grade is low, the detection and the grid determination are more difficult.
 - Finder pattern damages can prevent the recognition of a candidate as a valid code.
 - Timing pattern damages can prevent the correct computation of the grid.



The areas relevant for this grade

The minimum reflectance grade

- The *minimum reflectance grade* represents the difference between the calibration conditions and the acquisition conditions.
- ▶ It is set to 0.0 if the difference is too big for the grading results to be accurate and 4.0 otherwise.

9.6. SEMI T10-0701 for Data Matrix Codes

SEMI T10-0701 is a grading standard intended to provide quality indicators for DPM (Direct Part Mark) Data Matrix codes. It does not provide guidelines for paper-printed ones.

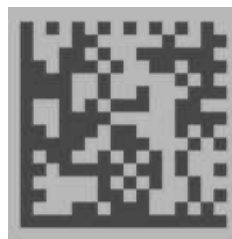
- **SEMI T10-0701** does not provide grades as defined by the other standards. It does not give ratings (A to F or 4.0 to 0.0) but only numerical values.
- The interpretation of these numerical values is left to the application and can depend on the application context.

NOTE: **SEMI T10-0701** is pixel-based, in contrast to the other standards. As such, it can be more suited to computer vision purposes, as the specifications are designed with computer processing in mind.

SEMI T10-0701 quality indicators

The symbol contrast

- The *symbol contrast* measures the relative distinctiveness between the light and the dark parts of the code.
- The closer the *symbol contrast* is to 100%, the more the marks and the spaces are color-separated and the code easier to read.
- A mark is defined as a cell of the data matrix that is modified by the marking process (where the substrate is altered, usually resulting in a darker color) while a space, conversely, is left untouched.
- ▶ A low contrast score can indicate either a bad printing quality, a bad image contrast or an incorrect code illumination. The detection and the digitization are more difficult as it is difficult to separate the code from the background.



A low contrast data matrix

The symbol contrast SNR

- The *symbol contrast SNR* (Signal to Noise Ratio) measures the relative strength of the noise in the code. You can also see it as the ratio of the useful dynamic (symbol contrast) used to separate white from black.
- The higher this value, the less noise is present in the image of the code.

- ▶ If this grade is low, the digitization is more difficult as it is difficult to separate light from dark.



A noisy data matrix

The mark growth

- The *mark growth* is the measure of the relative size of a mark compared to a space.
 - A value around 50% is ideal, as it means that the marks and the spaces are about the same size.
 - A value over 50% means that the marks cells are bigger than the spaces, hinting at overmarking.
 - A value under 50% hints at undermarking.
- The *mark growth* is computed as 2 separate values: the *horizontal mark growth* and the *vertical mark growth*.
- ▶ If the *mark growth* deviates from the ideal value (50%), the digitization is more difficult as some cells eat up part of the neighboring cells.

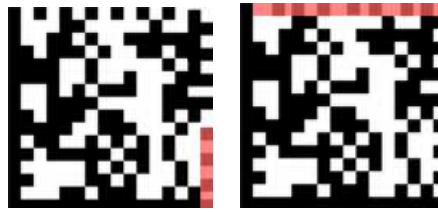


A data matrix showing mark growth

The data matrix mark misplacement

- The *data matrix mark misplacement* is the measure of the relative distance between the ideal position of a mark and its real position in the data matrix. The center of the mark is used as its position.
- The bigger this value, the more the data matrix is potentially deformed.
- The *data matrix mark misplacement* is computed as 2 separate values: The *horizontal data matrix mark misplacement* and the *vertical data matrix mark misplacement*.

- ▶ This grade is just an indicator of the cell position errors. Unless extreme, it should not hinder the processing.



Data matrices showing vertical (left) and horizontal (right) mark misplacement

The cell defects

- If the data matrix can be decoded, the correct placement of the marks and the spaces can be determined, and the *cell defects* value can be computed.
- The *cell defects* value is the measure of the ratio of incorrect pixels (pixels that have the wrong color) to the total number of pixels in the data matrix code.
- ▶ A high *cell defects* value hints at mark damage, space pollution, difficulties to separate the marks from the spaces or that there is a lot of noise in the image. This can hinder the reading of the code.



A data matrix showing cell defects

The finder pattern defects

- The *finder pattern defects* is the same kind of measure as *cell defects* but computed only on the cells of the finder pattern of the data matrix code.
- ▶ A high Finder Pattern Defects value hints at damage to the Finder Pattern, especially if the Cell Defect is otherwise markedly lower. Finder pattern damage might prevent the recognition of a candidate as a valid code.



A data matrix showing finder pattern defects

The unused error correction

- The *unused error correction* is the measure of how much of the error correction capability of the data matrix code was not used.
- The closer this value is to 0, the more error correction was applied to read the code.
- ▶ A low *unused error correction* can hint at errors in the code itself, difficulties to correctly determine the color of a cell or damage in the marks. It is also a sign that any further degradation of the acquisition conditions will lead to a decoding failure.

The data matrix cell size

- The *data matrix cell size* is the mean size, in pixels, of a cell in the data matrix.
- The *data matrix cell size* is computed as 2 separate values: the *data matrix cell width* (horizontal) and the *data matrix cell height* (vertical).
- ▶ You can use these values to compute the code anisotropy.

9.7. Implementation Specifics and Limitations

General limitations of the grading process

- **Open eVision** is an image processing software, and, as such, its inputs are arrays of pixels. Most standards discussed in this document provide guidelines based not on pixels, but on continuous surfaces presenting a continuous reflectance.
- Most standards also impose constraints on lighting and camera placement that **Open eVision** cannot check or enforce.
- All standards also make the hypothesis that the position of the code is perfectly known. That, in practice, is usually not true, especially for 1D bar codes.
- ▶ So, there is no guarantee that **Open eVision** returns exactly the same grade as other software tools (assuming there is no error in those), as choices made to alleviate or circumvent the previous points can differ.
- If you face important differences in grades for the same image or are unsure why some of your images are graded the way they are, feel free to contact **Euresys**' technical support.

Implementation specifics

- Given the pretty strict scope and requirements of the grading standards, it can be difficult, if not impossible, to grade some codes when the acquisition conditions are not perfect.
- ▶ Because of this, **Open eVision** sometimes implements some of the grading processes in a manner slightly different than the canonical one to allow a little more leeway in these conditions.
- These adaptations, however, are made in such a way that if the grading process is made within the requirements of the standard, it yields the required grades as values, as verified by the usage of tools such as conformance calibration test cards.

9.8. References

- **ISO/IEC 15415** - Bar code symbol print quality test specification – Two dimensional symbols: <https://www.iso.org/standard/54716.html>
- **ISO/IEC 15416** - Bar code symbol print quality test specification – Linear symbols: <https://www.iso.org/standard/65577.html>
- **ISO/IEC 15417** - Code 128 bar code symbology specification: <https://www.iso.org/standard/43896.html>
- **ISO/IEC 16022** - Data Matrix bar code symbology specification: <https://www.iso.org/standard/44230.html>
- **ISO/IEC 18004** - QR Code bar code symbology specification: <https://www.iso.org/standard/62021.html>
- **ISO/IEC 29158** - Direct Part Mark (DPM) Quality Guideline: <https://www.iso.org/standard/69411.html>
- **SEMI T10-0701** - Test Method for the Assessment of 2D Data Matrix Direct Mark Quality: <https://store-us.semi.org/products/t01000-semi-t10-test-method-for-the-assessment-of-2d-data-matrix-direct-mark-quality>
- **GS1** General Specifications: <https://www.gs1.org/standards/barcodes-epcrfid-id-keys/gs1-general-specifications>

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.
 - **EasyLocate** has two modes:
 - With **EasyLocate Axis Aligned Bounding Box**, the location of the object is represented by its bounding box.
 - With **EasyLocate Interest Point**, the location of the object is represented by its position only.
 - **EasyLocate** works well with partially occluded objects.
 - You can use it for defect detection, package verification 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

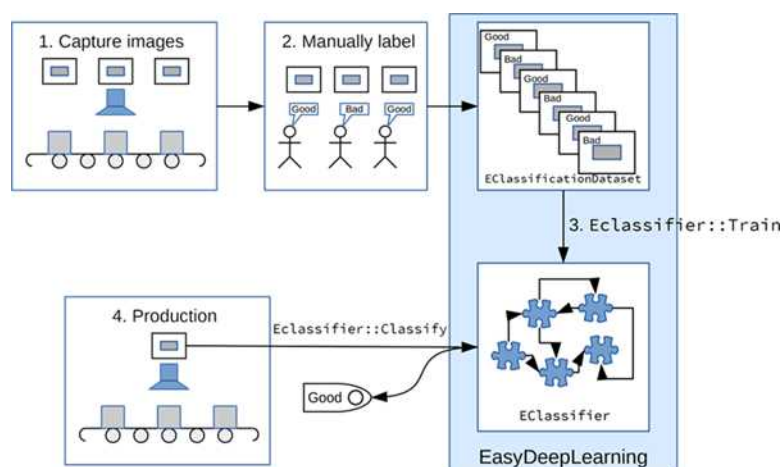
	EasyClassify	EasySegment Unsupervised	EasySegment Supervised	EasyLocate
Minimum image size	128 × 128	64 × 64		128 × 128
Maximum image size	1024 × 1024	10 000 × 10 000		500 000 pixels (for ex. 707 × 707 for square image)
Best image size	256 × 256 - 600 × 600	n.a.		n.a.
Number of channels	1 or 3 (grayscale 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 "[Engines and Hardware Support \(CPU/GPU\)](#)" on page 317 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:**
 - For **EasyLocate Axis Aligned Bounding Box**, you must annotate the defects or the objects with a bounding box and a label.
 - For **EasyLocate Interest Point**, you must annotate the defects or the objects with their position and a label.

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

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

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

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

4. Train the deep learning tool on the dataset.

5. Apply the trained tool in production.

Each tool returns a specific object.

1.2. Deep Learning Studio and Additional Resources

Deep Learning Studio

Deep Learning Studio is a graphical user interface that you can use to create datasets and train your deep learning tools,

Deep Learning Studio is organized around the concept of projects.

A project consists in:

- A tool type.
- A dataset (images and their annotations).
- Several dataset splits.
- The data augmentation settings.
- The tools that you can train on the dataset.
- The results and metrics for each tool to analyze their performance.

The project is stored as a folder containing:

- The project file with the `.edlproject` extension.
- The Images folder with the imported images.
- A dedicated folder for each tool.

By default, the projects are stored in the folder named Euresys Deep Learning Studio Projects located in your Documents folder. After loading a project, the project file is automatically locked when it is used. This means that when a program opens a **Deep Learning** project file, this project file is inaccessible by other programs as long as the initial program doesn't explicitly close the project.

Here are the various file extension used within a project:

- `.edlproject`: the project file containing the dataset, the splits, the list of tools and the data augmentation settings.
- `.edlsettings` (previously `.ecl`): the settings for a tool.
- `.edlmodel` (previously `.ecl`): a deep learning model for the tool.
 - The training model is the model obtained after the last training iteration.
 - The inference model is the model that is applied to images. It is the model that gave the best performances during the training.
- `.edltool` (previously `.ecl`): a complete usable tool (obtained by exporting the tool from **Deep Learning Studio**).
- `.edlmetrics` (previously `.edl`): the metrics for the tools.
- `.edlresult`: a result for a given image and tool.
- `.edldataset` (previously `.eds`): a dataset (obtained by exporting the dataset from the project).



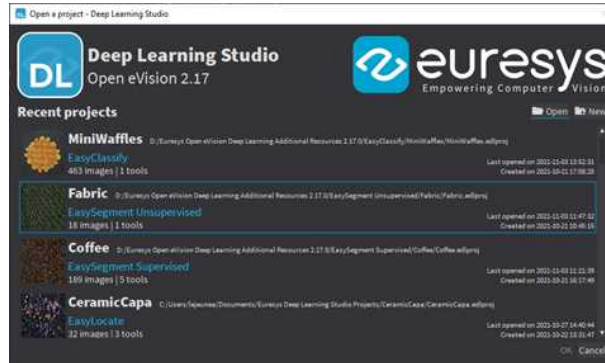
TIP

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

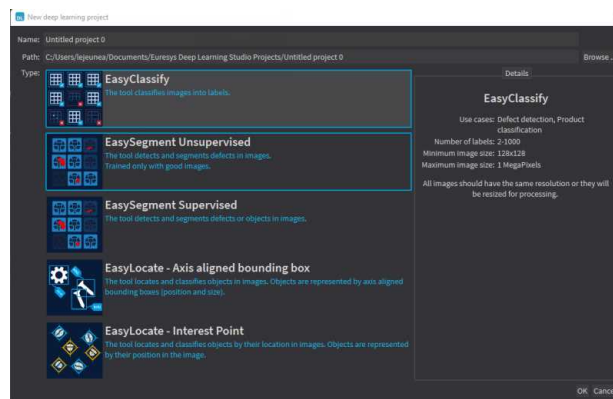
Deep Learning Studio workflow illustration

1. Load or create a project.

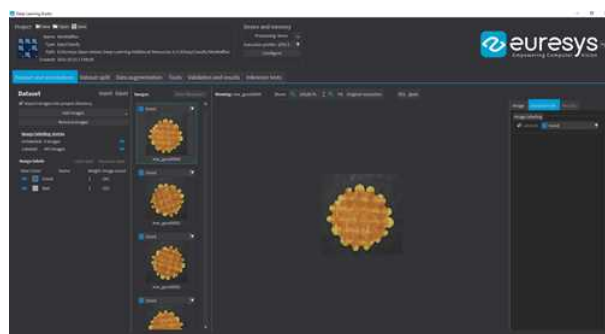
- Quickly go back to your work with the recent project lists.
 - Open existing projects or create new projects.



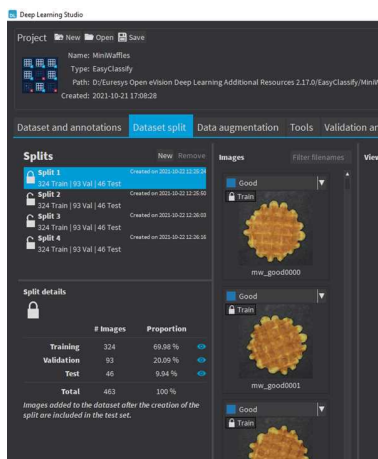
- Create a new project.
 - Specify the name and the type of the project.
 - If necessary, change the path to save the project.



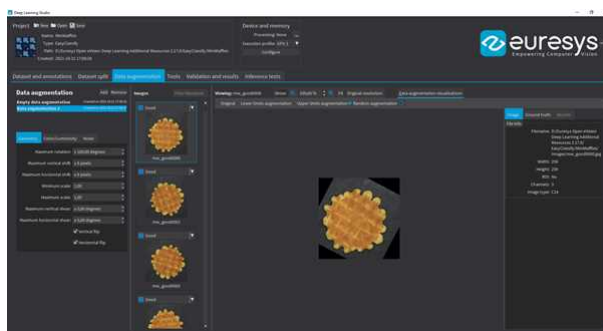
2. Import images in the project dataset and annotate the images.



- Split your dataset into training, validation, and test images (for more details, refer to "Managing the Dataset Splits" on page 334).



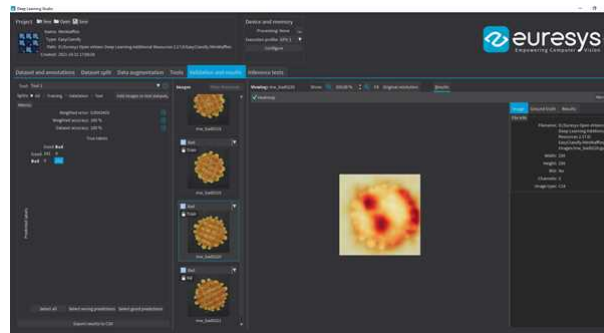
- Configure and view the data augmentation settings (for more details, refer to "Using Data Augmentation" on page 337).



- Create, configure, and train your tools

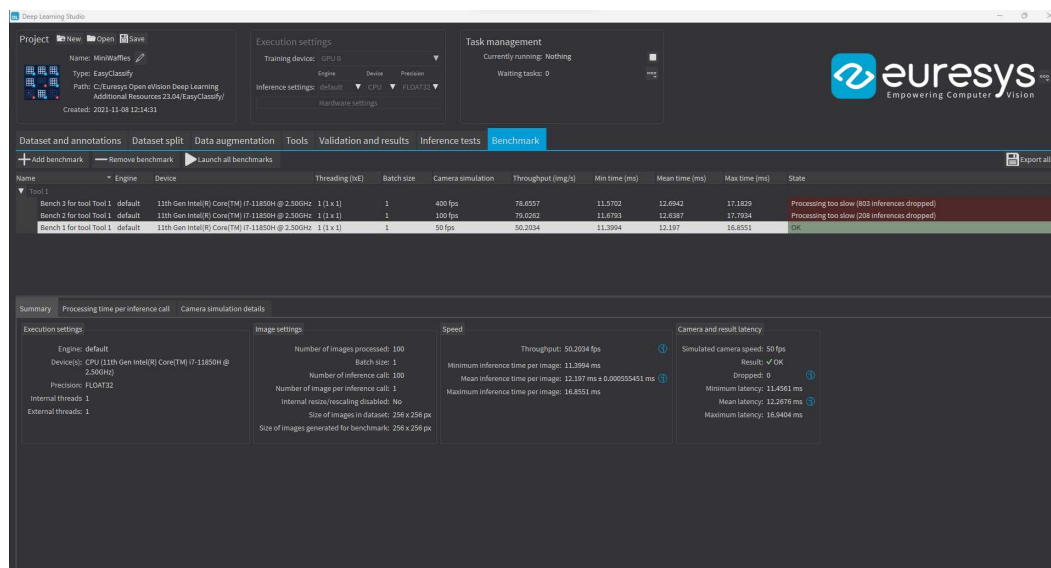


6. Analyze the performance and the results of the trained tools.



7. Benchmark your tools to optimize the execution settings and maximize the inference speed on your hardware.

 See ["Benchmarking a Deep Learning Tool"](#) on page 344.



Resources and code snippets

- The **Deep Learning Additional Resources** package, separate from the **Open eVision** installer, provide several sample datasets as well as **Deep Learning Studio** projects containing a trained tool for these datasets:
 - For **EasyClassify**: the MiniWaffle, Stone Tiles and CopperSilverTape datasets.
 - For **EasySegment Unsupervised**: the Fabric dataset.
 - For **EasySegment Supervised**: the Coffee dataset.
 - For **EasyLocate Axis Aligned Bounding Box**: the ElectronicComponentsBag dataset.
 - For **EasyLocate Interest Point**: the CeramicCapacitor dataset.
- Some sample programs in the folder **Sample Programs** show how to train and use a deep learning tool.
- Some [code snippets](#) are also available for illustration and reference.

1.3. Engines and Hardware Support (CPU/GPU)

Engines

- Starting with **Open eVision 23.04**, running (inference) or training a **Deep Learning** tool is done through an engine. The engines are specified through their names.
 - To set an engine for a tool, use `EDeepLearningTool.Engine`
 - To list the available engines, use `EDeepLearningTool.AvailableEngines`
- An engine can support different devices:
 - To list the detected devices for the current engine, use `EDeepLearningTool.AvailableDevices`
 - To list the detected devices for another engine, use `EDeepLearningTool.GetAvailableDevicesForEngine`. Note that this actually loads the engine in memory.
 - The devices are represented by the class `EDeepLearningDevice` or directly through their name (`EDeepLearningDevice.Name`) that are guaranteed to be unique for a given engine.
 - To set a device, use `EDeepLearningTool.ActiveDevices` or `EDeepLearningTool.ActiveDevicesByName`. You can pass multiple devices for multi-GPU training for example.
 - To know if a device supports inference and/or training, use `EDeepLearningDevice.HasInferenceCapability` and `EDeepLearningDevice.HasTrainingCapability`. Check the matrix below for each engine and device type support.
- A device or an engine can support various inference precision (FLOAT32 or FLOAT16).
 - By default, the precision is FLOAT32.
 - Use `EDeepLearningTool.InferencePrecision` to set or retrieve the current precision.
 - A lower precision (for example FLOAT16 instead of FLOAT32) can improve the speed.
- The main engine is named default:
 - It is always available and always supports the CPU of the computer on which it is running.
 - It always supports both inference and training.
- The other engines:
 - They are mainly used to increase the inference speed.
 - The other engines and the support for a GPU are provided by the package named `deep-learning-redist` (previously `cuda-redist`).

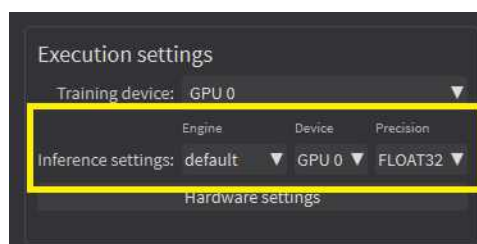
Engine name	Supported devices	Supported platforms	Training	Inference	Inference precision
default	CPU NVIDIA GPU *	All **	Yes	Yes	FLOAT32
EasyDeepLearningEngine_OpenVINO *	CPU Intel GPU	Windows 64-bit	No	Yes	FLOAT16 FLOAT32
EasyDeepLearningEngine_TensorRT *	NVIDIA GPU	Windows 64-bit Linux x64 Linux aarch64	No	Yes	FLOAT16 FLOAT32

* Requires the deep-learning-redist package

** The Windows 32-bit support is only for CPU and the limitation to 2 GB of the application memory can be a problem for the training or the inference on large images.

[In Deep Learning Studio](#)

- In the **Execution settings** panel, select the **Engine**, **Device** and **Precision** that you want to use for inference.



[Legacy API for selecting CPU and GPU](#)

- The API calls `EDeepLearningTool.EnableGPU` and `EDeepLearningTool.GPUIndexes` are deprecated.
- They can only be used with the default engine, otherwise, an exception is thrown.

[Additional consideration for NVIDIA CUDA® GPU](#)



TIP

Using a recent **NVIDIA** GPU greatly accelerates the processing speeds. Refer to the benchmarks for each tool type to compare the GPU and CPU speeds.

To use an **NVIDIA** GPU with the Deep Learning tools, install the deep-learning-redist package for your operating system. It is not recommended to use a system-wide installation of the **CUDA** libraries for GPU support.

- To use an **NVIDIA** GPU with the **Deep Learning** tools, install the deep-learning-redist package for your operating system.

NOTE: It is not recommended to use a system-wide installation of the CUDA libraries for the GPU support.

In this version of **Open eVision**, the GPU acceleration is based on:

- For **Windows** and **Linux Intel x64**:
 - **NVIDIA CUDA® Toolkit** version v11.8
 - **NVIDIA CUDA® Deep Neural Network** library (**cuDNN**) v8.6
- For **Linux ARM (aarch64 Jetson platforms)**:
 - The **CUDA** and **cuDNN** packages distributed with the platforms.

NOTE: The following versions have been tested:

- **JetPack 4.6 (L4T 32.6.1)**
- **JetPack 5.0 (L4T 34.1)**
- **JetPack 5.1.2 (L4T 35.4.1)**

2. Check that you have or install the up-to-date **NVIDIA** drivers.

 Refer to your GPU documentation to install recent drivers for your operating system.

3. For **Linux ARM (aarch64 Jetson platforms)**, install the **NVIDIA JetPack SDK 4.6** minimum that includes the **NVIDIA Jetson Linux Driver Package (L4T) 32.6**.

Starting from **NVIDIA JetPack SDK 4.3**, it is possible to upgrade without flashing the device to the version 32.6 of the **NVIDIA Jetson Linux Driver Package**:

- Edit the file `/etc/apt/sources.list.d/nvidia-l4t-apt-source.list` so that its content is:

```
deb https://repo.download.nvidia.com/jetson/common r32.6 main
deb https://repo.download.nvidia.com/jetson/t194 r32.6 main
```

- Upgrade the system using apt:

```
$ sudo apt update
$ sudo apt upgrade
```

4. Use the engine default for training and inference or the engine `EasyDeepLearningEngine_TensorRT` for inference only.



TIP

For recent **NVIDIA** GPUs or **Jetson Orin** boards (**NVIDIA** GPUs with a compute capability higher than 8.0 on **Intel x64** and 7.2 on **aarch64**), using **Deep Learning** triggers a long initialization (over a minute).

To avoid this initialization each time you use **Deep Learning**, we recommend to increase the **CUDA** cache max size to 1024 MB:

- On Linux, set the environment variable `CUDA_CACHE_MAXSIZE` to `1073741824` before launching **Deep Learning Studio** or your program.

In a terminal, this means executing:

```
$ CUDA_CACHE_MAXSIZE=1073741824 /path/to/your/program
```

- On Windows, launch the [Advanced System settings](#), go to the [Environment variables](#) dialog and add or modify the `CUDA_CACHE_MAXSIZE` variable with the value `1073741824`.

Using multiple GPUs

You can use multiple **NVIDIA** GPUs with the default engine for the training and the batch classification by specifying multiple **NVIDIA** GPUs devices with [EDeepLearningTool.ActiveDevices](#).

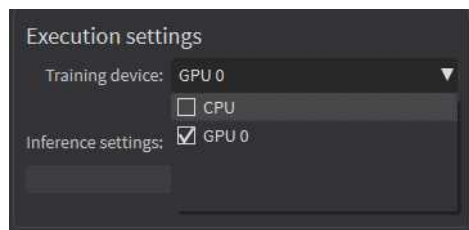
NOTE: Be careful to only set multiple GPUs that have similar performances: the performances are limited by the slowest device.



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 does not yield the same performance gain.

- In **Deep Learning Studio**, to choose the training devices, check all the devices that you want to use for training.



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

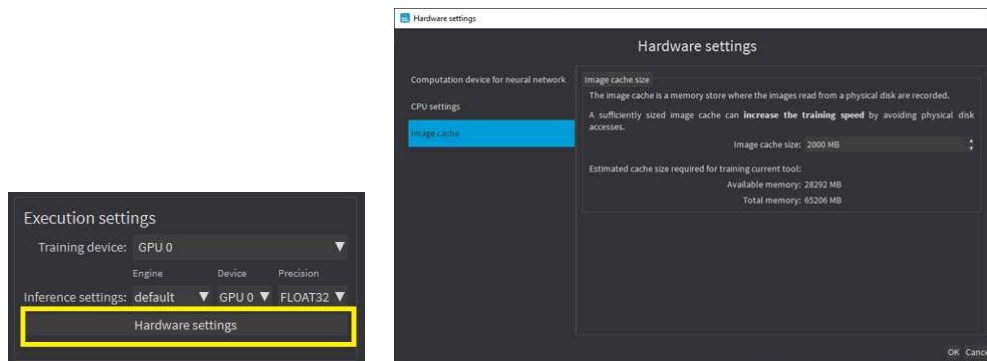
Image cache

The image cache is the part of the memory reserved for storing images during training.

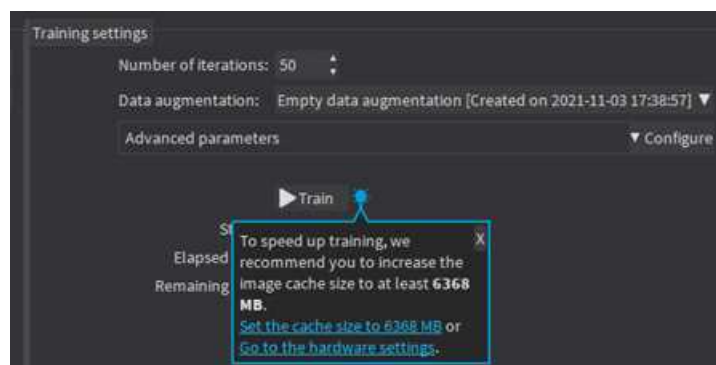
- The default size is 1 GB.
- The training speed is higher when the image cache is big enough to hold all the images of your dataset.
- With dataset too big to fit in the image cache, we recommend using a SSD drive to hold your images and project files as a SSD drive improves the training speed.

To specify the cache size in bytes:

- In the API, use the `EDeepLearningTool::SetImageCacheSize` method.
- In **Deep Learning Studio**, click on the **Hardware settings** button in the **Execution settings** panel and select **Image cache** in the menu.



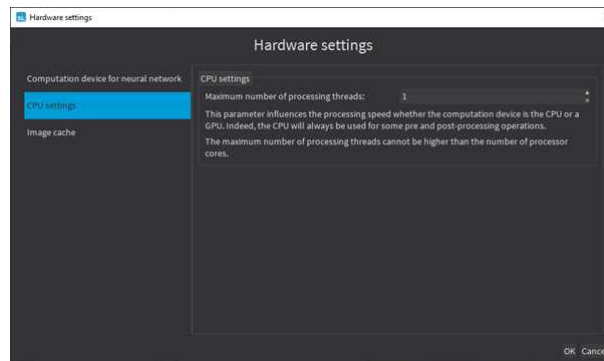
- When there is enough memory to increase the image cache so that it can hold all the images in the dataset, **Deep Learning Studio** displays a recommendation next to the training button.
 - Click on the recommendation to change the image cache size and improve the training speed.



Multicore processing

The deep learning tools support multicore processing with the engines default and EasyDeepLearningEngine_OpenVINO (see "[Multicore Processing](#)" on page 507):

- In the API, use the multicore processing helper function from **Open eVision** (that is `Easy.MaxNumberOfProcessingThreads` 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 Dataset and the Annotations

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_BW16`
 - `EImageType_C24`

- The supported image size depends on the type of the deep learning tool.
 - **EasyClassify** and **EasyLocate** require that all images have the same size. Images that do not have the size configured for the tool are automatically resized before being processed by the neural network.
 - **EasySegment** splits images into patches. As such, the tools can process images of different size and the images are processed at their native resolution.
 - In all cases, your dataset should cover the variability of sizes that you want to process in production.

File formats

The supported standard file formats for the dataset and image annotation are:

- The COCO Json for **EasySegment Supervised** and **EasyLocate Axis Aligned Bounding Box** annotations.
 - A COCO json file contains the annotation for a dataset (several images).
 - In **Deep Learning Studio**, use the **Import** feature to import COCO datasets.
 - 📄 See <https://cocodataset.org/#format-data> for a description of the COCO Json dataset format.
- The YOLO TXT annotation format for **EasyLocate Axis Aligned Bounding Box** annotations.
 - For each image, the annotations are in a file with the same filename as the image and the .txt extension.
 - In **Deep Learning Studio**, if the annotation files are located in the same folder as their corresponding images, use the **Add images** feature to import the images and their annotations.
 - 📄 See <https://pjreddie.com/darknet/yolo> for the original YOLO source code.
- PASCAL VOC XML annotation for **EasyLocate Axis Aligned Bounding Box** annotations.
 - For each image, the annotations are in a file with the same filename as the image and the .xml extension.
 - In **Deep Learning Studio**, if the annotation files are located in the same folder as their corresponding images, use the **Add images** feature to import the images and their annotations.
 - 📄 See <http://host.robots.ox.ac.uk/pascal/VOC> for resources on the PASCAL VOC XML annotations.

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**.

NOTE: **Deep Learning Studio** only displays the labels for the tool type of the project.

- 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.

The image shows three panels from the software interface:

- Image labeling status:** Shows 0 unlabeled and 605 labeled images. Below is a table of image labels:

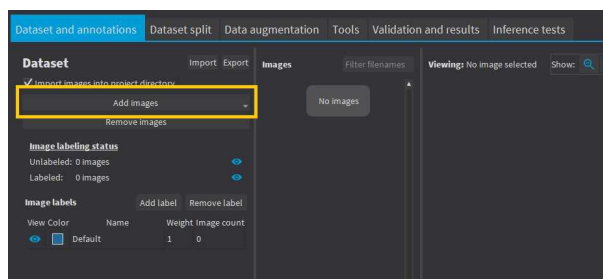
View	Color	Name	Weight	Image count
	■	Good	1	320
	■	Glue	1	106
	■	Damaged	1	83
	■	Broken	1	96
- Image segmentation status:** Shows 0 no segmentation, 189 with segmentation, and 57 good images. Below is a table of segmentation labels:

View	Color	Name	Weight	Image count	Pixel count
	■	Backg...	1	189	433612770
	■	Defect	1	132	1843230
- Object labeling status:** Shows 0 unlabeled and 335 labeled images. Below is a table of object labels:

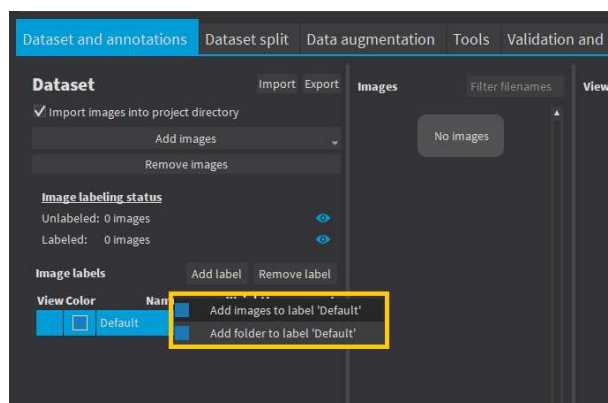
View	Color	Name	Weight	Image count	Object count
	■	Transis...	1	156	320
	■	Diode	1	155	324
	■	C1	1	151	299
	■	C2	1	173	343
	■	C3	1	135	287

Adding Images

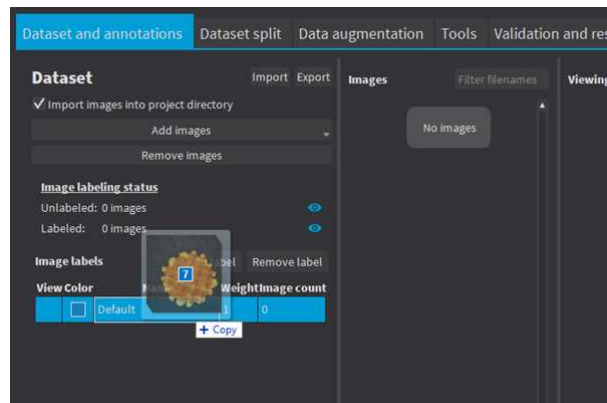
- In **Deep Learning Studio**, add image files (PNG, TIFF, JPEG, BMP and J2K types) to your datasets in one of the following ways:
 - Click on the **Add images** button to add images without any label nor segmentation.



- Right-click on an image label and click **Add images to label** or **Add folder 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.

**TIP**

If there is an `.txt` or `.xml` file with the same filename as the image next to it, **Deep Learning Studio** tries to load these files as YOLO TXT or PASCAL VOC XML annotations.

- 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.
- Import YOLO TXT and PASCAL VOC XML annotations with `EClassificationDataset::ImportYOLOTXTAnnotations(imgId)` or `EClassificationDataset::ImportPascalVOCXMLAnnotations(imgId)`.
 - Check the available annotations for a file with `EClassificationDataset::GetAvailableImageAnnotationFormat(filepath)`.

**TIP**

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

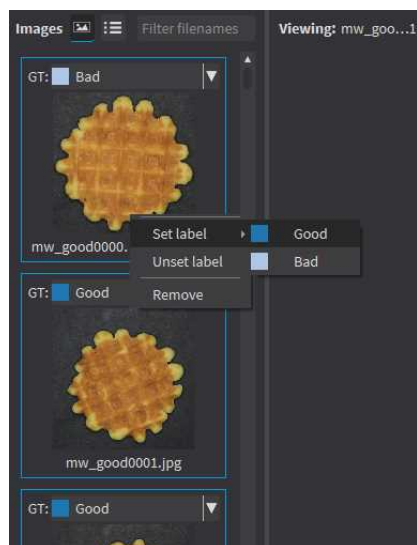
**NOTE**

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

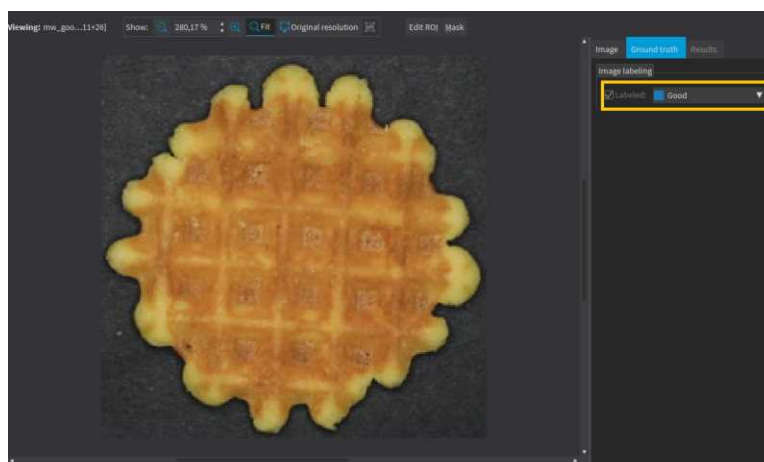
Editing the Label of an Image

In Deep Learning Studio:

- To change the label of images:
 - a. Select one or more images in the image list.
 - b. Right click on the selection.
 - c. Click on **Set label** and select the new label.



- To change the label of a single image:
 - a. Click on the image in the list to open the image in the viewer.
 - b. In the **Ground truth** tab on the right, select the new label.

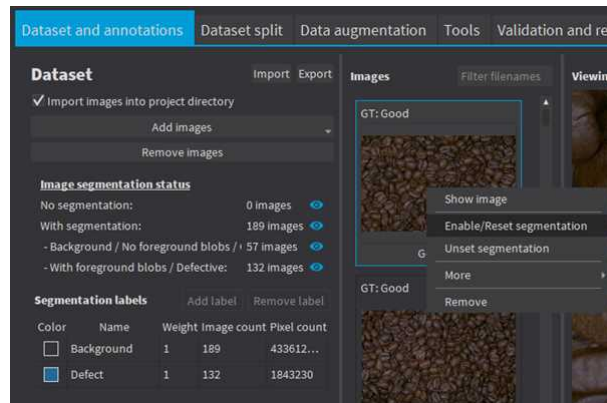


- c. Or use the keyboard keys **F1** ... **F10** to assign the 1st ... 10th label to the image.

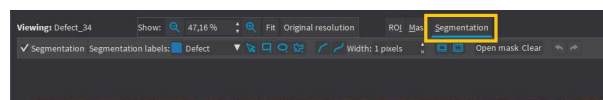
Editing the Segmentation of an Image

In Deep Learning Studio:

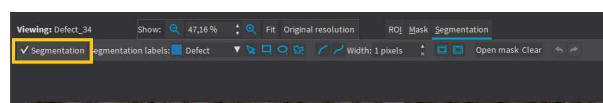
- To initialize or reset the segmentation of an image to all Background pixels:
 - Select one or more images in the image list.



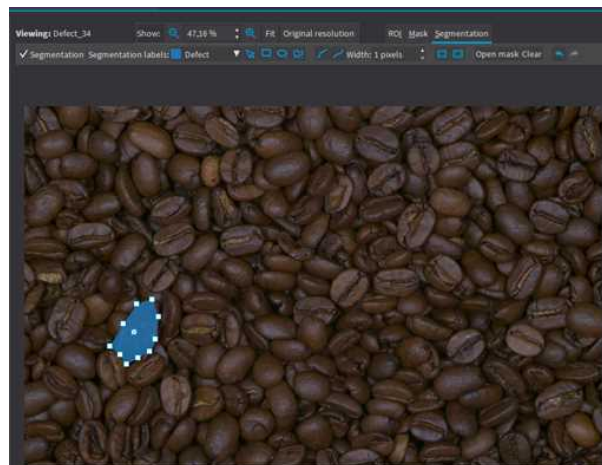
- Right click on the selection.
 - Click on **Enable/Reset segmentation**.
- To edit the segmentation:
 - Double click on the image to open it in the image editor.
 - Click on the **Segmentation** button (**ALT + S**).



- To enable or unset the segmentation, check or 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



- The following drawing tools are available:

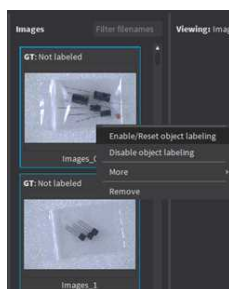
-  **Rectangle**: click and drag to create a rectangle.
 -  **Ellipse**: click and drag to create an ellipse.
 -  **Polygon**: click to create new vertices and double-click to close the polygon. When the polygon is closed, double click on an edge to add a new vertex.
 -  **Assisted segmentation**: click and drag to draw a rectangle around the zone you want to segment. To refine the segmentation, mark the pixels (click and drag) to include in the segmentation.
 -  **Lines**: click and drag to draw a line with a pen of the specified width.
 -  **Free-form drawing**: click and draw any shape with a pen of the specified width.
 -  **Eraser**: same as free-form drawing but it assigns a background label instead of the selected segmentation label.
 -  **Shrink**: shrinks the blobs of the selected segmentation label.
 -  **Grow**: grows the blobs of the selected segmentation label.
- ▶ Except for the eraser tool, all the tools operate on the label specified in the segmentation tool bar.

Editing the Objects of an Image

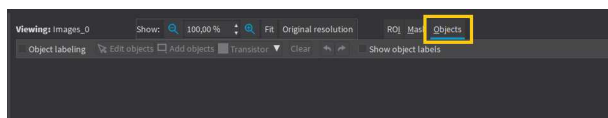
In Deep Learning Studio:

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

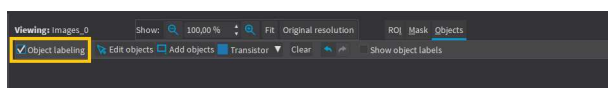
- a. Select one or more images in the image list.



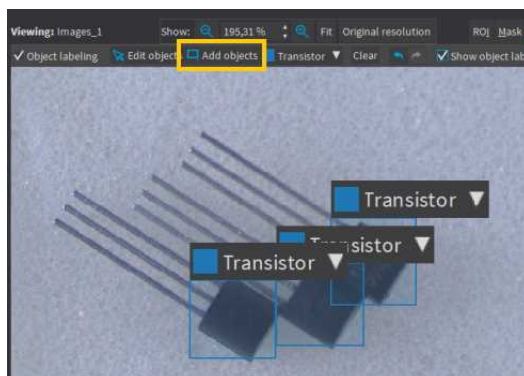
- b. Right click on the selection.
- c. Click on **Reset object labeling**.
- To edit the objects:
 - a. Double click on the image to open it in the image editor.
 - b. Click on the **Objects** button (**ALT + O**).



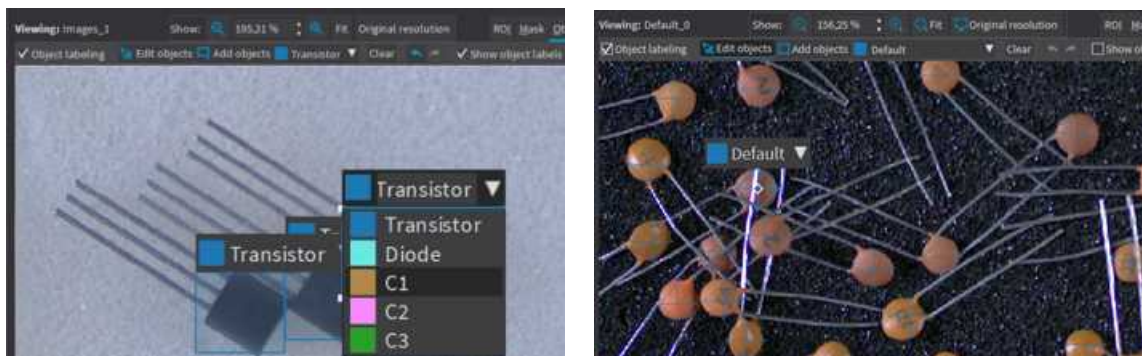
- c. To reset or unset the object labeling, uncheck the **Object labeling** checkbox (**CTRL + L**).



- 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 label of an object and the bounding box or the position, according to the **EasyLocate** mode.



ROI and Mask

Setting a ROI

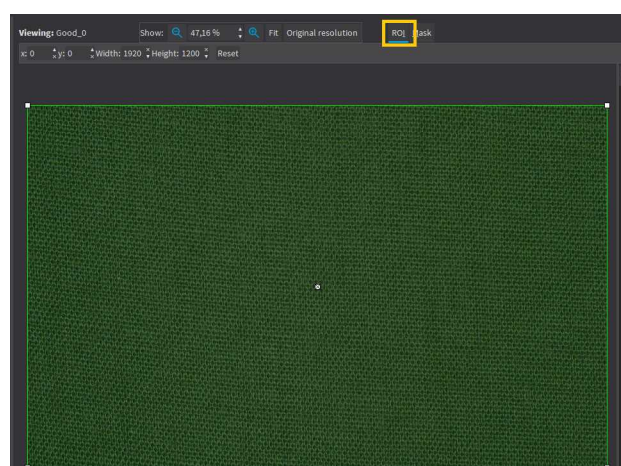
Use an ROI (region of interest) to crop an image or a whole dataset to a rectangular area aligned with the axis.

In the API:

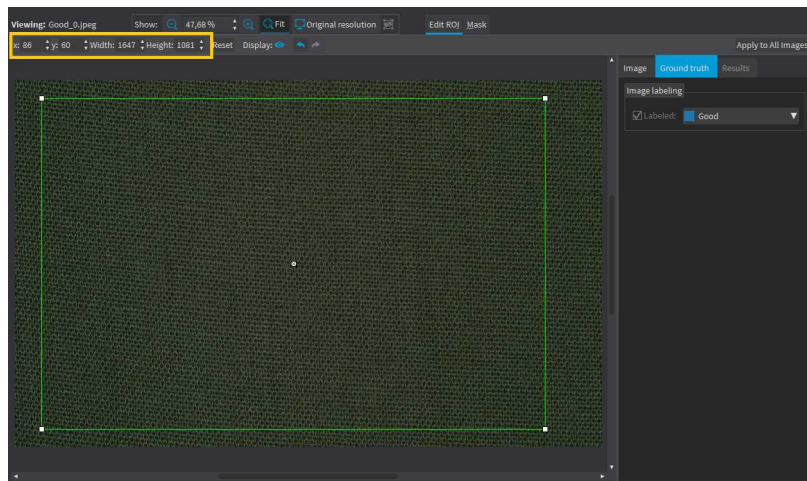
- To define an ROI for an image:
 - Specify the ROI when you add the image to the dataset.
 - Or use `EClassificationDataset::SetRegionOfInterest`.

In Deep Learning Studio:

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



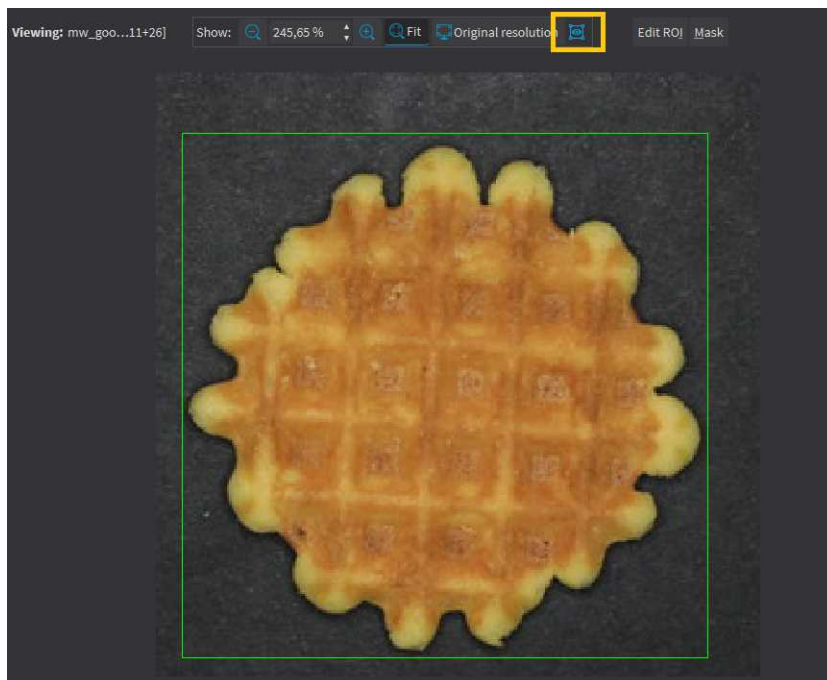
- c. Drag the ROI green box, or directly set the ROI origin (x and y), Width and Height.



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



- To visualize the ROI within its parent image:
 - Click on the button.



Setting a mask

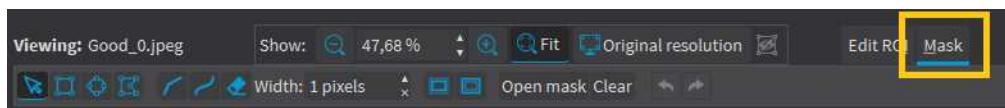
Set a mask on an image in a dataset to remove the pixels in the mask area from any computation. The mask works as a “don’t care area”.

In the API:

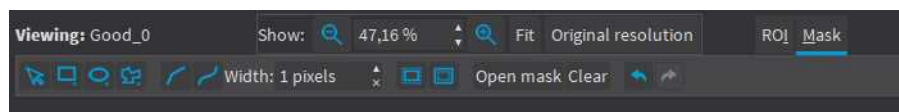
- To define a mask for an image:
 - Specify the mask when you add the image to the dataset.
 - Or use `EClassificationDataset::SetMask`.

In Deep Learning Studio:

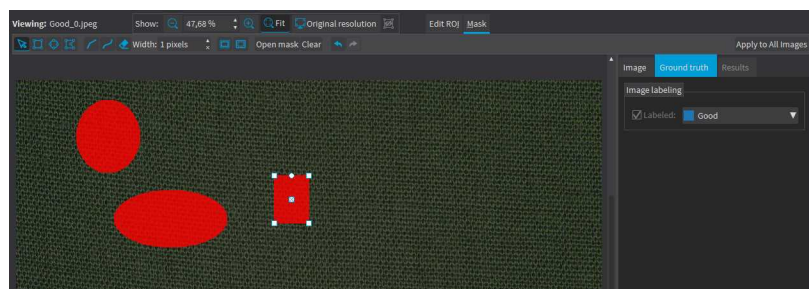
- To change the mask:
 - a. Select an image from the dataset.
 - b. Click on the **Mask** button (**ALT+M**).



- c. Select a drawing tool to draw the mask.



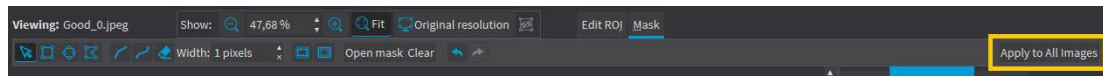
- d. Draw the mask.



TIP

Click on the **Open mask** button to use an image to specify a mask. All the pixels of the image (such as an EROI BW8) 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).

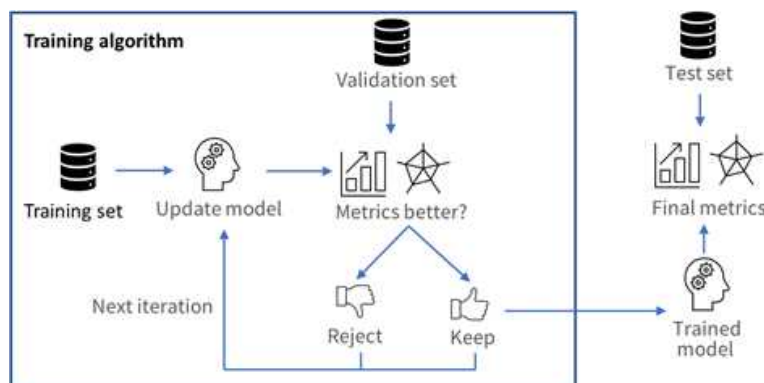


1.5. Managing the Dataset Splits

It is important to split the dataset into 3 sets:

- The *training set* contains the images that are used during training to update and optimize the deep learning model.
- The *validation set* contains the images that are used during training to select the model that gives the best performance.
- The *test set* contains images that are not used during training and that are used to evaluate the final performance of your classifier.

The following picture shows how and when each set is used.



WARNING

These sets MAY NOT contain:

- Images of the other sets.
- Images of an object for which there are other images in other sets.

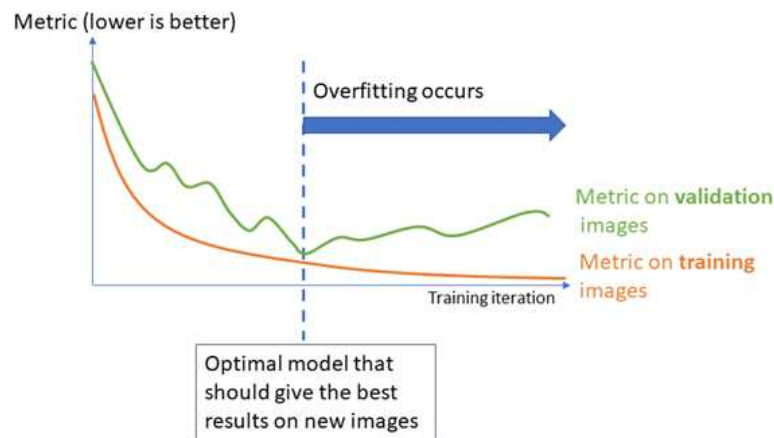
Why is it important?

Deep learning techniques can suffer from overfitting; this means that the trained classifier is too focused on the specific images present in the training set and it is not able to learn a general model of your data. For example, an overfitted model can learn to recognize the exact images present in the training set and not the underlying defects that you want to detect. Such tools perform poorly in production.

The validation set 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 set. Without the validation set, it is impossible to know if a tool that performs well on its training set can also perform well in production.

Thus, a tool that gives high performance on the training set but much lower performance on the validation set has overfitted.

The training algorithm is designed to avoid overfitting by keeping the model at the iteration that gives the highest performance on the validation set.



To minimize overfitting and increase the performances:

- You can add more images to your dataset.
- Or, in some cases, you can use data augmentation.



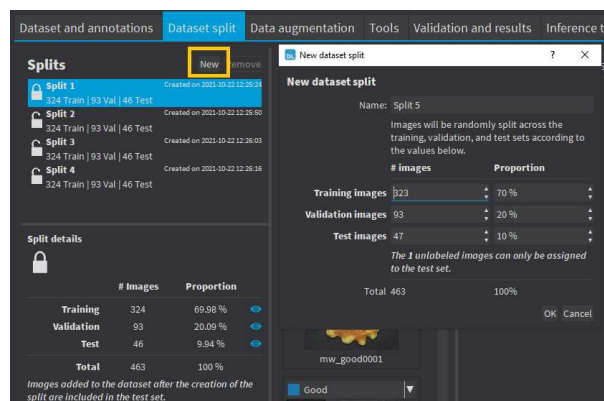
TIP






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.

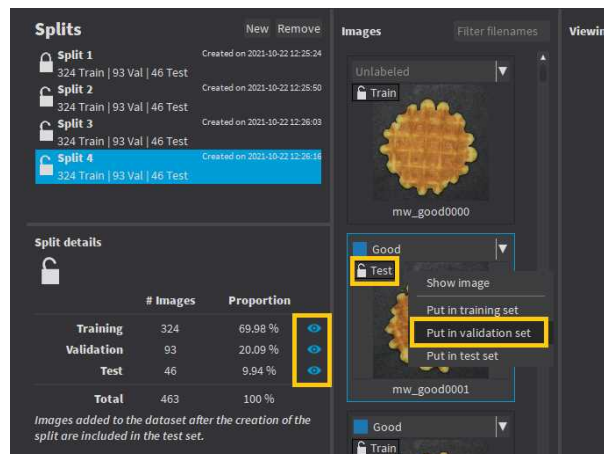
Splitting the dataset in Deep Learning Studio

In **Deep Learning Studio**, open the **Dataset split** tab:

1. To create new dataset splits, click on **New**.
 - A new split is created randomly according to the specified proportion or number of images.
 - The default proportion of images is 70% of training images, 20% of validation images and 10% of test images.
 - Images without a proper annotation must be in the test set.



2. The dataset splits can be locked  or unlocked . A dataset split is locked when it was used to train a tool. When a dataset split is locked:
 - You cannot move images out of the training or validation sets.
 - You can only move images from the test set to either the training set or the validation set.
3. Select a dataset split to display the set of the images.
 - Use the icons  or  to filter the image list according to the set type.
 - If an image is unlocked , you can move it to another split by clicking on its split or by right-clicking on the image and selecting the set.



4. Double-click on a split to change its name.

**NOTE**

Images added to the dataset after you have created a split are automatically assigned to the test set of that split.

**TIP**

We recommend to experiment with several different splits to see how the training behaves with different training sets.

Splitting the dataset in the API

- Create directly `EClassificationDataset` objects for your training, validation and test sets.
- Randomly split an `EClassificationDataset` dataset into a training set and a validation set 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)`
- Or randomly split an `EClassificationDataset` dataset using `EClassificationDataset::GetSplit` to get a `EDatasetSplit` object.
 - Use `EClassificationDataset::ExtractSplit` with the `EDatasetSplit` object to get a `EClassificationDataset` object containing only the images of a given type

1.6. 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).

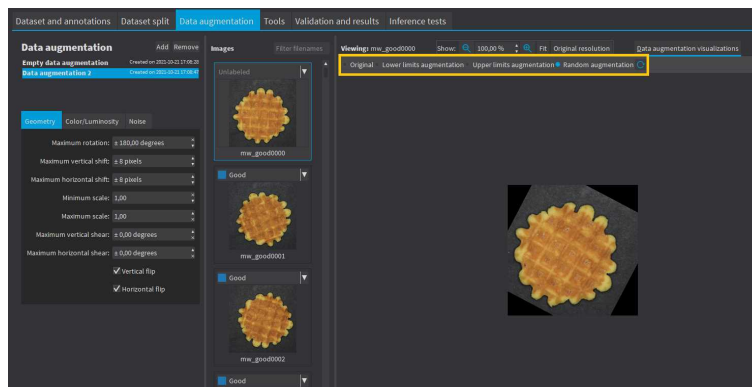


NOTE

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

- Create and configure the data augmentation settings in the **Data augmentation** tab.
- 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**).

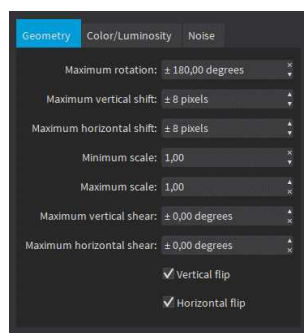


In the API

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations or directly use an object `EDataAugmentation` that you give to the method `EDeepLearningTool::Train`.

The transformations

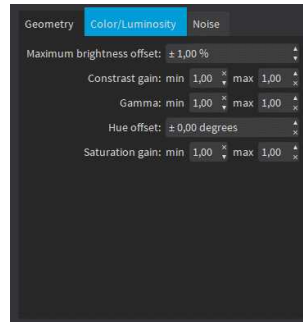
Geometric transformations



- 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)`)

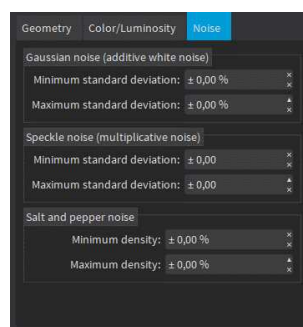
- Rotations (between 0 and a maximum value defined with `EClassificationDataset::SetMaxRotationAngle`)
- Horizontal and vertical shear (between $-\text{maxValue}$ and maxValue defined with `EClassificationDataset::SetMaxHorizontalShear` and `EClassificationDataset::SetMaxVerticalShear`)

Color and luminosity transformations



- Brightness offset (between $-\text{maxValue}$ 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 $-\text{maxValue}$ and maxValue defined with `EClassificationDataset::SetMaxHueOffset`)
- Saturation gain (between a minimum and maximum value defined with `EClassificationDataset::SetMinSaturationGain` and `EClassificationDataset::SetMaxSaturationGain`)

Noise transformations





TIP

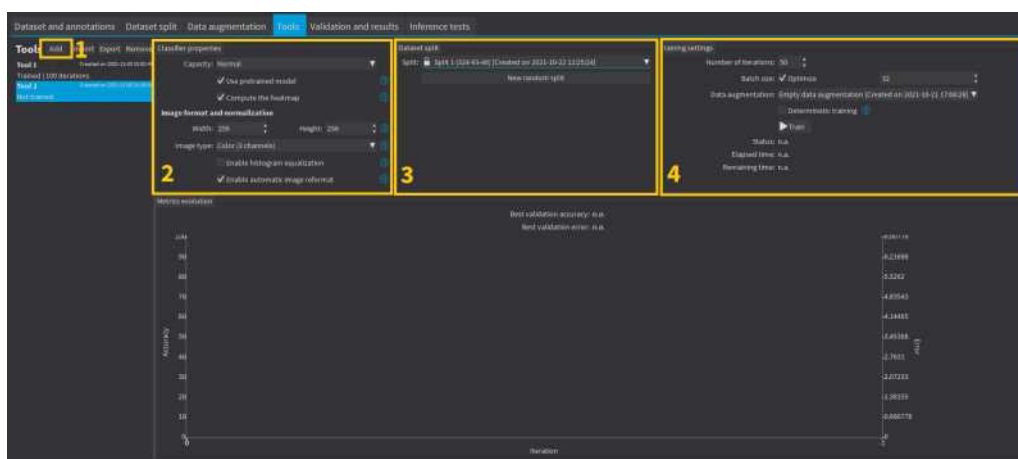
The standard deviation is expressed as a percentage of the maximum pixel value.

- 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`).

1.7. Training a Deep Learning Tool

In Deep Learning Studio

1. Create a new tool.
2. Configure the tool settings.
3. Select the dataset split to use for this tool.
4. Configure the training settings and click on `Train`.



In the API

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

The training settings

- The **Number of iterations**. An *Iteration* corresponds to going through all the images in the training set once.
 - The training process requires a large number of iterations to obtain good results.
 - The larger the number of iterations, the longer the training is and the better the results you obtain.
 - The default number of iterations is 50.

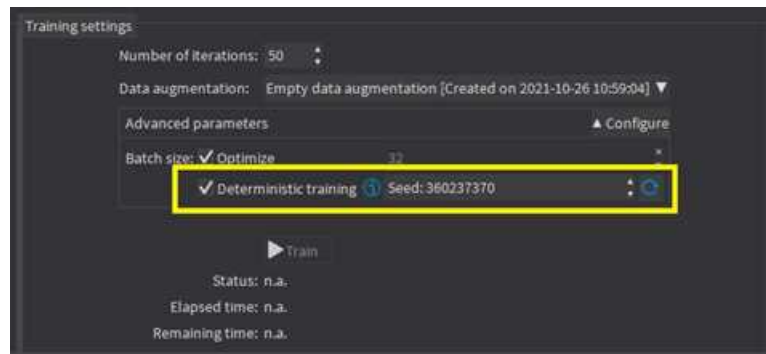


TIP

We recommend to use more iterations than the default value:

- For **EasyClassify** without pretraining, **EasyLocate** and **EasySegment**.
 - For smaller dataset because the training is harder (for example, 100 iterations for a dataset with 100 images, 200 iterations for a dataset with 50 images, 400 iterations for a dataset with 25 images...).
- 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.
 - The *Data augmentation*.
 - Whether to use *Deterministic training* or not.
 - The deterministic training allows to reproduce the exact same results when all the settings are the same (tool settings, dataset split and training settings).
 - The deterministic training fixes random seeds used in the training algorithm and uses deterministic algorithms.
 - The deterministic training is usually slower than a non-deterministic training.

- In **Deep Learning Studio**, the option to use deterministic training and the random seed are available in the advanced parameters.



- In the API, use `EDeepLearningTool::EnableDeterministicTraining.` and `EDeepLearningTool::DeterministicTrainingRandomSeed.`

Continue the training

You can continue to train a tool that is already trained.

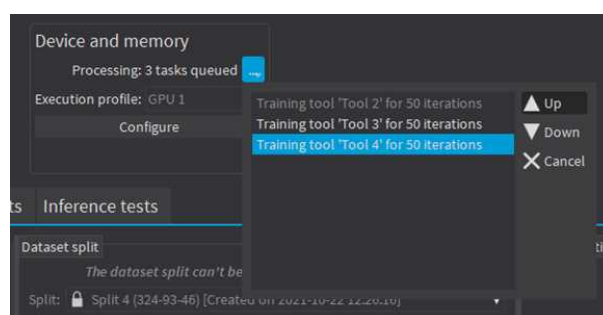
In **Deep Learning Studio**, the dataset split associated with a trained tool is locked.

- You can only continue training a tool with the same dataset split.
- You can still add new training or validation images to the split by moving test images to the training set or the validation set of that split.

Asynchronous training

The training process is *asynchronous* and performed in the background.

- In **Deep Learning Studio**:
 - The training processes are queued.
 - They are automatically executed one after the other.
 - You can manually reorder the training in the processing queue.



- In the API:
 - `EDeepLearningTool::Train` launches a new thread that does the training in the 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.

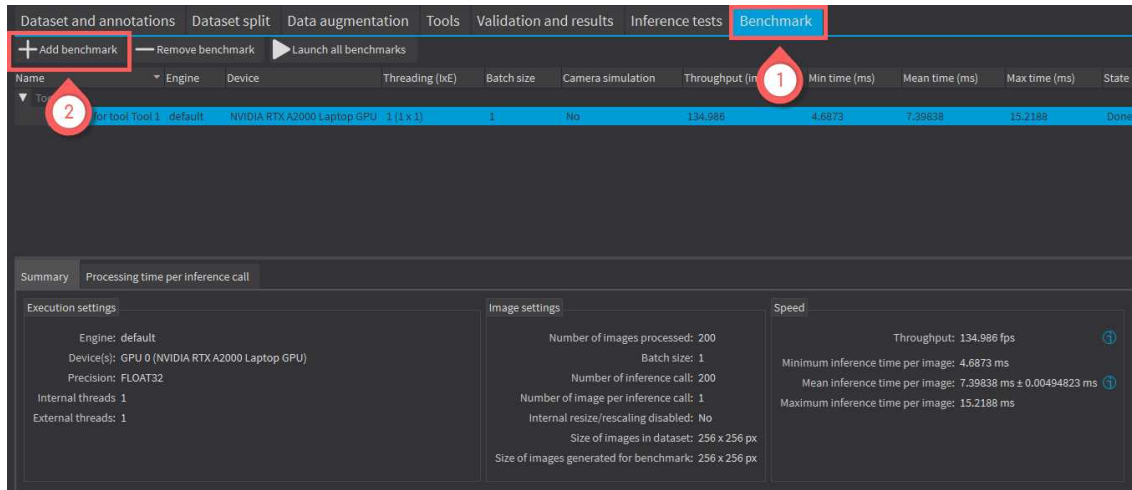
1.8. Using a Deep Learning Tool

In the API

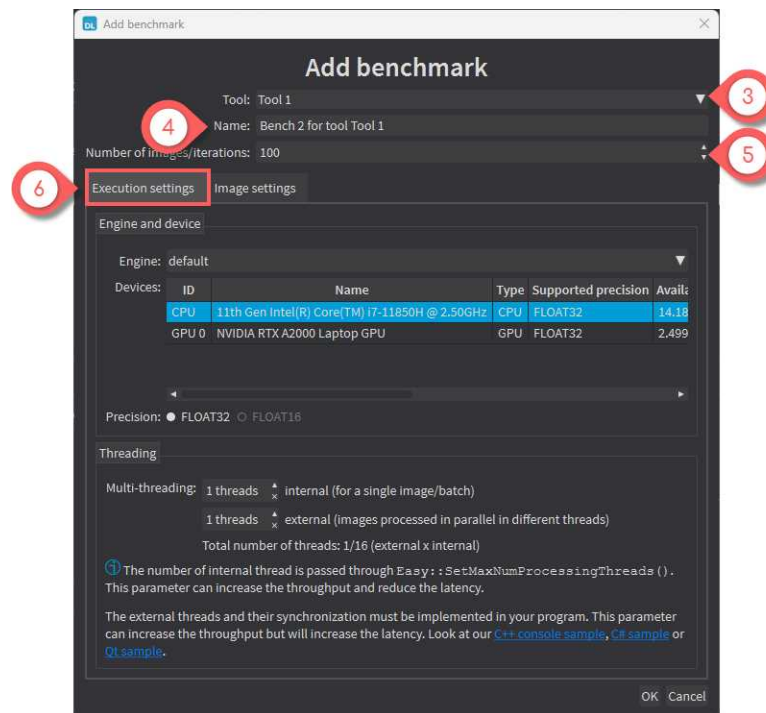
- The API to use a **Deep Learning** tool on a new image or on a set of images is different for each tool:
 - **EasyClassify** > "Classifying New Images" on page 355
 - **EasySegment** > Unsupervised > "Applying the Tool to New Images" on page 369
 - **EasySegment** > Supervised > "Using the Supervised Segmenter" on page 375
 - **EasyLocate** > "Locating Objects" on page 391
 - You can apply each tool to individual images or to sets of images.
 - With a set of images and with GPU processing enabled, the tool processes `EDeepLearningTool::BatchSize` at the same time.
 - **EasySegment** can automatically split individual images in sub-images according to the patch size, the scaling and the sampling density parameters. These sub-images are processed by batch.
- NOTE:** The processing of the first prediction after loading a model, changing the batch size or with a different number of images (smaller than the batch size), is slower than the previous one because the tool must perform various initializations that can be very slow.
- Use the methods `EDeepLearningTool::InitializeInference` with an image or a set of images to perform these initializations before actually using the tool.

1.9. Benchmarking a Deep Learning Tool

In **Deep Learning Studio**, if you have a **Deep Learning** license, you can benchmark your trained tools to measure their speed according to various execution parameters.



1. Open the **Benchmark** tab.
2. To add a benchmark, click on the **Add benchmark** button.



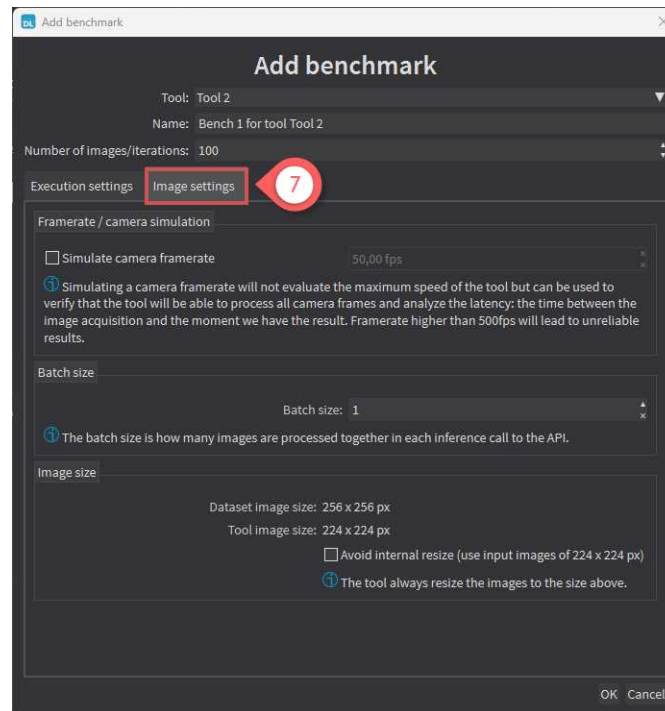
In the **Add benchmark** window:

3. Select the **Tool** that you want to benchmark. Only trained tool appear in the list.
4. **Name** your benchmark.
5. Set the **Number of images/iterations** to perform in the benchmark. The more images, the better the precision.
6. In the **Execution settings** tab, set the parameters that control how the neural network runs:
 - The engine, device and inference precision.
 - The threading parameters.



Use the threading parameters when running the neural network on the CPU:

- The internal threading can improve the processing speed for a single image. It also improves both the throughput (number of images processed by seconds) and the latency (time taken to process one image).
- The external threading involves using the same tools in different threads on different images. This can greatly improve the throughput but it decreases the latency (it means that the number of images processed each seconds increases but the time to process a single image is longer on average).
- Check the code samples referenced in the dialog for how to implement internal and external multithreading.



7. In the **Image settings** tab, set the parameters that control how the images are preprocessed and given to the neural network:
- **Simulate camera framerate**: When checked, the benchmark does not try to perform inference at the maximum speed. Instead, images are produced at the specified frame rate.
Use this setting to check if your execution settings can match the speed of your camera. Simulating a camera frame rate also gives you an estimation of the time spent between the acquisition of an image and the moment the result is available.
 - **Batch size**: Increasing the batch size can greatly improve the throughput, especially for GPU processing. However, it also increases the latency.
 - **Image size**: By default, when applying a tool on an image that has a size different from the input size of the tool, the image is automatically resized by the tool. The time spent performing this resizing is part of the inference time.
Check **Avoid internal resize** to avoid the resizing performed during inference and slightly accelerate the tool.
In a real application, you can configure your camera to capture images directly of the input size of the tool to avoid this automatic resizing step.

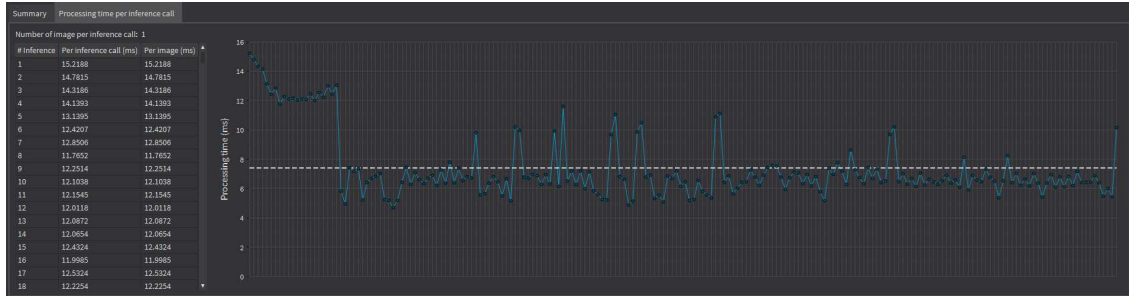
Benchmark results

The following benchmark results are available:

- In the **Summary** tab:
 - The settings of the benchmark.
 - The throughput (number of images processed by seconds).
 - The minimum/ mean / maximum inference time per image.

- In the **Processing time per inference call**:
 - A table and a graph with the processing times.

NOTE: An inference call means calling the inference API with a batch of images (see batch size setting).



- In addition in the **Summary** tab, if you are simulating a camera frame rate:

Camera and result latency

Simulated camera speed: 50 fps

Result: ✔ OK

Dropped: 0 ⓘ

Minimum latency: 7.0608 ms

Mean latency: 8.92305 ms ⓘ

Maximum latency: 11.7943 ms

- Whether all images could be processed in time.
- The number of images that were dropped because the frame rate was too fast compared to the inference speed.
- The minimum/ mean / maximum latency (the time between the acquisition of the image and the moment the result for this image is available).
- The camera simulation details tab that contains a table showing the various timings for each image (acquisition time, inference start time, inference end / latency, inference time).

Summary		Processing time per inference call		Camera simulation details	
# Image	Acquisition	# Inference	Inference start	Inference end / Latency	Inference time
1	0 ms	1	+ 0.0341 ms	+ 7.3899 ms	7.3558 ms
2	20.6381 ...	2	+ 0.1175 ms	+ 9.9168 ms	9.7993 ms
3	40.0852 ...	3	+ 0.1584 ms	+ 9.8796 ms	9.7212 ms
4	60.1939 ...	4	+ 0.1463 ms	+ 9.8619 ms	9.7156 ms
5	79.2444 ...	5	+ 0.0438 ms	+ 7.131 ms	7.0872 ms
6	99.9194 ...	6	+ 0.1281 ms	+ 9.504 ms	9.3759 ms
7	120.171 ...	7	+ 0.1129 ms	+ 7.7202 ms	7.6073 ms
8	139.956 ...	8	+ 0.1469 ms	+ 7.538 ms	7.3911 ms
9	159.66 ms	9	+ 0.1283 ms	+ 7.5549 ms	7.4266 ms
10	180.554 ...	10	+ 0.1316 ms	+ 9.1361 ms	9.0045 ms
11	199.644 ...	11	+ 0.1032 ms	+ 10.6869 ms	10.5837 ms
12	220.045 ...	12	+ 0.0891 ms	+ 10.921 ms	10.8319 ms
13	239.628 ...	13	+ 0.136 ms	+ 10.7854 ms	10.6494 ms
14	259.544 ...	14	+ 0.124 ms	+ 9.5091 ms	9.3851 ms
15	280.265 ...	15	+ 0.0368 ms	+ 7.0699 ms	7.0331 ms
16	299.579 ...	16	+ 0.1138 ms	+ 7.9086 ms	7.7948 ms
17	320.003 ...	17	+ 0.1234 ms	+ 9.3457 ms	9.2223 ms
18	339.995 ...	18	+ 0.1296 ms	+ 9.4134 ms	9.2838 ms
19	359.35 ms	19	+ 0.1235 ms	+ 8.8115 ms	8.688 ms

2. EasyClassify - Classifying Images

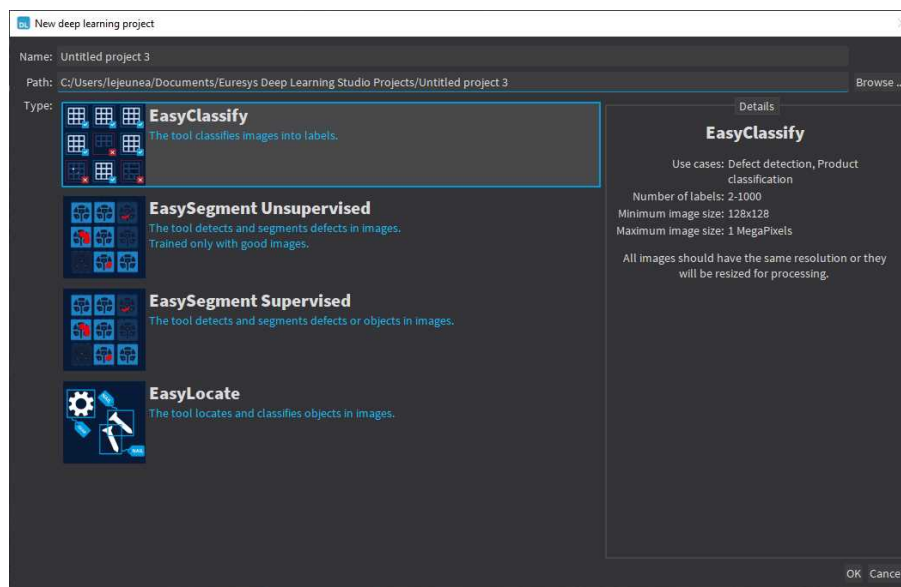
2.1. Tool and Images

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

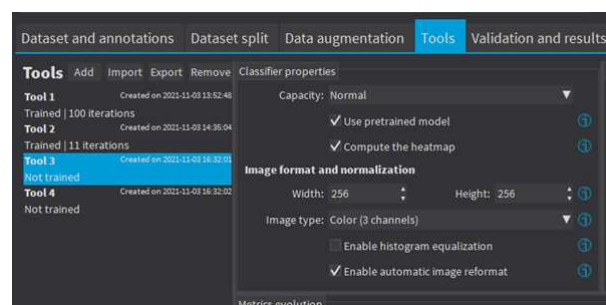
Deep Learning Studio

To create an **EasyClassify** project in **Deep Learning Studio**:

1. Start **Deep Learning Studio**.
2. Create a new project and select **EasyClassify** in the **New deep learning project** dialog.



Tool properties



Capacity

- The **Capacity** of the neural network (default: Normal) represents the quantity of information that it is capable of learning.
 - We recommend you to first train a Normal network on your dataset.
 - ▶ If the learning is working:
 - Try to use the Small or the Large network to get the performance (inference speed and accuracy) that best matches your application.
 - ▶ If the learning does not work it means either that:
 - Your dataset image resolution is too small (below 128×128): try to use the Extra Small network that specifically targets such cases.
 - Your dataset is complex and difficult: try to use the Extra Large network that specifically targets such cases.
- In the API:
 - Use the string parameter `EClassifier::ModelType` to select the model capacity.
 - Use `EClassifier::SetModelType` to set the capacity of your tool.
 - Use `EClassifier::GetAvailableModelTypes` to list the available models for training.

Use pretrained model

- A pretrained model should give faster convergence time and better accuracy for all the datasets except for extremely large datasets (> 10 000 images).
- A pretrained model works by initializing the neural network with the weights learned on a large and complex dataset.
 - This allows to transfer the knowledge of some universal features (such as edges, corners...) learned on this dataset to your dataset.
 - If you do not use a pretrained model, the neural network is initialized randomly.
- In the API, use `EClassifier::SetUsePretrainedModel(false)` to disable the user of pretrained models.

NOTE: You cannot use the histogram equalization with pretrained weights as it would lead to poor training results.

Compute the heatmap

- If this option is checked, the tool computes the heatmap along with each result.
 - The heatmap is available with `EClassificationResult::GetHeatmap` or with `EClassificationResult::GetHeatmap`.
 - ▶ Note that this option makes the computation of the results a bit slower but twice faster than computing the result and then computing the heatmap separately.
- In the API, use `EClassifier::SetComputeHeatmapWithResult` to enable or disable this option.
- If this option is disabled, use the method `EClassifier::GetHeatmap` to get the heatmap for an image.

Image format and normalization

Width, Height, Image type and Enable automatic image reformat

- 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.



TIP

It is recommended to use a width and height of at least 224×224 , as the performances may start to deteriorate if the height or/and the width are smaller.

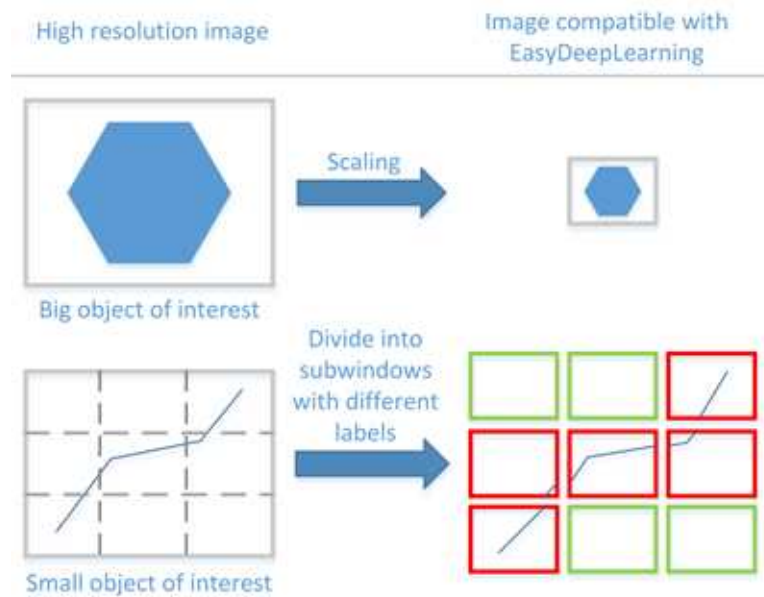
- In **Deep Learning Studio**, you can set the input image format in the **Classifier properties** of the tools that you create in the **Tools** tab.
- In the API, you can also set manually the input image format with the methods `SetWidth`, `SetHeight` and `SetChannels` (1 channel for grayscale images and 3 channels for color images).

Image resolution

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.



Enable histogram equalization

The classifier can also apply an histogram equalization to every input image:

- In **Deep Learning Studio**, activate it in the image format controls in the **Image properties and augmentation** tab.
- In the API, use `EClassifier::SetEnableHistogramEqualization(true)` to activate it.

NOTE: You cannot use the histogram equalization with pretrained weights as it would lead to poor training results.

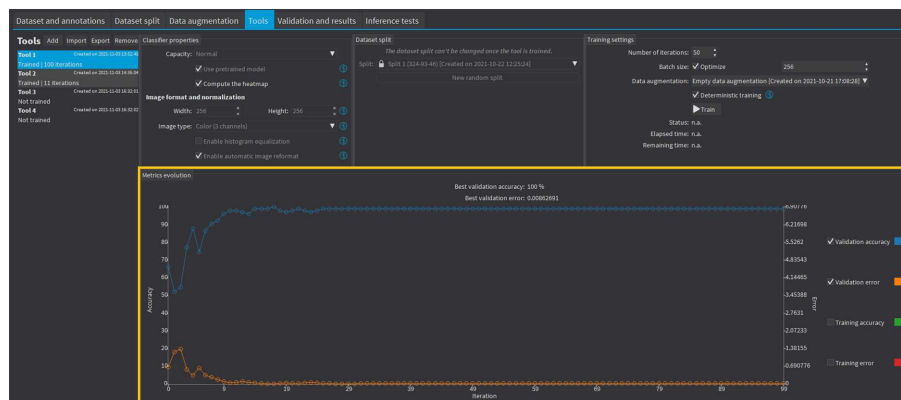
Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 340.

2.2. Validating the Results

In Deep Learning Studio:

- The metrics are always computed without applying data augmentation on the images.
- In the **Tools** tab, the metrics **Best validation error** and **Best validation accuracy** are computed during the training using the label weights. The evolution of several metrics during the training is also available.



- In the **Validation and 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).

Tool: Tool 1

Splits: All Training Validation Test Add images to test dataset

Metrics

Weighted error: 0.00437196

Weighted accuracy: 100 %

Dataset accuracy: 100 %

True labels

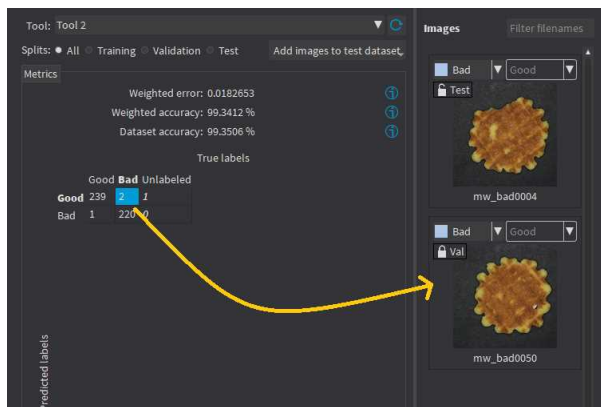
	Good	Bad	Unlabeled
Good	240	0	1
Bad	0	222	0



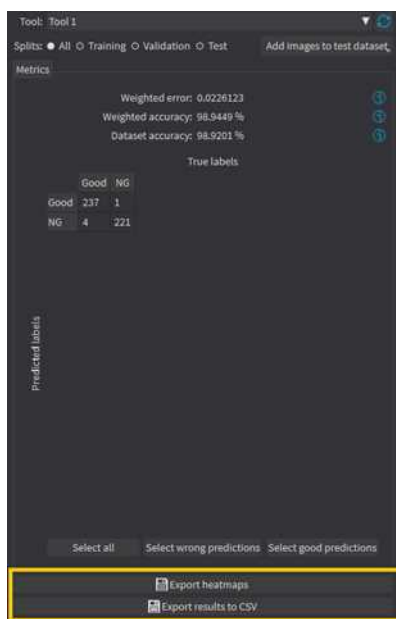
TIP

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 **Validation and 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 **Validation and results** tab, you can export the results using the 2 buttons below the metrics and confusion matrix.
 - The **Export heatmaps** button exports the heatmap for each image of your dataset.
 - The heatmaps are saved in the PNG format under the name ImageName_ToolName_heatmap_UniqueId.png.
 - This feature requires a **Deep Learning** license.
 - The **Export results to CSV** button exports the results for each image of your dataset as rows of a CSV file.
 - Each record contains the image filename, the ground truth label, the predicted label, the prediction probability and the probabilities for all the labels recognized by the tool.



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`.

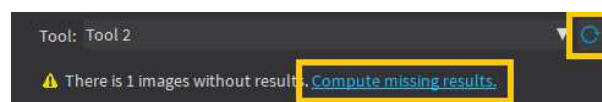


TIP

Call `EClassifier::Evaluate` to evaluate a dataset independently of the training.

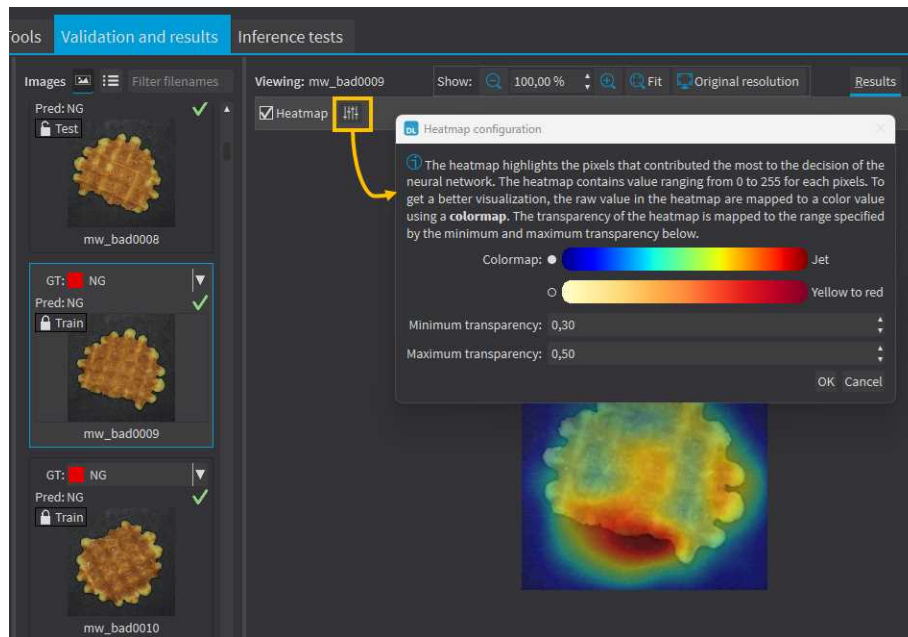
2.3. Classifying New Images

- In **Deep Learning Studio**:
 - Add new images to the dataset and refresh the results.



- Open the **Inference tests** tab to classify the new images and display the detailed results for these images.
- You can also export the results to CSV or export the heatmap from this tab.

- In the image viewer, you can enable or disable the heatmap visualization and configure how the heatmap is displayed (color map and transparency).



- 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).
- `EClassificationResult::GetHeatmap` and `EClassificationResult::GetColorizedHeatmap` return a heatmap highlighting the pixels that have contributed the most to get the most probable label.
 - The heatmap is only available when the classifier property `EClassifier::ComputeHeatmapWithResult` is set to True.
 - Use `EClassificationResult::HasHeatmap` to check if the result contains a heatmap.

- 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 a heatmap highlighting the pixels that contribute the most to a label.
 - In some cases, this heatmap can provide a rough localization of the object corresponding to the label.
 - The heatmap is colored, and the important parts are displayed in red.

**TIP**

Since large memory allocations take a lot of time, a classification does not release its 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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- **On Windows:**
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.

Capacity Extra Small

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	2.72	1.42	15.14	19.58
	4	0.97	=	2.38	=
	16	0.72	=	1.69	=
	64	0.55	=	0.96	=
256 × 256	1	5.36	3.95	8.29	65.00
	4	2.58	=	5.27	=
	16	1.91	=	3.26	=
	64	0.54	=	2.24	=
512 × 512	1	5.10	19.20	19.76	311.00
	4	7.52	=	12.11	=
	16	2.01	=	7.49	=
	64	2.73	=	7.40	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	132	n/a
	4	139	159
	16	166	242
	64	276	571
256 × 256	1	139	n/a
	4	166	241
	16	275	569
	64	712	1 879
512 × 512	1	166	n/a
	4	275	568
	16	711	1 877
	64	2 455	7 114

Capacity Small

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	3.26	2.65	22.99	33.00
	4	1.67	=	3.81	=
	16	0.88	=	2.02	=
	64	0.53	=	1.38	=
256 × 256	1	3.89	7.40	11.85	108.00
	4	3.00	=	7.95	=
	16	1.59	=	3.80	=
	64	0.59	=	3.35	=
512 × 512	1	6.58	34.50	27.94	521.00
	4	5.84	=	14.47	=
	16	2.19	=	10.82	=
	64	2.79	=	10.02	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	137	n/a
	4	147	180
	16	188	304
	64	350	799
256 × 256	1	147	n/a
	4	187	303
	16	349	796
	64	996	2 766

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
512 × 512	1	187	n/a
	4	349	795
	16	995	2 763
	64	3 579	10 635

Capacity Normal

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	5.61	4.65	15.26	47.30
	4	4.01	=	4.45	=
	16	1.55	=	2.27	=
	64	0.52	=	1.23	=
256 × 256	1	5.74	14.40	36.33	179.00
	4	5.87	=	8.73	=
	16	2.11	=	4.54	=
	64	0.67	=	3.74	=
512 × 512	1	7.75	60.00	49.12	760.00
	4	7.63	=	17.72	=
	16	2.74	=	15.28	=
	64	2.97	=	14.44	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	142	n/a
	4	153	201
	16	199	357
	64	383	983
256 × 256	1	153	n/a
	4	199	357
	16	381	980
	64	1 110	3 473
512 × 512	1	199	n/a
	4	381	979
	16	1 109	3 470
	64	4 021	13 433

Capacity Large

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	3.75	8.50	40.42	94.00
	4	2.44	=	6.49	=
	16	1.64	=	2.61	=
	64	0.29	=	2.26	=
256 × 256	1	4.25	25.60	55.36	367.00
	4	5.45	=	8.80	=
	16	0.84	=	6.46	=
	64	0.89	=	6.53	=
512 × 512	1	6.06	128.00	32.07	1 630.00
	4	3.12	=	23.61	=
	16	3.17	=	21.72	=
	64	3.25	=	20.43	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	152	n/a
	4	178	272
	16	281	611
	64	695	1 965
256 × 256	1	178	n/a
	4	281	610
	16	693	1 960
	64	2 341	7 360
512 × 512	1	281	n/a
	4	692	1 959
	16	2 338	7 355
	64	9 075	29 094

Capacity Extra Large

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	6.05	13.30	37.56	181.50
	4	3.78	=	8.91	=
	16	1.71	=	3.42	=
	64	0.40	=	3.80	=

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
256 × 256	1	6.81	50.00	28.23	726.00
	4	6.07	=	12.64	=
	16	1.25	=	10.76	=
	64	1.57	=	11.17	=
512 × 512	1	10.24	244.00	50.34	3 233.00
	4	4.76	=	42.70	=
	16	5.88	=	39.70	=
	64	-	=	-	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	170	n/a
	4	251	442
	16	577	1 295
	64	1 879	4 705
256 × 256	1	251	n/a
	4	576	1 293
	16	1 875	4 698
	64	7 074	18 333
512 × 512	1	576	n/a
	4	1 874	4 696
	16	7 070	18 326
	64	28 004	73 212

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

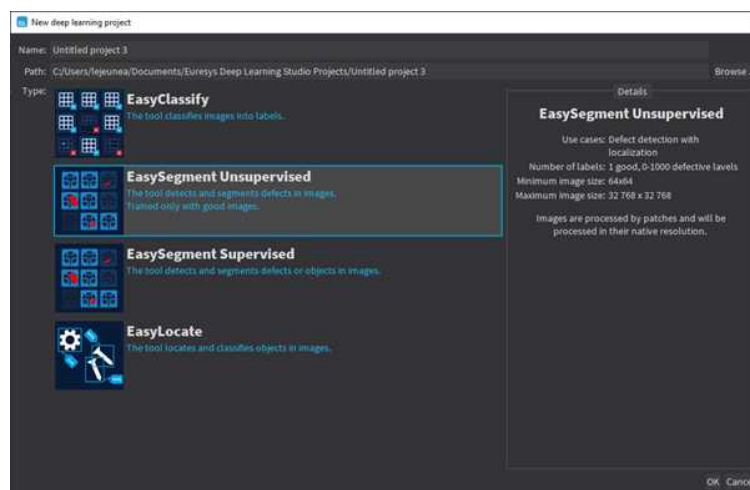
EasySegment Unsupervised is the deep learning tool part of the **EasySegment** segmentation library of **Open eVision**. It detects and segments defects in images.

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.

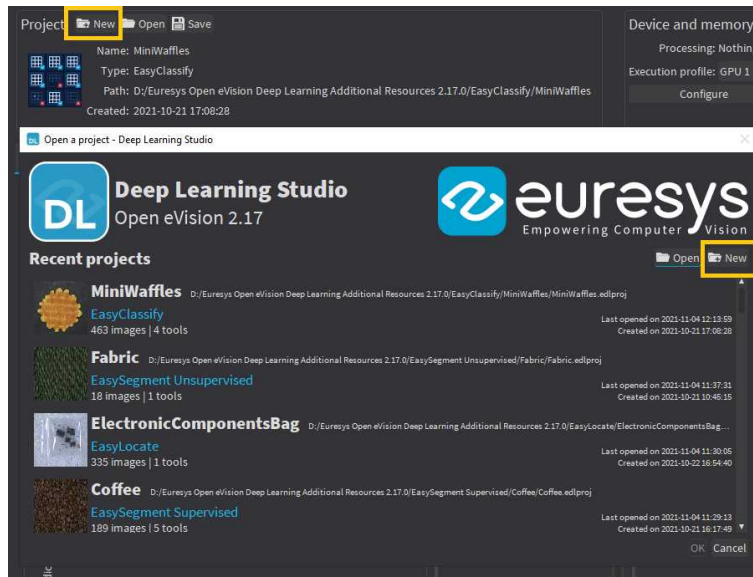
Deep Learning Studio

To create an **EasySegment Unsupervised** project in **Deep Learning Studio**:

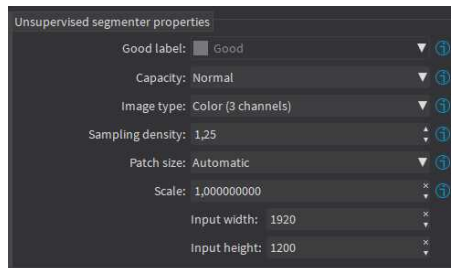
1. Start **Deep Learning Studio**.
2. Create a new project and select **EasySegment Unsupervised** in the **New deep learning project** dialog.



The following dialog is displayed when clicking on **New** in the **Open a project** dialog displayed at the start of **Deep Learning Studio** or when you click on **New** in the toolbar.



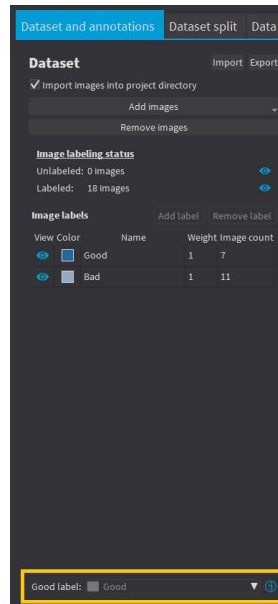
Configuration



The unsupervised segmenter tool has 6 parameters:

1. The `Good label` is the name of the class that contains the good images.

In **Deep Learning Studio**, the `Good label` is a project property that you must select in the `Dataset and annotations` tab below the list of labels.



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 (1 channel)`):

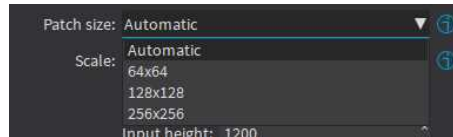
In the API:

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

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 340.

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).

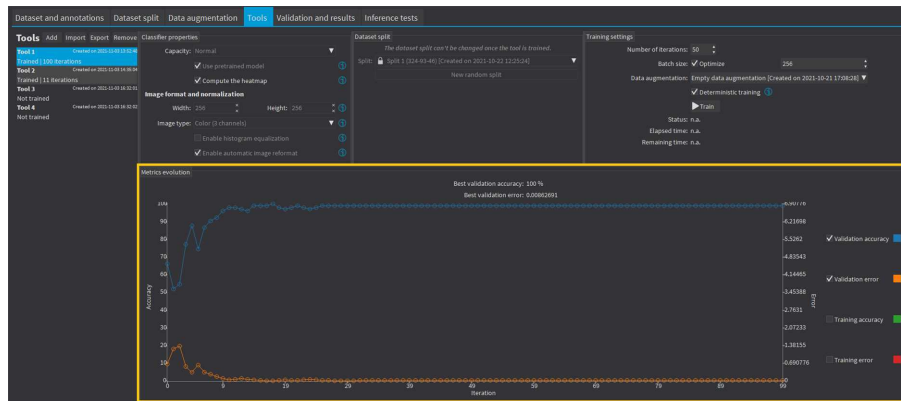
The unsupervised segmentation tool computes a score for each image (see `EUnsupervisedSegmenterResult::ClassificationScore`). The label of a result is obtained by thresholding this score with the segmenter classification threshold (`EUnsupervisedSegmenter::ClassificationThreshold`). So, the supervised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

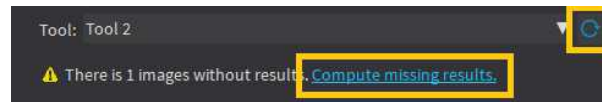
In the API:

- 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:

- Add new images to the dataset and refresh the results.



- Open the `Inference tests` tab to apply the segmenter to new images and display detailed results for these images.
- 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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- On **Windows**:
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.

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 approximately 25% of the time reported below.

Capacity Small

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	722.34	1 280	949.36	9 314
	4	237.25	=	344.92	=
	16	97.55	=	221.60	=
	64	68.11	=	191.46	=
128 × 128	1	289.34	1 472	619.84	6 946
	4	96.46	=	286.99	=
	16	51.83	=	201.88	=
	64	40.60	=	167.90	=

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
256 × 256	1	92.99	896	325.74	6 509
	4	48.62	=	214.72	=
	16	34.67	=	164.97	=
	64	38.92	=	150.79	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	5	n/a
	4	8	11
	16	13	26
	64	52	101
128 × 128	1	12	n/a
	4	23	41
	16	60	117
	64	221	430
256 × 256	1	44	n/a
	4	80	151
	16	233	454
	64	869	1 690

Capacity Normal

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	851.08	2 533	1 379.48	18 734
	4	245.62	=	532.91	=
	16	106.22	=	357.82	=
	64	70.44	=	271.87	=
128 × 128	1	312.90	3 216	1 095.83	17 763
	4	104.09	=	557.38	=
	16	55.60	=	369.82	=
	64	42.79	=	275.37	=
256 × 256	1	103.61	2 246	740.05	13 881
	4	53.91	=	421.52	=
	16	38.22	=	309.25	=
	64	46.27	=	257.49	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	3	n/a
	4	8	15
	16	26	50
	64	102	192
128 × 128	1	37	n/a
	4	53	98
	16	126	240
	64	441	834
256 × 256	1	145	n/a
	4	211	386
	16	487	935
	64	1 588	3 128

Capacity Large

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	965.58	6 178	2 183.06	40 833
	4	278.08	=	938.51	=
	16	117.13	=	622.66	=
	64	75.41	=	387.08	=
128 × 128	1	361.31	8 882	2 824.60	57 329
	4	129.73	=	1 394.44	=
	16	66.03	=	798.73	=
	64	53.80	=	470.68	=
256 × 256	1	187.35	7 254	2 969.24	38 020
	4	79.50	=	1 461.80	=
	16	51.84	=	667.86	=
	64	67.15	=	544.54	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	11	n/a
	4	20	36
	16	56	103
	64	209	383
128 × 128	1	133	n/a
	4	164	293
	16	300	561
	64	840	1 632

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
256 × 256	1	523	n/a
	4	654	1 163
	16	1 180	2 208
	64	3 664	6 768

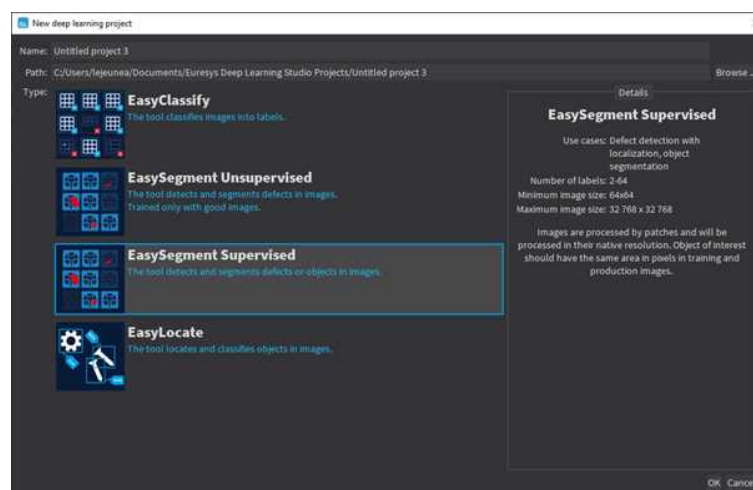
3.3. EasySegment Supervised

Tool and Configuration

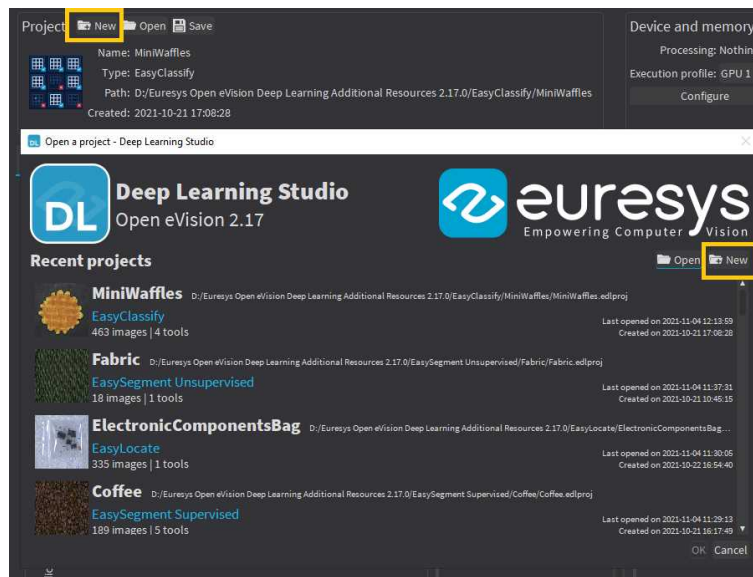
Deep Learning Studio

To create an **EasySegment Supervised** project in **Deep Learning Studio**:

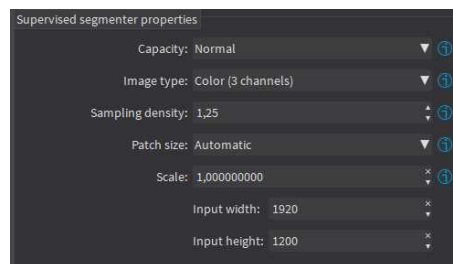
1. Start **Deep Learning Studio**.
2. Create a new project and select **EasySegment Supervised** in the **New deep learning project** dialog. .



The following dialog is displayed when clicking on **New** in the **Open a project** dialog displayed at the start of **Deep Learning Studio** or when you click on **New** in 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:

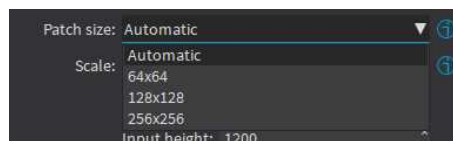
- The capacity is represented by the enumerate type `ESupervisedSegmenterCapacity`.
- `ESupervisedSegmenter::Capacity` sets the capacity of the tool.

2. The **Image type** (default: **Monochrome (1 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 340.

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.

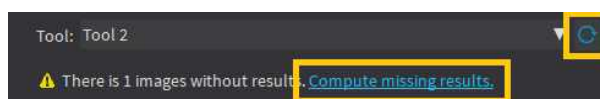


NOTE

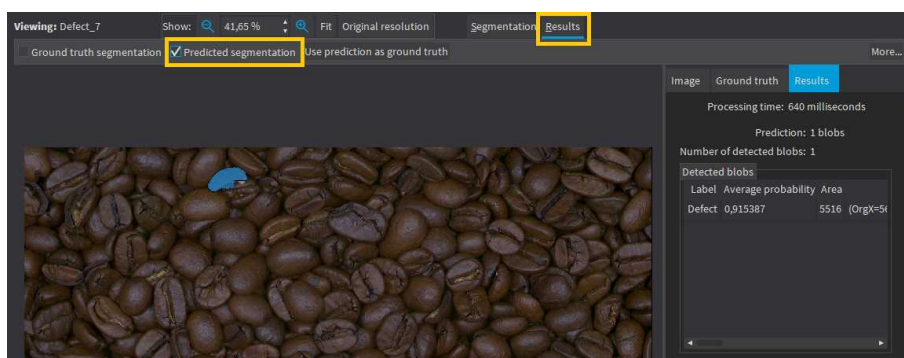
In the context of defect detection, an image is considered to be without defect when its score is below or equal to the threshold of the supervised segmenter tool.

In Deep Learning Studio:

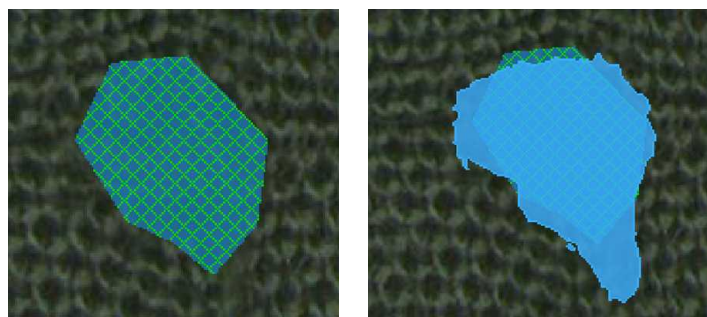
- Add new images to the dataset and refresh the results.



- Open the **Inference tests** tab to apply the segmenter to new images and display detailed results for these images.
- To visualize the segmentation of an image, check the **Predicted segmentation** option (CTRL + P) in the **Result** menu (ALT + R) of the image viewer.

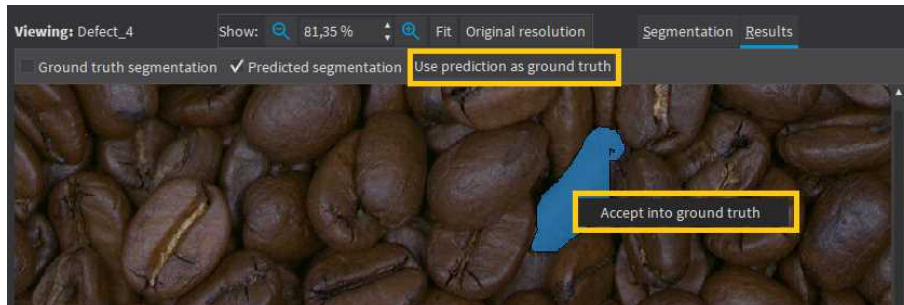


- If the image has a ground truth, check the **Ground truth segmentation** option (CTRL + G) to display it. It appears with a green pattern drawn over it.

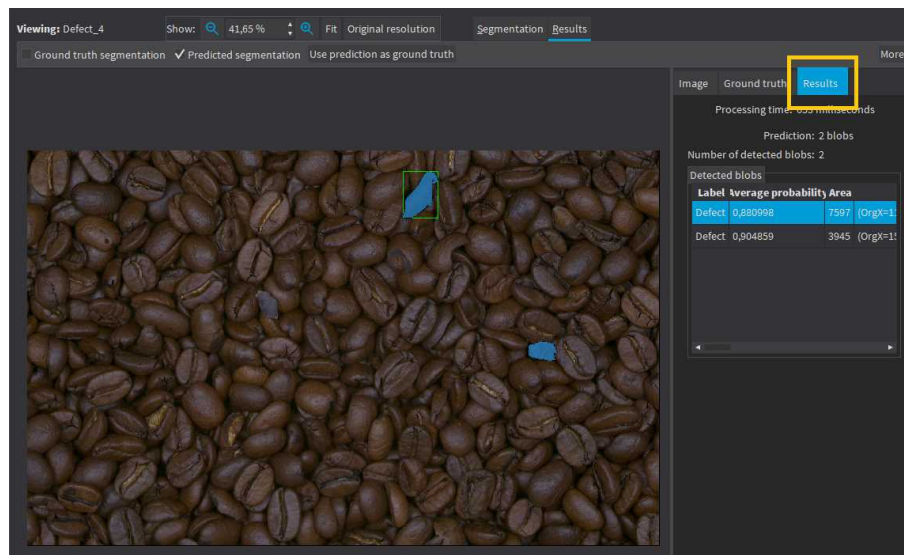


Ground truth (left) and prediction on top of ground truth (right)

- 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 `Results` tab of the image viewer.



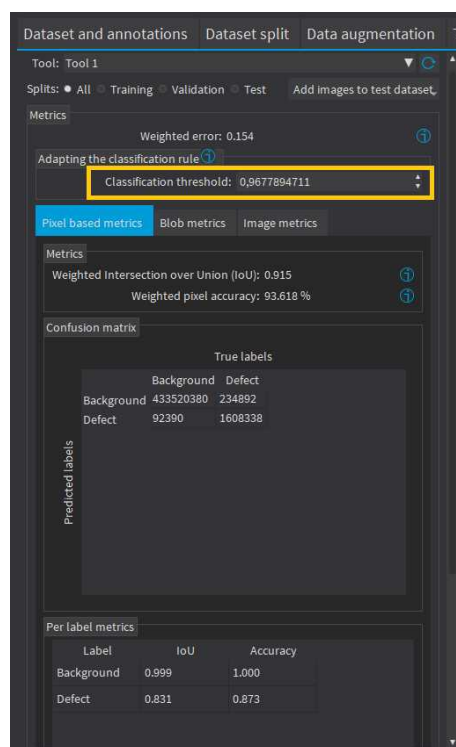
In the API:

- 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**.

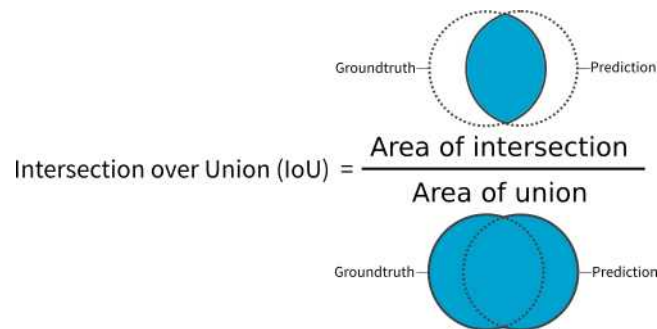


- In the API:
 - The metrics are represented by an `ESupervisedSegmenterMetrics` object.

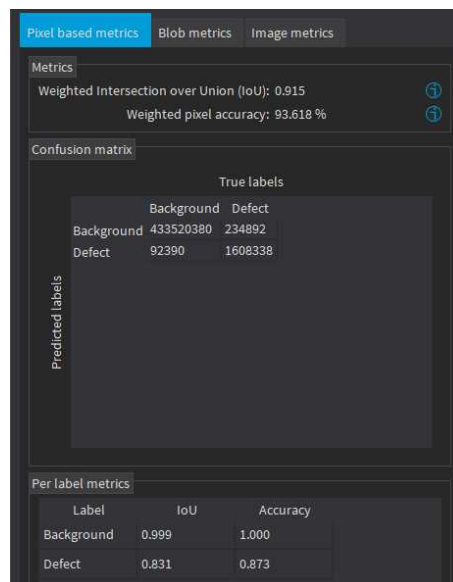
The pixel-based metrics

- 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.



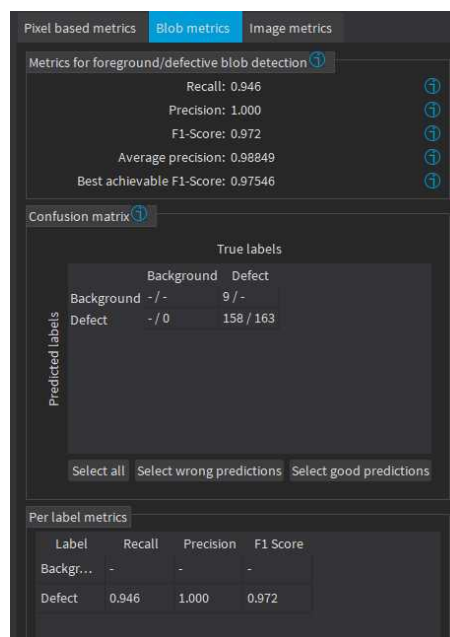
- The *accuracy* that is the proportion of the pixels of the label that are correctly predicted.



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*

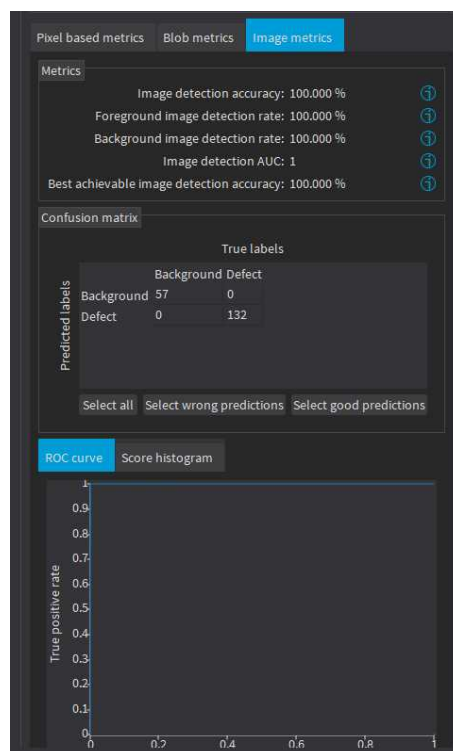


The image-based metrics

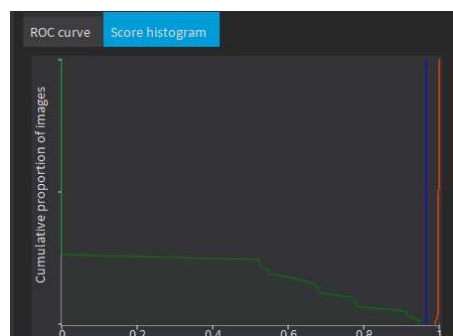
- The metrics related to the correct detection of the class of the images (background / foreground or good / defective in the context of defect detection):
 - The *image detection accuracy* that is the proportion of image correctly predicted to have foreground blobs or not.
 - The *foreground image detection 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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- On **Windows**:
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.

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 approximately 25% of the time reported below.

Capacity Small

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	583	2 008	669	32 443
	4	196	=	418	=
	16	90	=	354	=
	64	71	=	325	=
128 × 128	1	216	2 122	444	28 278
	4	92	=	359	=
	16	65	=	331	=
	64	64	=	358	=

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
256 × 256	1	105	2 435	435	28 811
	4	74	=	390	=
	16	66	=	365	=
	64	94	=	470	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	7	n/a
	4	22	41
	16	85	154
	64	288	560
128 × 128	1	33	n/a
	4	85	164
	16	310	604
	64	1 217	2 371
256 × 256	1	119	n/a
	4	344	669
	16	1 257	2 466
	64	5 770	10 516

Capacity Normal

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	651	5 771	1 080	67 544
	4	212	=	768	=
	16	104	=	661	=
	64	90	=	599	=
128 × 128	1	246	6 390	1 039	73 102
	4	113	=	862	=
	16	90	=	737	=
	64	89	=	815	=
256 × 256	1	125	8 120	1 127	86 212
	4	104	=	990	=
	16	95	=	873	=
	64	135	=	1 068	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	143	n/a
	4	168	209
	16	157	297
	64	604	1 145
128 × 128	1	193	n/a
	4	191	362
	16	644	1 240
	64	2 486	4 780
256 × 256	1	304	n/a
	4	750	1 453
	16	2 642	5 092
	64	10 208	19 648

Capacity Large

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	703	17 220	3 025	191 224
	4	263	=	2 367	=
	16	174	=	1 857	=
	64	164	=	1 694	=
128 × 128	1	321	22 050	3 392	264 352
	4	219	=	2 839	=
	16	188	=	2 158	=
	64	176	=	2 125	=
256 × 256	1	273	28 420	4 188	312 504
	4	219	=	3 459	=
	16	203	=	2 569	=
	64	-	=	-	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	57	n/a
	4	223	319
	16	422	716
	64	1 281	2 369
128 × 128	1	229	n/a
	4	528	929
	16	1 437	2 681
	64	5 388	10 007

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
256 × 256	1	915	n/a
	4	1 830	3 467
	16	5 880	10 991
	64	20 545	39 553

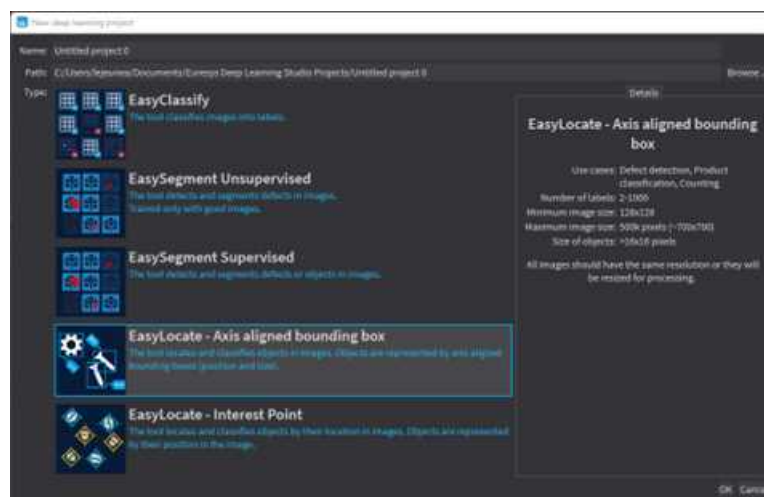
4. EasyLocate - Locating Objects and Defects

4.1. Tool and Configuration

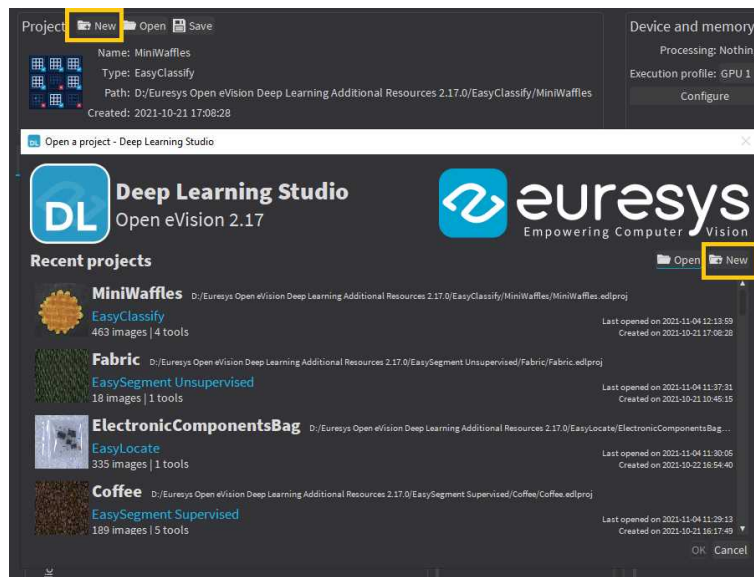
Deep Learning Studio

To create an **EasyLocate** tool in **Deep Learning Studio**:

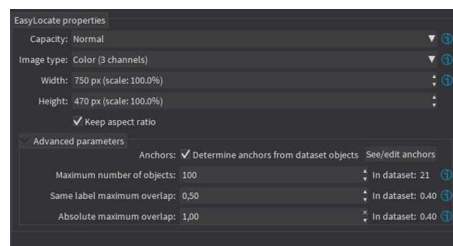
1. Start **Deep Learning Studio**.
2. Create a new project and select **EasyLocate Axis Aligned Bounding Box** or **EasyLocate Interest Point** in the **New deep learning project** dialog.



The following dialog is displayed when clicking on **New** in the **Open a project** dialog displayed at the start of **Deep Learning Studio** or when you click on **New** in 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)**):

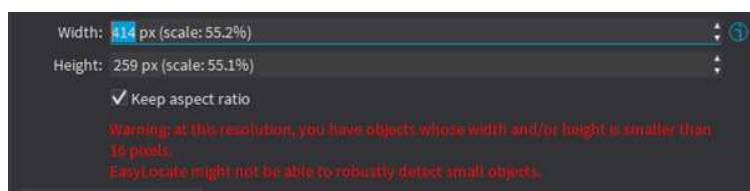
In the API:

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



In the API:

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

EasyLocate Axis Aligned Bounding Box configuration (advanced parameters)

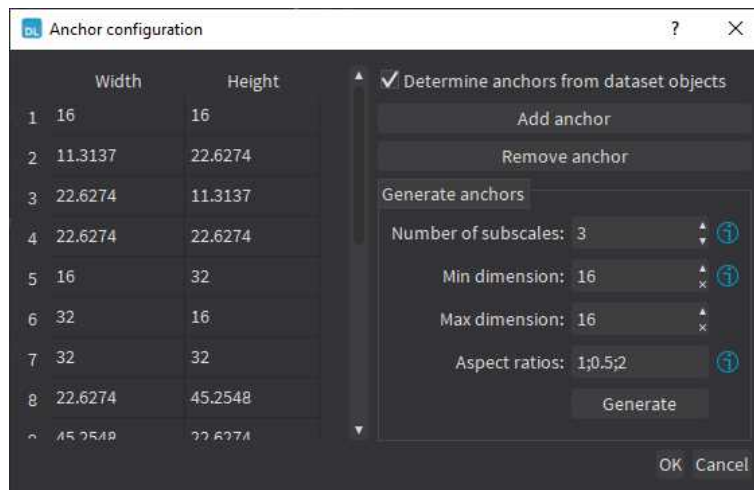
The **EasyLocate Axis Aligned Bounding Box** tool has 4 advanced parameters linked to **EasyLocate** neural network design.

- 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.



TIP

The anchors must be defined with respect to the width and height of the current tool. Thus, when specifying the anchors manually, the width and height parameters are locked.

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 `ELocatorBase::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`.

- 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.

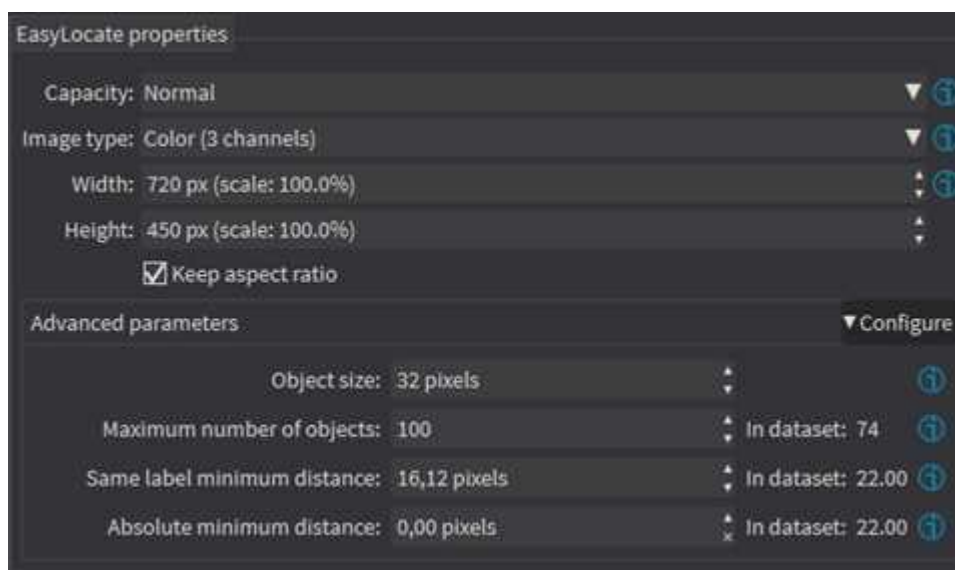
In the API:

- 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.

EasyLocate Interest Point configuration (advanced parameters)

The **EasyLocate Interest Point** tool has 4 advanced parameters.



- The **Object size** is the size of the objects that you want to detect in the image.
 - By default, the value of this parameter is the object size specified for the dataset.
 - The object size has the same role as the anchors for **EasyLocate Axis Aligned Bounding Box**.
 - The object size is defined with respect to the size of the images in the dataset. When training a tool with a different input resolution, the object size parameters should not change (they are internally adapted according to the resolution of the training images and the width and height set for the tool).

In the API:

- Use `EInterestPointLocator::ObjectSize`

- 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 `ELocatorBase::MaxNumberOfObjects`.

3. The `Same label minimum distance` is the minimum distance between objects with the same label.

- By default the value is a third of the object size.

In the API:

- Use `EInterestPointLocator::SameLabelMinDistance`

4. The `Absolute minimum distance` is the minimum distance between objects, regardless of their label.

- By default the value is 0 and it means that the tool can predict two objects with different labels but with the exact same position.

In the API:

- Use `EInterestPointLocator::AbsoluteMinDistance`

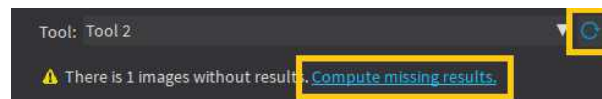
Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 340.

4.2. Locating Objects

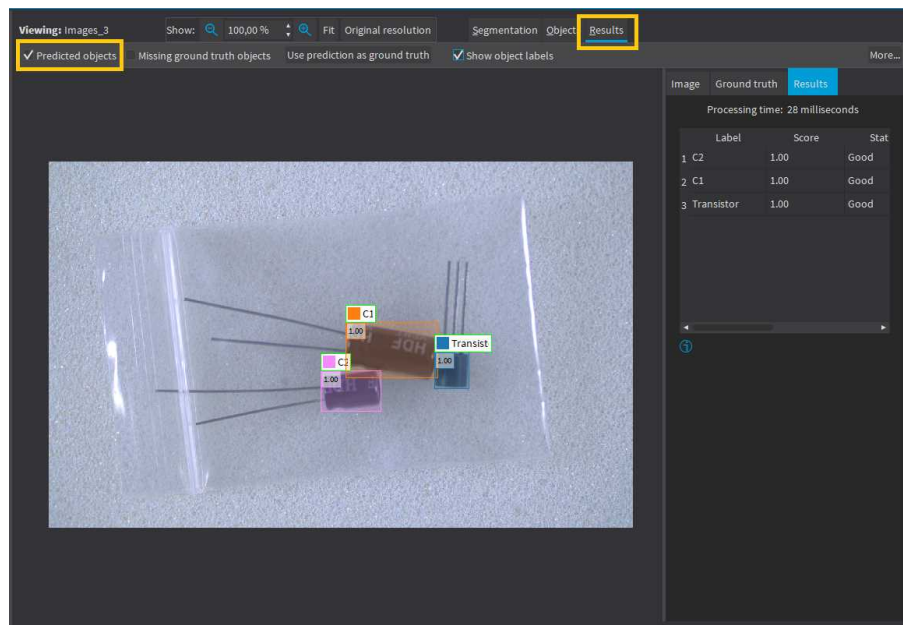
[In Deep Learning Studio:](#)

- To apply **EasyLocate** to new images:
 - Add new images to the dataset and refresh the results.



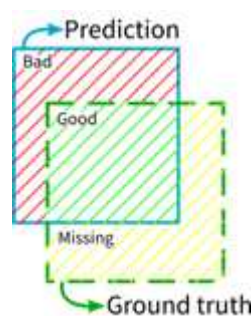
- Open the `Inference tests` tab to apply the tool to new images and display detailed results for these images.

- 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).



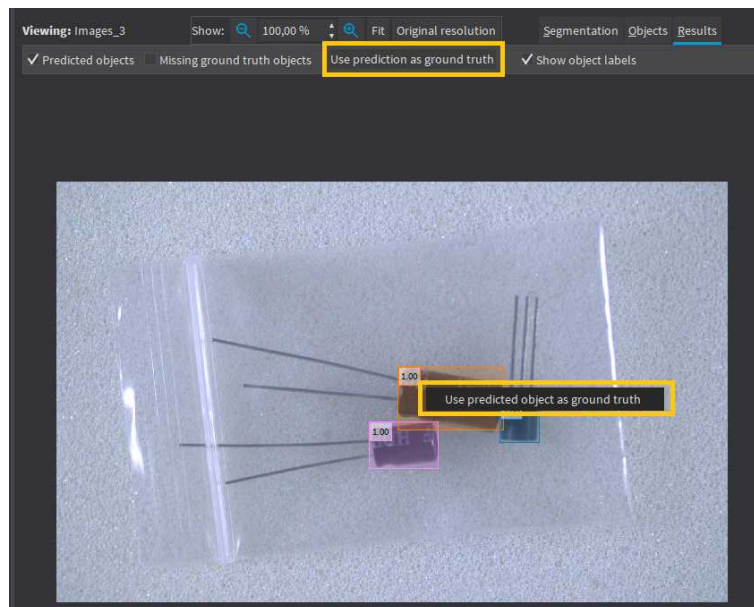
- If the image has a ground truth, check the **Missing ground truth object** option (CTRL + G) to display missing objects. They appear with a yellow pattern drawn over it.
- In the **Results** tab on the right side:
 - The list of detected objects shows their label, their score, whether they are matched to a ground truth object and their predicted bounding box or position.
 - 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.

For **EasyLocate Axis Aligned Bounding Box**, 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.



In the API

- To apply **EasyLocate** to an image, use `ELocatorBase::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.
- Use `ELocatorResult::LocatorFeatures` to check if the result was computed by **EasyLocate Axis Aligned Bounding Box** or **EasyLocate Interest Point**.

The value of `ELocatorResult::LocatorFeatures` is a combination (using the binary OR (`|`) operator) of one or more `ELocatorFeature` values. It indicates the type of information predicted by the tool:

- For **EasyLocate Interest Point**: `ELocatorFeature_Position`.
 - For **EasyLocate Axis Aligned Bounding Box**: `ELocatorFeature_Position` | `ELocatorFeature_Size`.
- With **EasyLocate Interest Point**, for each predicted object, use:
 - `ELocatorObject::PositionX` to get the X coordinate of the object.
 - `ELocatorObject::PositionY` to get the Y coordinate of the object.
 - `ELocatorObject::Label` to get its label.
 - `ELocatorPredictedObject::Probability` to get its predicted probability.
 - With **EasyLocate Axis Aligned Bounding Box**, for each predicted object, use:
 - `ELocatorObject::PositionX` to get the X coordinate of the center of the bounding box.
 - `ELocatorObject::PositionY` to get the Y coordinate of the center of the bounding box.
 - `ELocatorObject::OrgX` to get the X coordinate of the top left origin of the bounding box.

- `ELocatorObject::OrgY` to get the Y coordinate of the top left origin of the bounding box.
- `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.

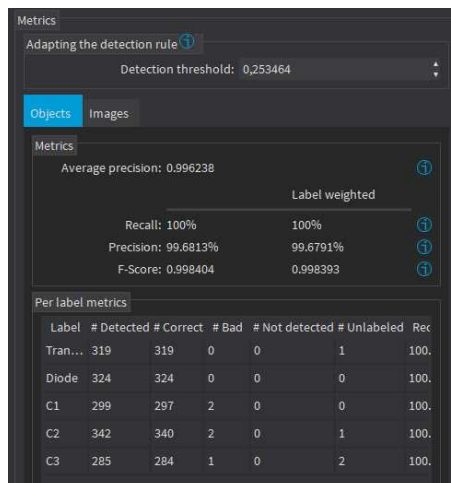
4.3. Validating the Results

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**.



Label	# Detected	# Correct	# Bad	# Not detected	# Unlabeled	Rec
Tran...	319	319	0	0	1	100.
Diode	324	324	0	0	0	100.
C1	299	297	2	0	0	100.
C2	342	340	2	0	1	100.
C3	285	284	1	0	2	100.

In the API:

- The metrics are represented by an object `ELocatorMetrics`.

The object-based metrics

The object-based metrics are computed by matching actual, ground truth objects to detected objects.

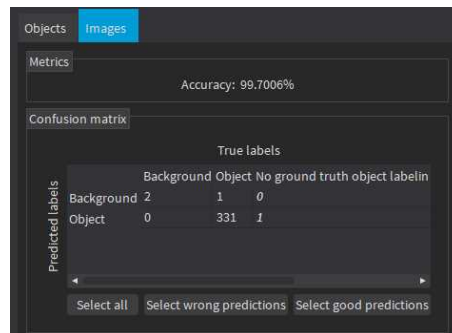
A ground truth object and a detected object are matched together if:

- They have the same label.
- With **EasyLocate Axis Aligned Bounding Box**:
 - 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.
- With **EasyLocate Interest Point**:
 - Their distance is lower than the `Same label minimum overlap` parameter of the tool
 - There is no other ground truth that has a smaller distance with the detected object and there is no other detected object that has a smaller distance 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.

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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- **On Windows:**
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.
- The benchmarks were obtained using **EasyLocate Axis Aligned Bounding Box**.

- For **EasyLocate Interest Point**, the training and inference speeds are approximately the same. The small variations (a few percent slower or faster) in the processing speed depend on the parameters of the tool.

Capacity Small

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	4.19	30.13	18.55	504
	4	4.44	=	7.43	=
	16	1.81	=	4.16	=
	64	0.41	=	3.45	=
256 × 256	1	4.85	84	32.02	1 959
	4	6.88	=	16.74	=
	16	1.45	=	13.51	=
	64	1.37	=	14.19	=
512 × 512	1	11.32	341	66.10	9 314
	4	5.70	=	60.85	=
	16	5.38	=	53.89	=
	64	-	=	56.28	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	175	n/a
	4	241	354
	16	503	879
	64	1 553	2 979
256 × 256	1	241	n/a
	4	503	879
	16	1 553	2 979
	64	5 884	11 511
512 × 512	1	503	n/a
	4	1 553	2 979
	16	5 884	11 511
	64	23 455	45 885

Capacity Normal

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	3.75	32	19.03	645
	4	2.43	=	8.14	=
	16	1.90	=	4.48	=
	64	0.42	=	3.69	=
256 × 256	1	7.00	91	32.36	2 717
	4	6.83	=	18.14	=
	16	1.59	=	14.88	=
	64	1.54	=	15.47	=
512 × 512	1	8.93	391	71.63	12 646
	4	5.62	=	66.94	=
	16	5.39	=	58.83	=
	64	-	=	61.78	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	178	n/a
	4	248	369
	16	528	929
	64	1 648	3 168
256 × 256	1	248	n/a
	4	528	929
	16	1 648	3 168
	64	6 256	12 255
512 × 512	1	528	n/a
	4	1 648	3 168
	16	6 256	12 255
	64	24 937	48 849

Capacity Large

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	6.31	76	48.57	1 194
	4	4.99	=	14.32	=
	16	1.08	=	10.05	=
	64	0.85	=	7.34	=

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
256 × 256	1	8.80	205	65.37	5 694
	4	3.46	=	38.04	=
	16	2.25	=	30.14	=
	64	2.32	=	29.54	=
512 × 512	1	25.93	866	168.90	26 320
	4	9.11	=	128.96	=
	16	8.52	=	115.15	=
	64	-	=	-	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	288	n/a
	4	421	714
	16	952	1 776
	64	3 075	6 023
256 × 256	1	421	n/a
	4	952	1 776
	16	3 075	6 023
	64	11 701	23 144
512 × 512	1	952	n/a
	4	3 075	6 023
	16	11 701	23 144
	64	46 450	91 874

PART VI
3D PROCESSING TOOLS

1. Easy3D - Using 3D Toolset

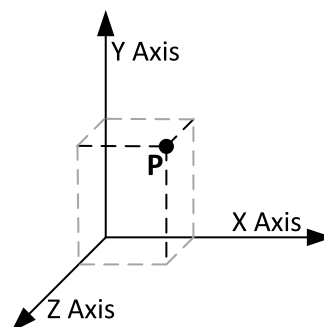
1.1. Basic Concepts

Easy3D

- **Easy3D** is a set of tools for solving computer vision problems using 3D acquisition and processing.
- **Easy3D** introduces the depth map, the ZMap, the point cloud and the mesh representations to **Open eVision**.
- **Easy3D** offers many operators to manipulate these 3D representations, including loading and saving in several standard formats. Additional classes, like 3D box, 3D plane or 3D sphere, are available in **Easy3D** and are used in the high level tools **Easy3DObject** and **Easy3DMatch**. Finally, an interactive 3D viewer, **E3DViewer**, is part of **Easy3D**.
- 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.

E3DPoint

In **Open eVision**, the 3D points are represented by a container called **E3DPoint**. **E3DPoints** are the basic containers for the data processing in a 3D space.

- A **E3DPoint** is defined by its X, Y and Z coordinates.
- Use the function **DistanceTo** to compute distances between points
- You can combine points to form point clouds (see below).

E3DLine

To represent lines in a 3D space, **Open eVision** offers the object [E3DLine](#).

- An [E3DLine](#) is defined by two points.

E3DPlane

To represent planes in a 3D space, **Open eVision** offers the object [E3DPlane](#).

- An [E3DPlane](#) is defined by its normal vector and the signed distance to the origin of the axes.
- Use the function [DistanceTo](#) to compute the distance to any point.
- Use the function [ProjectPoint](#) to project points onto planes.
- Use the function [Transform](#) with transformation matrices to rotate and translate planes.

E3DSphere

To represent spheres in a 3D space, **Open eVision** offers the object [E3DSphere](#).

- An [E3DSphere](#) is defined by its center point and its radius.
- Use the function [DistanceTo](#) to compute the distance to any point.
- Use the function [Transform](#) with transformation matrices to rotate and translate spheres.

E3DBox

[E3DBoxes](#) are rectangular parallelepipeds in a 3D space. They can have any orientation and be centered on any point in the 3D space.

- An [E3DBox](#) is defined by a center point, 3 axes and the extension of the box along each axis.
- Use the function [Transform](#) with transformation matrices to rotate and translate boxes.

E3DObject

In **Open eVision**, an [E3DObject](#) is a geometric representation of a set of 3D points.

- Use the [E3DObject](#) to get the characteristics such as the extend along the 3 axes, the area, the total volume and the average position of the set of points.
- Use the function [BasePlane](#) to get the fitted plane of the set of points.
- Use the function [BoundingBox](#) to get the bounding box, represented by a [E3DBox](#) enclosing the set of points.

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 ($-3.402823 \text{ e}+38$) for the 32 bits floating point depth map types ([EDepthMap32f](#)).

The calibration process aims to convert the depth map representation to real, metric 3D representations such as point clouds or meshes.

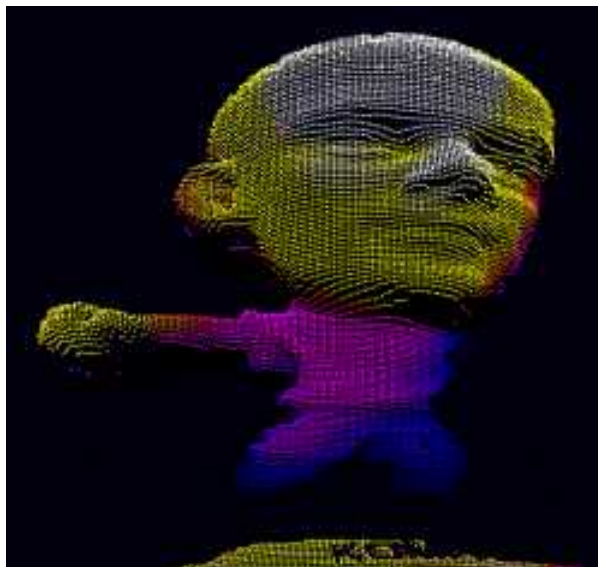


TIP

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.

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

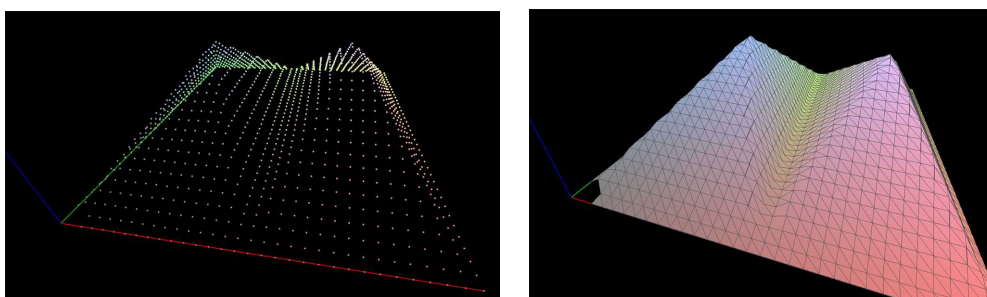
A Mesh is a geometric representation of a 3D surface, a set of connected 3D points.

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.

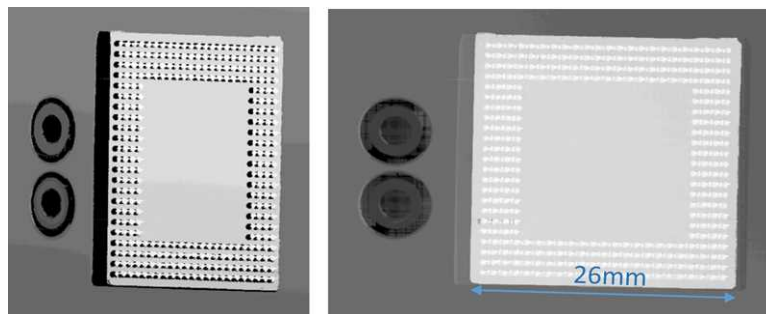
EMesh uses a metric space representation that can be generated from a depth map and that can be used to produce a ZMap.

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

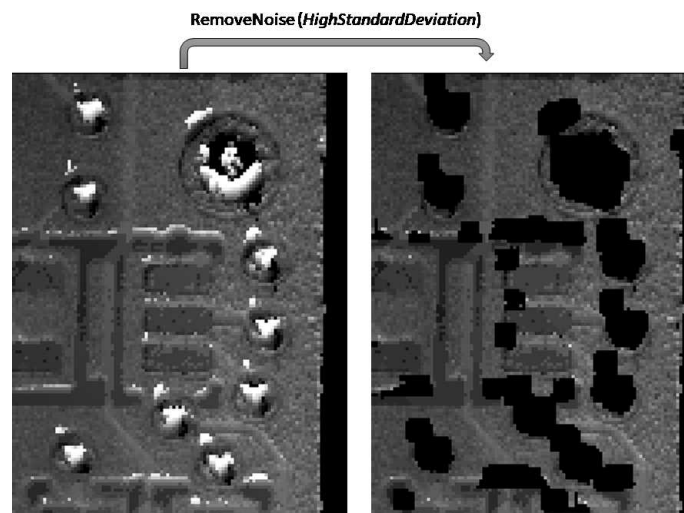
The `EFilters` class contains static methods used to apply filters to ZMaps or depth maps.

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
EZMap16 zmap;
zmap.Load(...);

// Compute the filtered ZMap. The new ZMap is called filteredZmap
// The size of the kernel is 7x7, the threshold is 30.0
EZMap16 filteredZmap;
filteredZmap.SetSize(zmap);
EFilters::RemoveNoise(zmap, filteredZmap, ENoiseRemovalMethod_HighStandardDeviation, 3, 30.0, 0.0);
```

Median

The **Median** method removes the 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.
- There are options for this method:
 - You can adjust the width and the height of the median filter.
 - The function can ignore the undefined pixels or take them into account (as the lowest possible value) when computing the medians.
 - Taking the undefined pixels into account removes spickles in the middle of undefined pixels and smoothens the edges between defined and undefined pixels.
 - Ignoring them does not. It can also be much slower.

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.

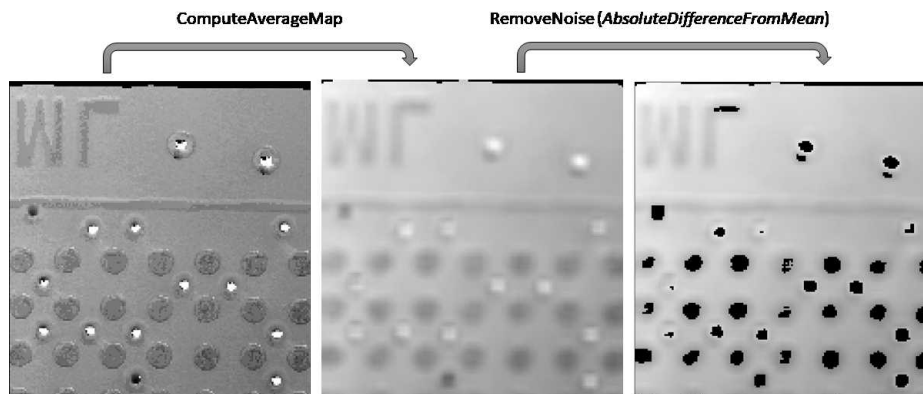
- You can use this method to visually determine the threshold value to use with the **RemoveNoise** method when using the **ENoiseRemovalMethod_HighStandardDeviation** setting.



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
EZMap16 zmap;
zmap.Load(...);

// Compute the filtered ZMap. The new ZMap is called averagedZMap
// The size of the kernel is 7x7, the threshold is 30.0
EZMap16 averagedZMap;
averagedZMap.SetSize(zmap);
EStatistics::ComputeAverageMap(zmap, averagedZMap, 3, 0.2);

// Compute the filtered ZMap. From averagedZMap, compute filteredZMap
// The size of the kernel is 31x31, the threshold is 20.0
EZMap16 filteredZMap;
filteredZMap.SetSize(zmap);

EFilters::RemoveNoise(averagedZMap, filteredZMap, ENoiseRemovalMethod_AbsoluteDifferenceFromMean, 15, 20.0, 0.2);
```

ComputePixelStatistics

The `ComputePixelStatistics` method returns basic statistical information about pixel values:

- Minimum
- Maximum
- Average
- Standard deviation
- Number of valid (not undefined) pixels).

Use an `ERegion` object to specify the region of the ZMap or depth map used to compute the statistics.

[ComputeStatistics](#)

The [ComputeStatistics](#) method returns the same information as the [ComputePixelStatistics](#) method, but scaled with respect 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, distances, curvatures 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 methods [FillAttributeBuffer](#) or [AddCustomAttributeBuffer](#).
 - To retrieve the attribute, use the method [GetAttributeBuffer](#).
 - To allocate the attribute, use the methods [AllocateAttributeBuffer](#) and [AllocateCustomAttributeBuffer](#).
 - 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.

These methods are showcased in the following code snippets:

- ["Add an Attribute to an EPointCloud with Initial Data" on page 1](#)
- ["Add an Attribute to an EPointCloud without Initial Data" on page 1](#)
- ["Retrieve an Attribute from an EPointCloud" on page 1](#)

Normals and Curvatures

- Use the function [EPointCloud::ComputeNormalsAndCurvatures](#) to compute the normals and, optionally, the curvatures of all of the points in the point cloud:
 - The normals are stored in the `Normal` attribute buffer.
 - The curvatures are stored in the `Curvature` attribute buffer.

Coordinates Transformations

[Geometric transformations](#)

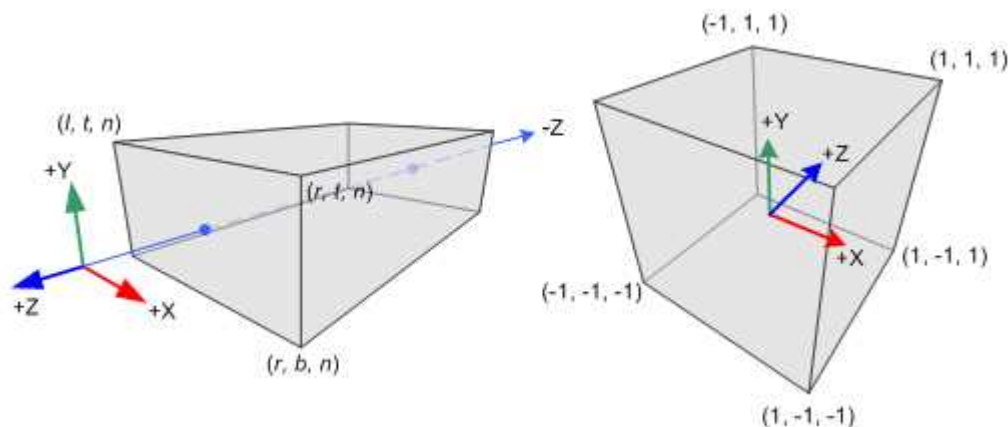
Transforms allow you to reposition the point cloud inside the 3D space.

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.Load("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.Load("c:\\images\\data.pcd");

// Explicitly decimate the point cloud

EGridDecimator decimator(10.f);
EPointCloud pcDecimated;

decimator.Decimate(pc, pcDecimated);
pcDecimated.SavePCD("c:\\images\\decimatedData.pcd");
```

Filtering out noise

The [EPointCloudFilter](#) class filters a point cloud in the same way [EFilters::RemoveNoise](#) filters a depth map or ZMap.

A criterion based on the neighborhood of each point is computed. The points that have a criterion score above the threshold are removed.

- Select between the following criterions:
 - The *average distance* of the point from its neighborhood. It may remove some points at the borders of a noiseless surface.
 - The *standard derivation* within the point neighborhood. It is more likely to remove noiseless points that are close to noisy points.
- You can set the size of the neighborhood.
- The threshold is defined as:

$$\text{the mean of the criterion over the cloud} + \text{the standard derivation of the criterion over the cloud} \times \text{a factor provided by the user}$$

- The default value for the factor is 1, as it is a good value in most cases.
- You can replace the noisy points by the average of their neighborhood instead of removing them.

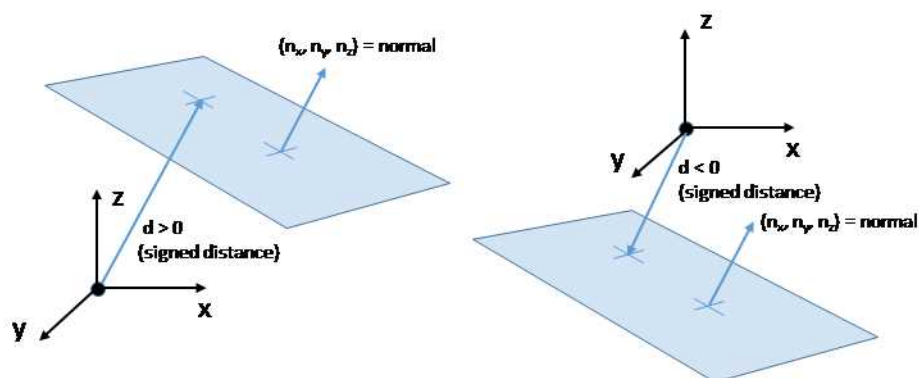
Managing Planes

E3DPlane

A plane can be represented as an [E3DPlane](#) object.

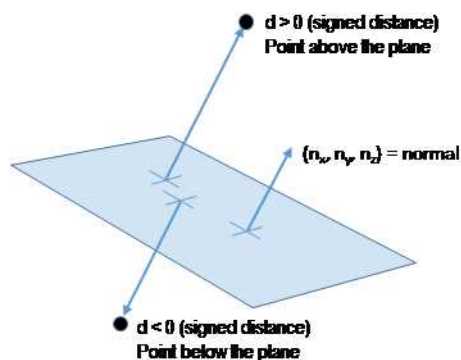
This plane is characterized by:

- 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.



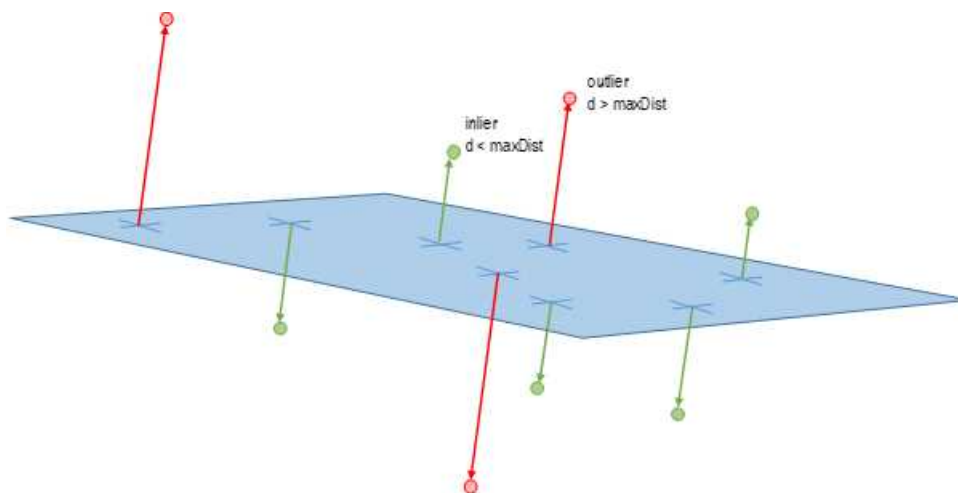
EPlaneFinder

You can search for a plane in a point cloud using the object [EPlaneFinder](#) object.

The main parameters of this object are:

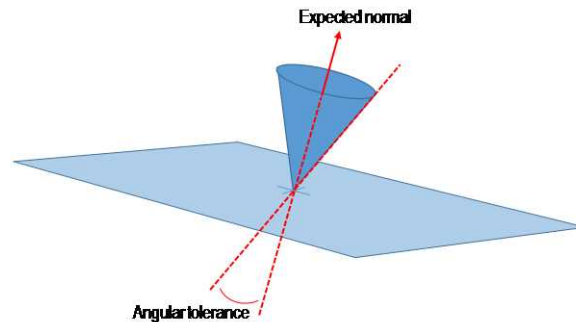
- The maximum distance between the searched plane and a point that belongs to this plane.
- The expected ratio between the numbers of inliers and the total number of points in the point cloud.
 - An **inlier** is a point that belongs to a plane (closer than this maximum distance).
 - An **outlier** is 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

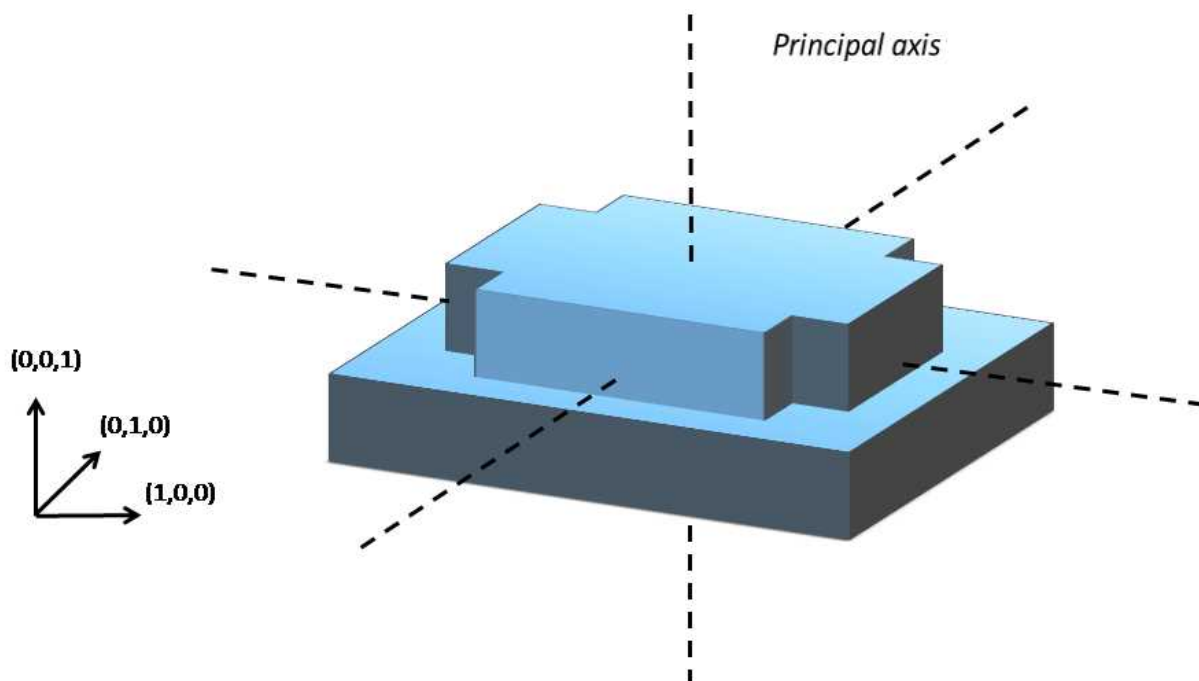
EPrincipalAxisExtractor

The `EPrincipalAxisExtractor` computes the “principal axis” of an object from a point cloud (`EPointCloud`) and returns a `E3DTranformMatrix` containing a solid transformation that defines a new orthogonal basis.

This new orthogonal basis has the following characteristics:

- The center is the center of gravity of the point cloud.
- The 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.

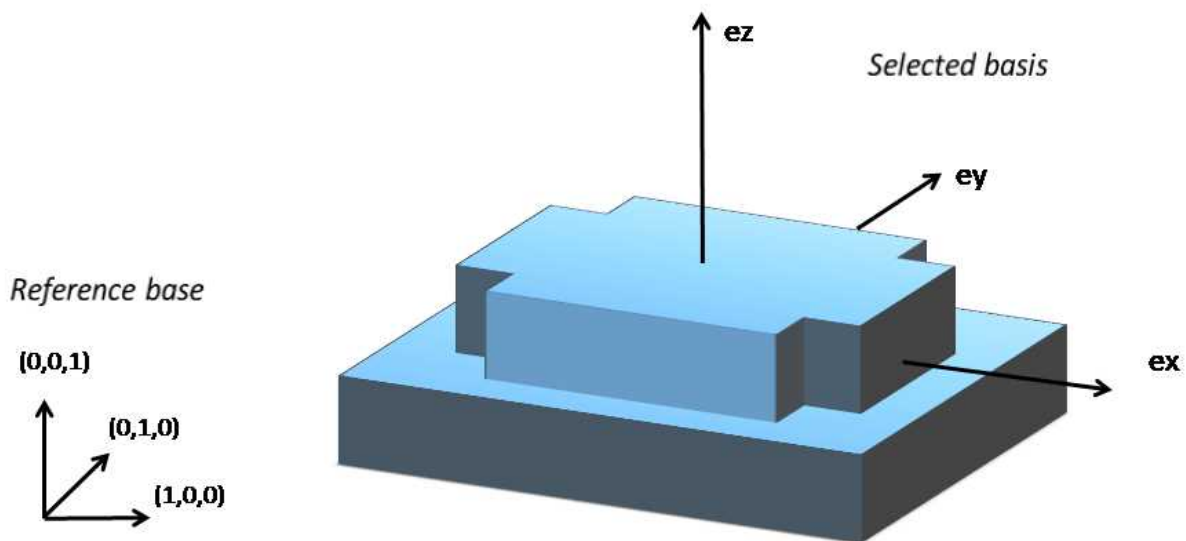
Specification of a reference transformation

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

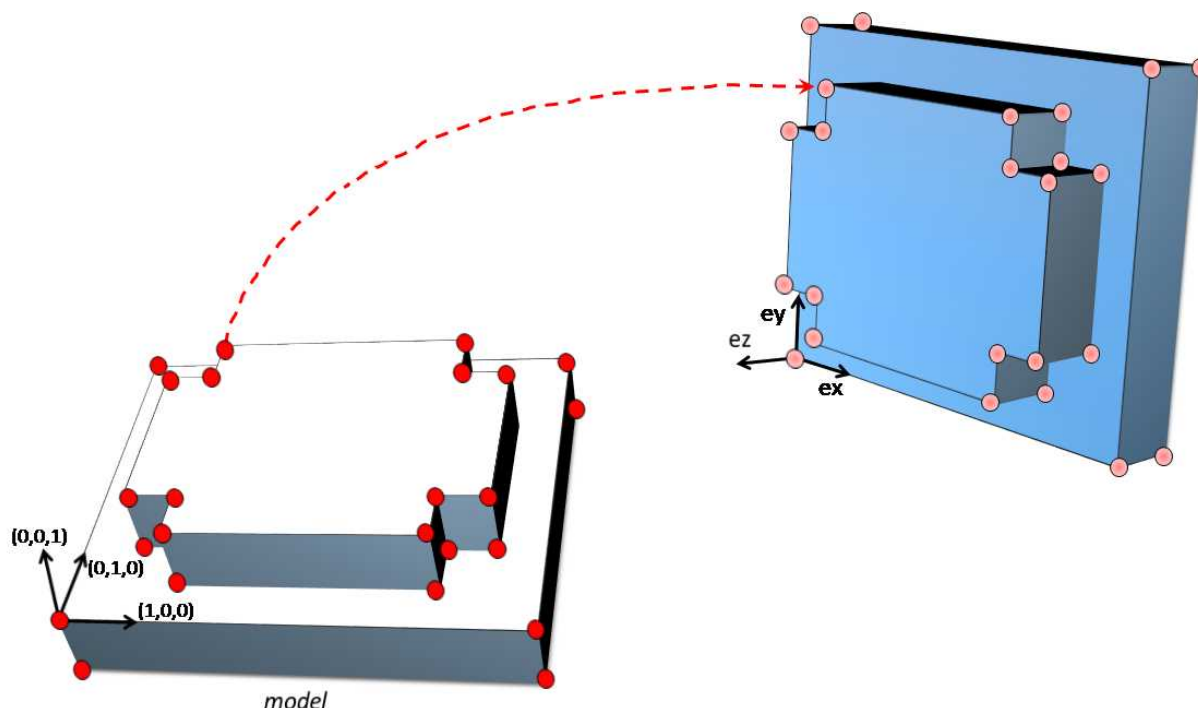
A [EFeaturesAligner](#) object finds the best transformation that maps a list of points to another list of points.

- 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 P1, results in point P1b
E3DPoint P1 = E3DPoint(...);
E3DPoint P1b = alignBase*P1;
```

As you can see, the application of the transformation on a point is simply done by multiplying the transformation matrix by the point (as done in the example above).

On the other hand, if you need to transform a point cloud or a list of points, it is more efficient to use the `ApplyTransform()` method of an `EAffineTransformer` object.

Using Spheres

E3DSphere

Use an object [E3DSphere](#) to represent a sphere characterized by:

- Its center as a [E3DPoint](#).
- Its radius.

Once a sphere is defined, use the method [DistanceTo](#) to measure the distance between the sphere and any point in the space, defined as the distance to the nearest point of the sphere.

ESphereFitter

The operator [ESphereFitter](#) computes a fit on all the points of a point cloud and returns the object [E3DSphere](#) that minimizes the distance between the point cloud and the sphere (calculated according to the least squares).

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 and each of these triangles has an associated normal.

- 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 or a ZMap and use it to produce a ZMap.

Generation from a depth map

- An [EMesh](#) object is generated from a depth map using the class [EDepthMapToMeshConverter](#).
- 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 the method [SetCalibrationModel](#) to select a calibration model and the method [Convert](#) to generate an [EMesh](#) from an 8 bits or 16 bits depth map.

Generation from a ZMap

- An [EMesh](#) object is generated from a ZMap using the class [EZMapToMeshConverter](#).
- Use the method [SetMaxEdgeLength](#) to filter out the large triangles from the mesh. These large triangles occur when the colors of two neighboring pixels are very far apart.

Generation from a point cloud

- An **EMesh** object is generated from a point cloud using the class **EPointCloudToMeshConverter**.
 - The **EPointCloud** is first converted to a ZMap that is then converted to an **EMesh**.
- Use the method **SetProjectionPlane** to set the projection plane.
- Use the method **SetResolution** to specify the size of the triangles of the mesh.
- Use the method **SetMaxEdgeLength** to filter out the large triangles from the mesh.

 This functionality is illustrated in a C++ and a C# MFC sample application called **Easy3DPointCloudToMesh**.

Access and usage

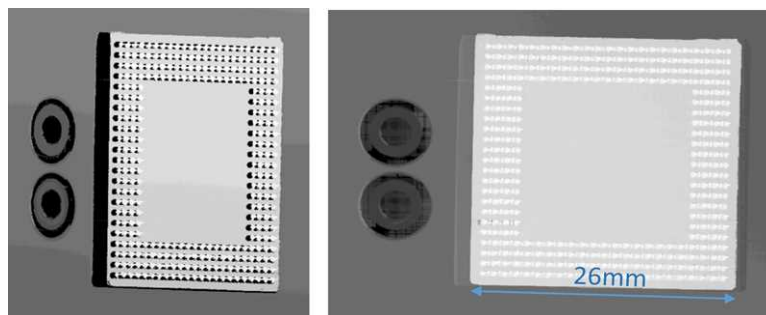
- In an **EMesh** object the 3D world positions are stored as an **EPointCloud** (accessible through the method **GetPointCloud()**). The triangle mesh is stored as an array of point indexes, where 3 consecutive indexes define a triangle.
 - The method **GetTriangleIndexes()** provides a read-only access to the triangle mesh.
 - The normals are stored as an array of **E3DPoints** readable through the **GetNormals** method.
- 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 420). 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.

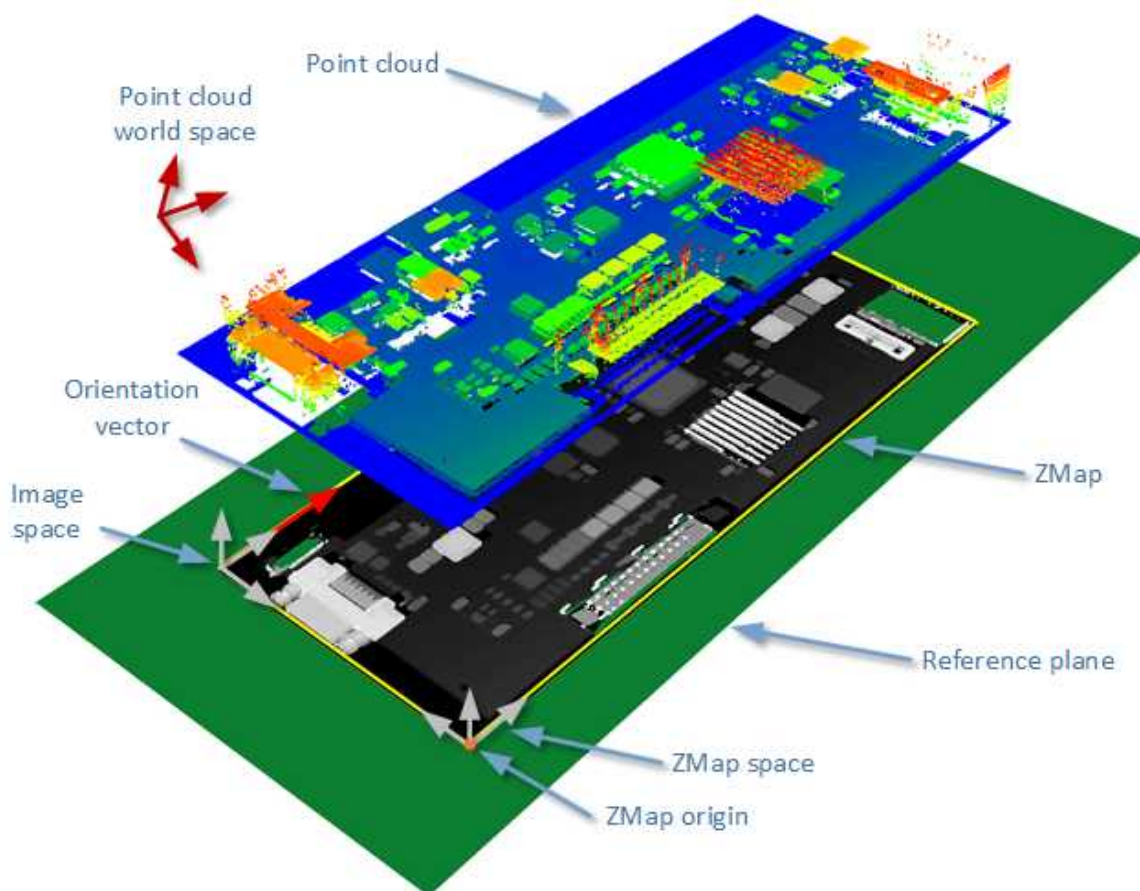
The orientation vector allows you to “rotate” the object around the normal of the reference plane.

- [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.

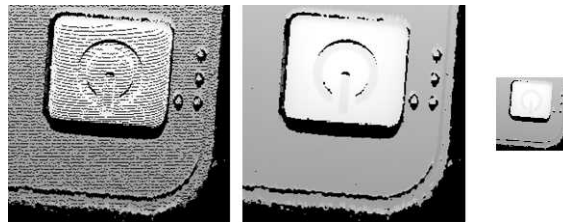


The projection of a point cloud on a ZMap, showing 3 coordinate systems: the world space, the ZMap space and the pixel space.

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.
- 3 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.
 - Whether to compute the normals of the points of the point cloud during the conversion.
This is not done by default.

Managing the Coordinates

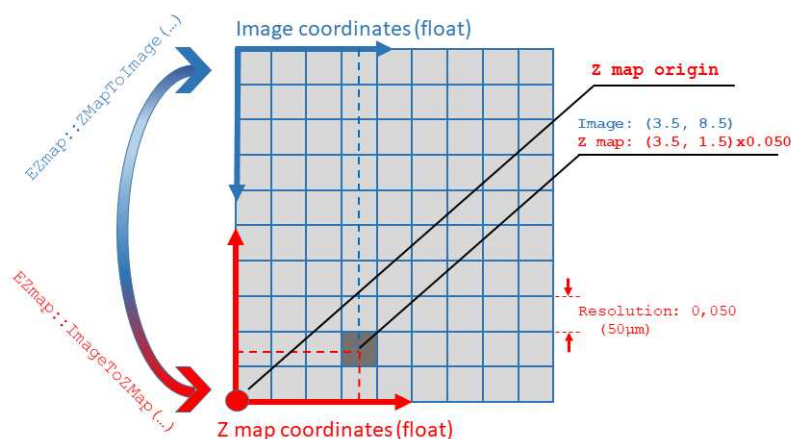
Coordinate systems on a ZMap

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.
- [GetWorldPositionFromMapPosition](#) is a method to get a 3D world position corresponding to a ZMap 2D coordinate. The world position is in the original point cloud space. If the 2D coordinate is undefined, an exception is thrown.

1.6. 3D Viewer

The class [E3DViewer](#) is an interactive 3D viewer for point clouds, ZMaps, meshes and basic shapes (lines, planes, boxes and spheres).

- 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, void* parent)`
- The way to use it depends on the Operating System and the User Interface API.

Windows only

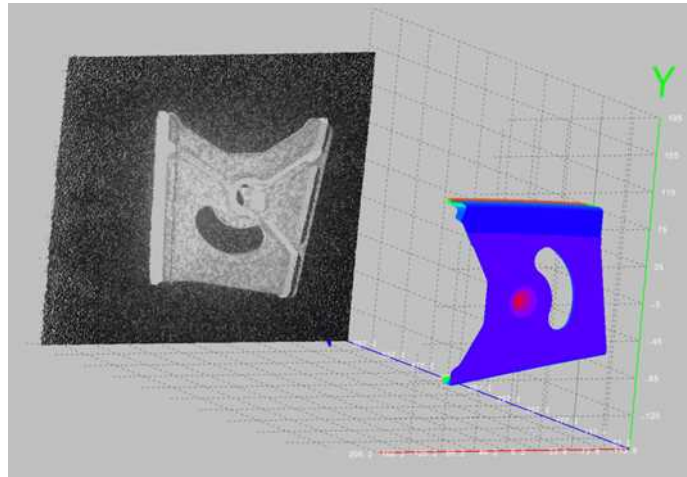
- To create a 3D viewer stand-alone window:
 - Use `E3DViewer(EUIAP::EUIAPI_Win32, orgX, orgY, width, height)`
- To create a 3D viewer as a part of another window in an MFC application:
 - Use `E3DViewer(EUIAP::EUIAPI_Win32, orgX, orgY, width, height, handle to the parent window)`
 - See the MsVc or MsVcs Easy3DViewer 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 Easy3DViewer Qt sample.

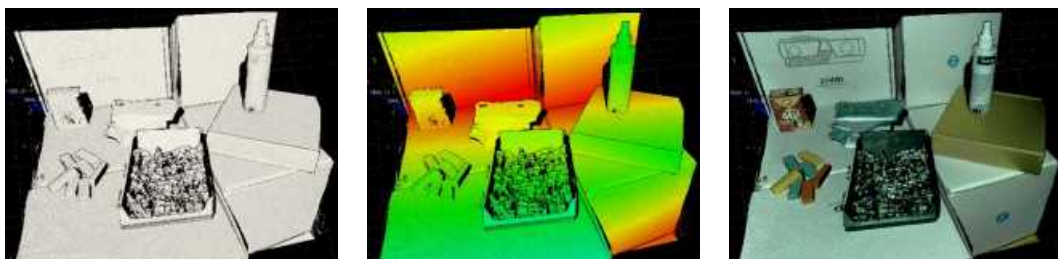
Managing the render sources

- A render source is a displayed entity. It can be an [EPointCloud](#), an [EZMap](#), an [EMesh](#), an [E3DBox](#), an [E3DLine](#), an [E3DPlane](#) or and [E3DSphere](#). You can display one or several render sources simultaneously in the 3D viewer.

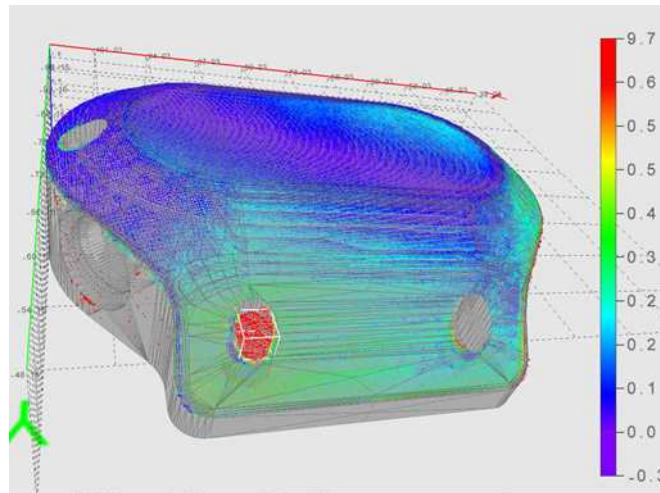


A point cloud displayed in gray scale and a mesh in false colors in the 3D viewer

- 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 (the only option for basic shapes), color ramp or point cloud color attributes.
 - *Color* ([SetRenderSourceConstantColor](#)) for render sources with constant color.
 - *Opacity* ([SetRenderSourceOpacity](#))
 - *Point size* ([SetRenderSourcePointSize](#)): applies to point clouds and ZMaps only.
 - *Wire frame* ([SetRenderSourceWireFrame](#)): applies to meshes, boxes and planes.



A point cloud displayed with constant color, color ramp or color attributes (data courtesy of Zivid)



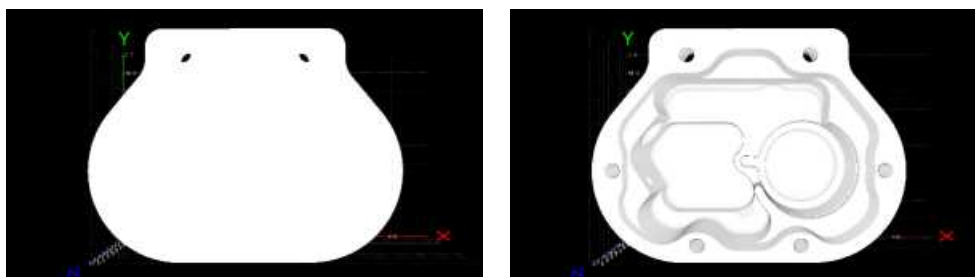
A mesh with wireframe and transparency, combined with a point cloud with color ramp

Shading

E3DViewer can shade rendering sources using a custom version of the *Eye Dome Lighting (EDL)* technique.

- Use `SetEnableEDLShading` to enable the shading.
- Use `SetEDLShadingFactor` to adjust the shading between 0 and 1:
 - 0 means no shading.
 - 1 means strong shading and 1, respectively meaning no shading and strong shading.

EDL is a post processing technique. It impacts all opaque render sources all together. With EDL, pixels that are closer to the camera occlude neighbor pixels that are further away from the camera.

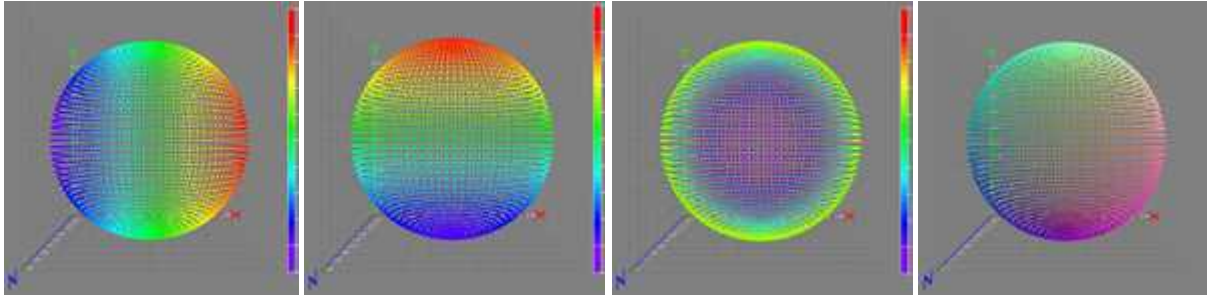


A mesh without and with EDL enabled

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.

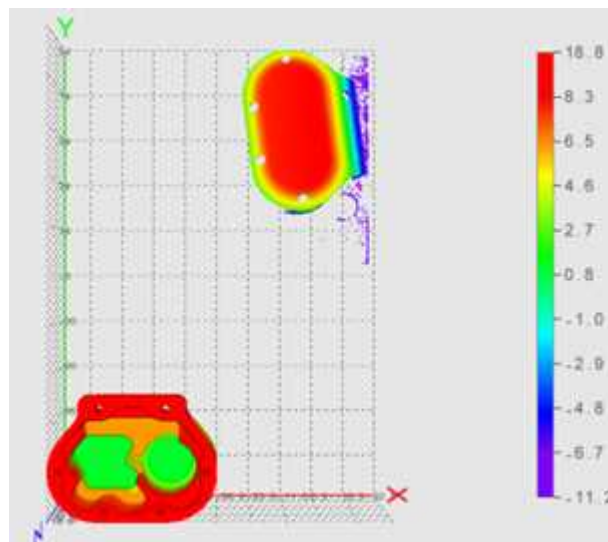
- [EColorRampMode_HueFromIntensity](#) computes the colors from the intensity attribute of the point.
- [EColorRampMode_HueFromNormal](#) computes the colors from the normal attribute of the point.
- [EColorRampMode_HueFromConfidence](#) computes the colors from the confidence attribute of the point.
- [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.

To keep the same color ramp bounds instead of adaptive ones, use [SetFixColorRampBounds](#).



- 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 [Easy3DViewer](#) sample for a use case of this view control API.

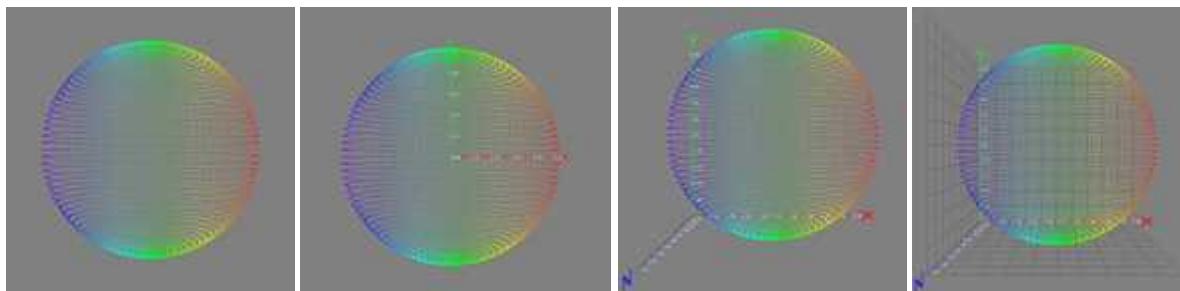
- You can also directly configure the view position with the methods:
 - [SetViewTarget](#) (by default, it is the center of the object).
 - [SetViewAngle](#) to choose the orientation of the view.
 - [SetViewDistance](#) to choose the distance to the view target.

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](#).

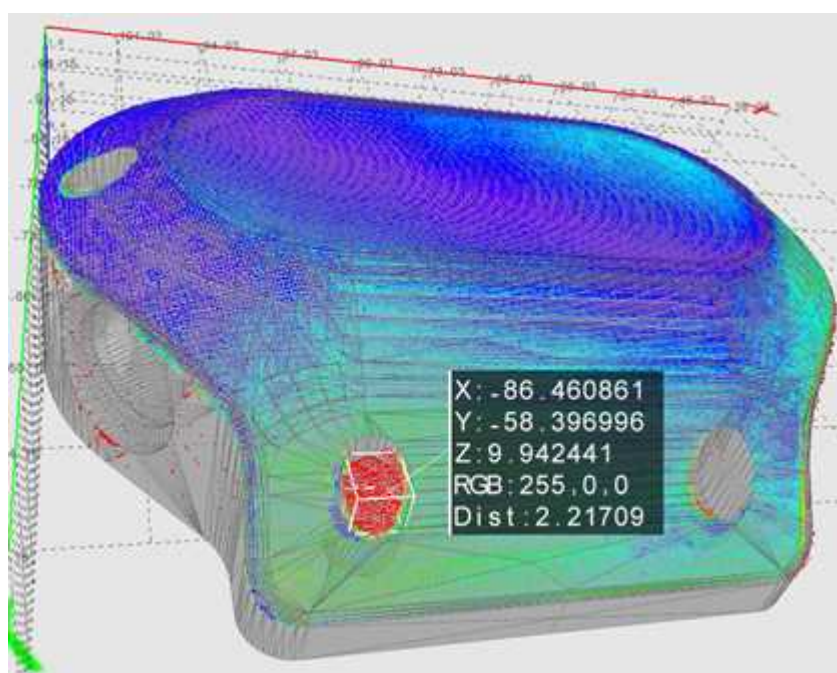
- Activate an auto rotate animation with [SetAutoRotate](#).
- Use a decimation level (remove some points to speed up the rendering) with [SetRenderDecimationLevel](#).



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.



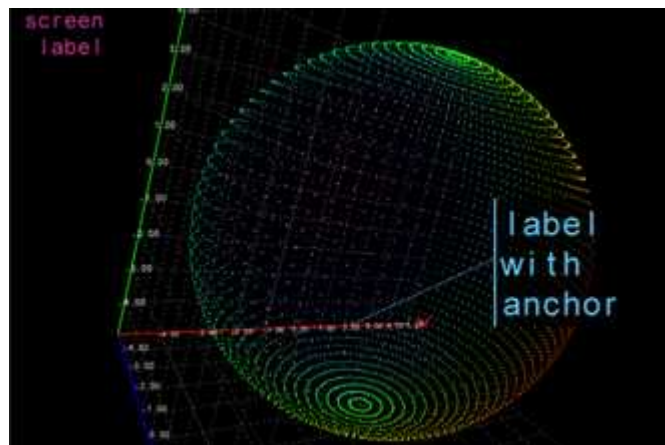
Displayed coordinates and attributes of a picked point on the 3D view

- 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 + mouse button**.
 - To control the picking, use the methods [Pick3DPoint](#), [GetLastPickedPoint](#) and [ResetPicking](#).
 - To configure the [E3DViewer](#) to call a specified function when you pick a point, use the method [SetPickedPointCallback](#).
- 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.



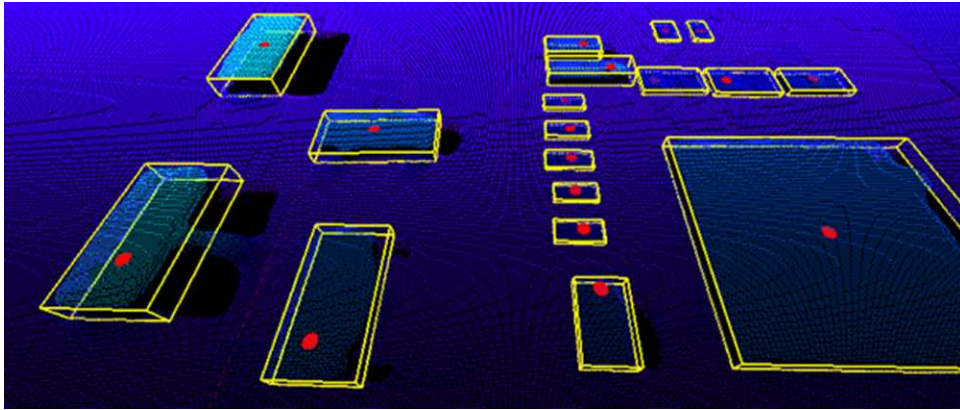
A screen label in the top left corner and a text label with 3D anchor

- 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 [Easy3DObjectExtract](#) MSCV sample as an example for the display of [E3DObjects](#).



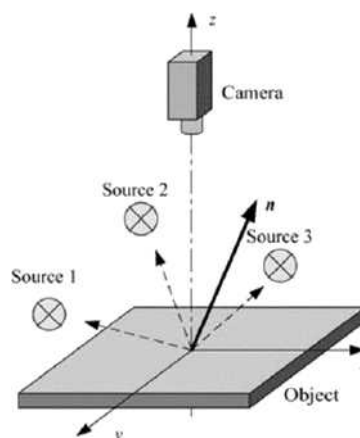
3D objects drawn with a point cloud displaying the bounding boxes and the top positions

1.7. Photometric Stereo

Photometric Stereo and Process

Introduction

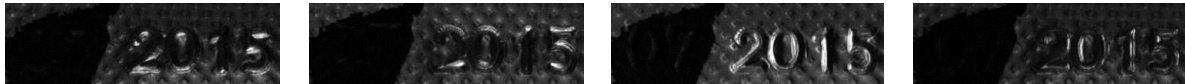
The Photometric Stereo is a technique used to estimate the normals at the surface of an object.



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. You can also integrate the normals to compute the height map.



Outputs: gradients X - gradients Y - Gaussian curvatures

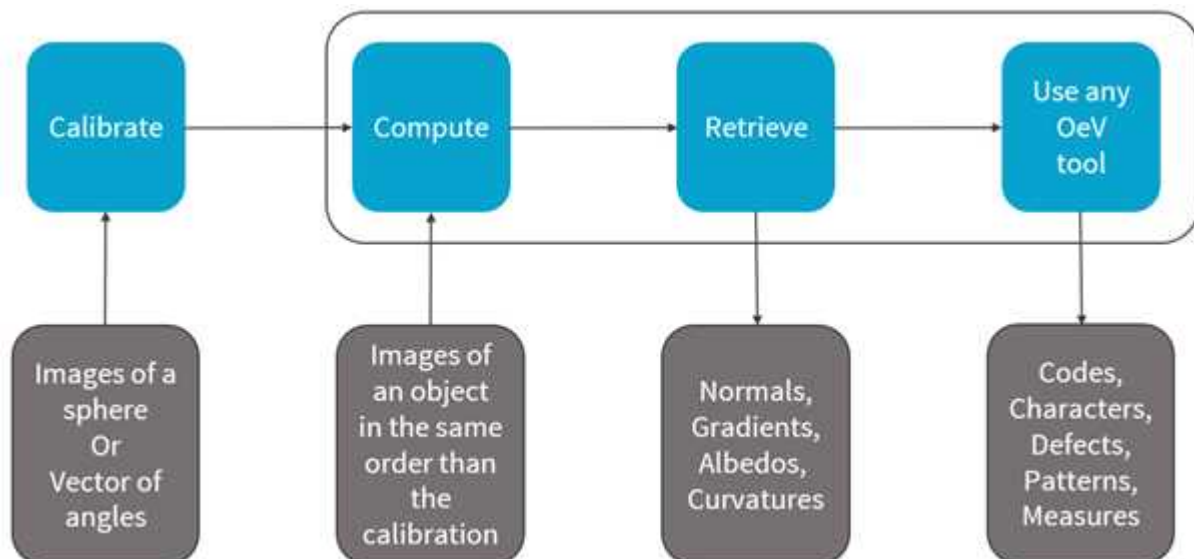


Outputs: mean curvatures - height map

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\\3D Processing\\Easy3DPhotometricStereo.
- 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_SphereDetectionFailed`).

In that case, you can pass the position of the circle to the method:

```

ECircle circle;

// Define circle (Todo)

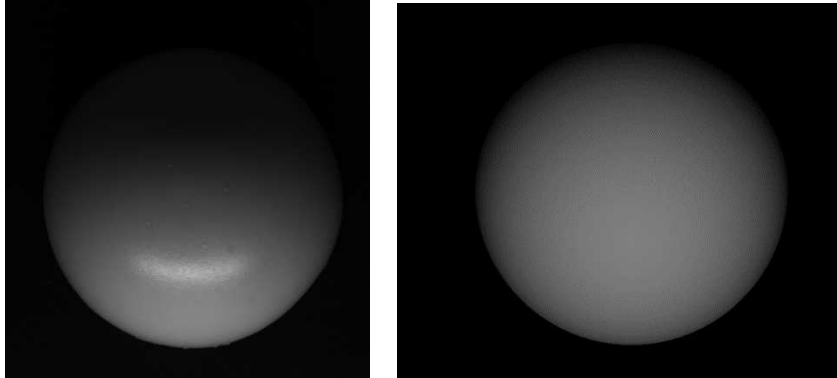
photometricStereo.CalibrateFromSphere(calibrationROIs, circle)

```

- 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 (ms)
4	4096 × 3072	1	479
		4	337
4	1024 × 768	1	52
		4	42

↪ / 1.4

↪ / 1.2

↪ / 11

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();

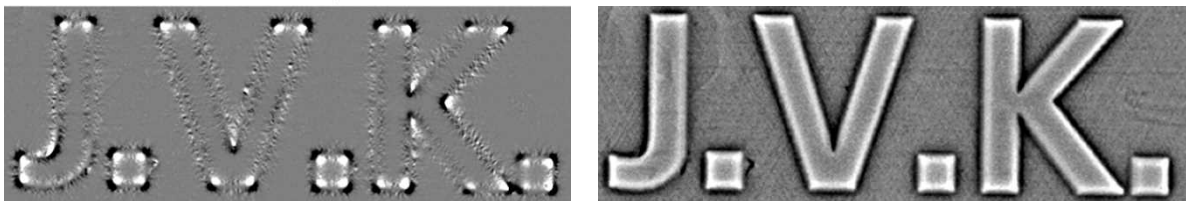
EImageBW8 gaussianCurvatures = photometricStereo.ComputeGaussianCurvatures(Easy3D::EPhotometricStereoContrast_HighContrast);

EImageBW8 meanCurvatures = photometricStereo.ComputeMeanCurvatures(Easy3D::EPhotometricStereoContrast_HighContrast);

EZMap8 heightMap = photometricStereo.ComputeHeightMap();
    
```

- 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

- Each pixel of the height map [EZMap8](#) represents the height of the object.
 - The pixel furthest from the camera has a height of 1.
 - The pixel closest to the camera has a height of 255.

[GetAlbedos](#), [ComputeMeanCurvatures](#), [ComputeGaussianCurvatures](#), [ComputeHeightMap](#)

Most of the computations are done in the method [Compute](#), however:

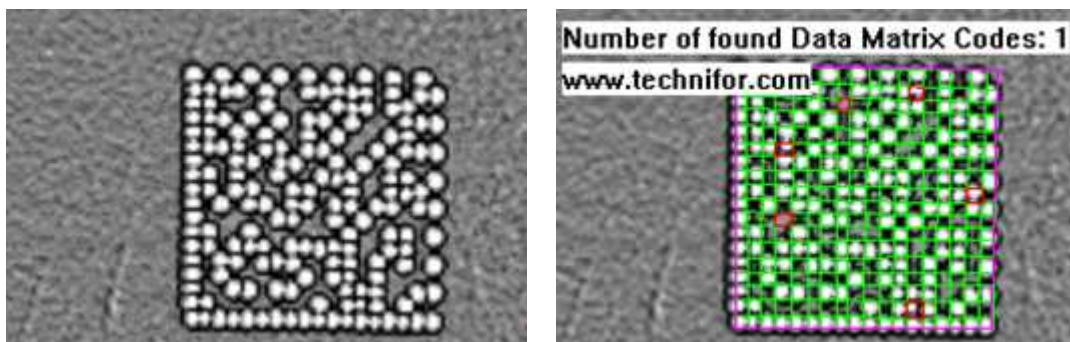
1. The last computations required for the curvatures and the height map are only performed when retrieving them to avoid computing them unnecessarily.
 - This is why these methods are named `ComputeXXX` instead of `GetXXX`.
 - Intermediary results common to the mean and the Gaussian curvatures are cached 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**.

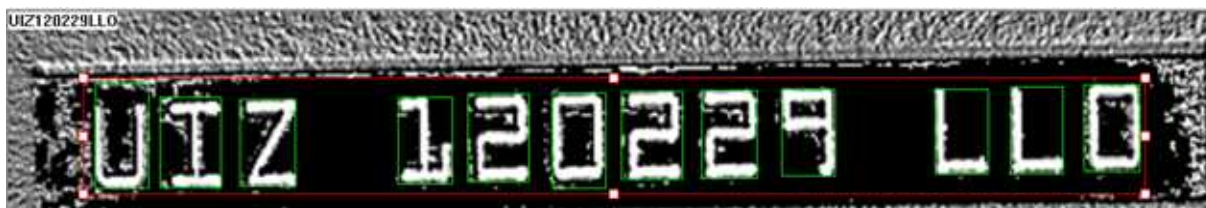
Here are some examples:

- Reading an embossed code with **EasyBarCode2**, **EasyMatrixCode** and **EasyQRCode**.



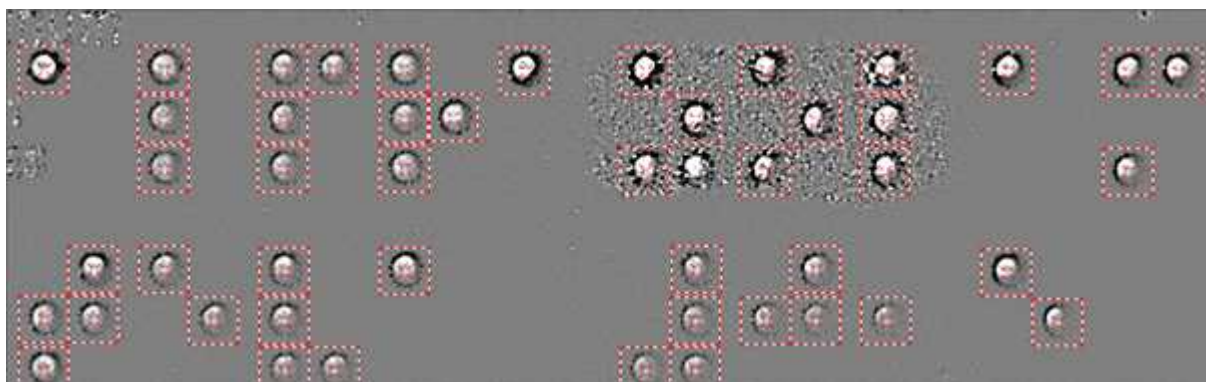
Mean curvatures of an engraved matrix code and EasyMatrixCode2 reading

- Reading an engraved text with **EasyOCR2**.



EasyOCR2 reading on the mean curvatures

- Finding patterns on embossed surfaces with **EasyFind**, **EasyMatch** and **EasyObject**.



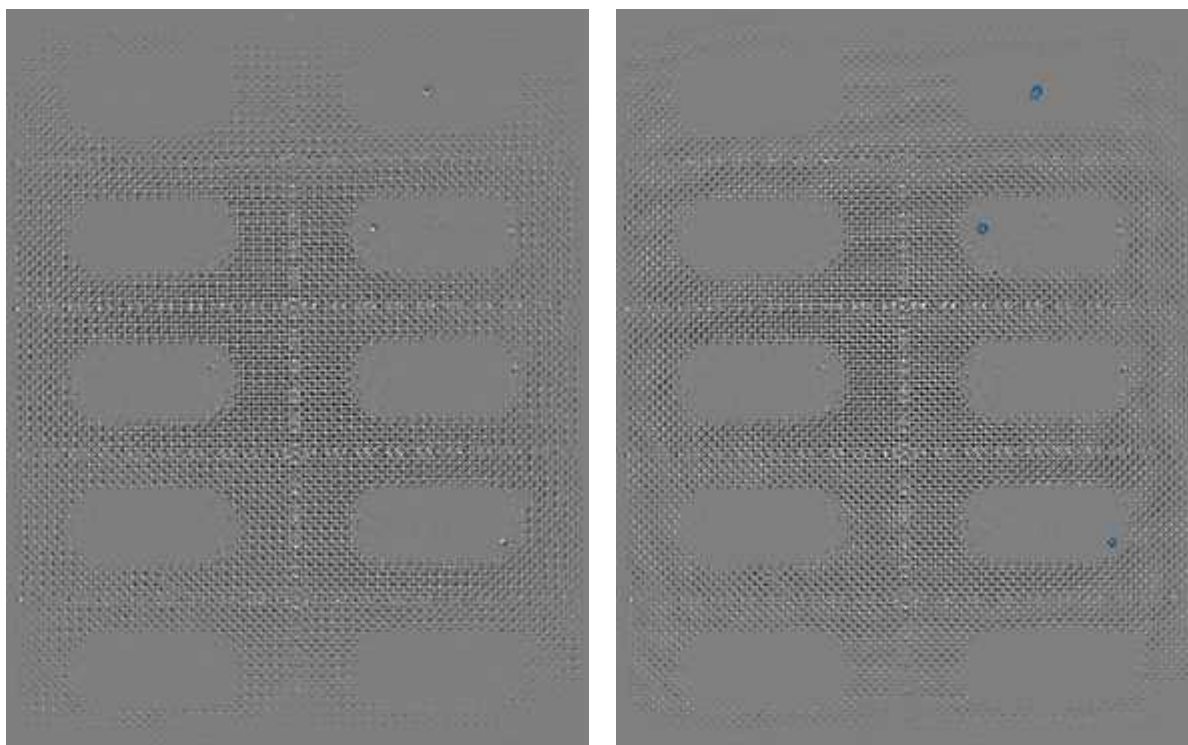
EasyFind results on the Gaussian curvatures to detect braille characters

- 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

Optimizing your Setup

Photometric stereo is based on two main assumptions:

1. The light has the same intensity and direction for each pixel of the image.
2. The object is mostly matte and not too specular.
 - Matte surfaces send back light in all directions (wood is a good example).
 - Specular surfaces send all the light in a single direction (mirrors are a good example).

While these conditions are never met in practice, deviating too much from them often causes poor results.

Uniformity of the lights

Ideally, all pixels of interest should be lit with the same intensity and from the same direction.

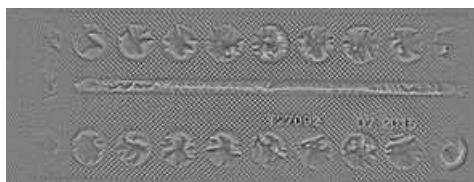
To achieve this:

- We recommend using bar lights or ring lights designed for photometric stereo.
- They should be larger than the objects you want to acquire.

NOTE: The sample images were taken with a bar light of around 15 cm in length and a ring light of around 15 cm of diameter. Both gave similar results.



- The image of the sheet of paper is whiter on the left when, ideally, it should be uniformly white.



- If you are interested in curvatures, this does not prevent acquiring good images, while the albedos and normals images are more affected by this.



- In the mean curvature and albedo image above, you see artifacts on the left and on the right of the object, while not at the top and the bottom. This is because the object is large and not so high, so the lights are mostly uniform on the object when coming from the top or bottom but not when coming from left or right.
- It can be alleviated by using Flat Images, as shown in the next section but it does not solve the problem completely.

**TIP**

In practice, you can measure the uniformity of your lights by acquiring an image of a matte sheet of paper.

Specularity of the object

Photometric Stereo works better on matte surfaces as the reflection of the source light by the object causes artifacts. Please note however that a little reflection is not necessarily a problem. This is something to keep in mind when analyzing results.

This also applies to the calibration (hemi-)sphere. The current calibration algorithm is designed to handle a bit of reflection but too much of it induces calibration errors. Nevertheless, though there is reflection in our sample images, the calibration still works on them.

**TIP**

We recommend using a wooden hemisphere or one covered in matte paint. Alternatively, you can measure and specify your angles manually as no loss of quality is observed in the images in that case.

Improving the Results

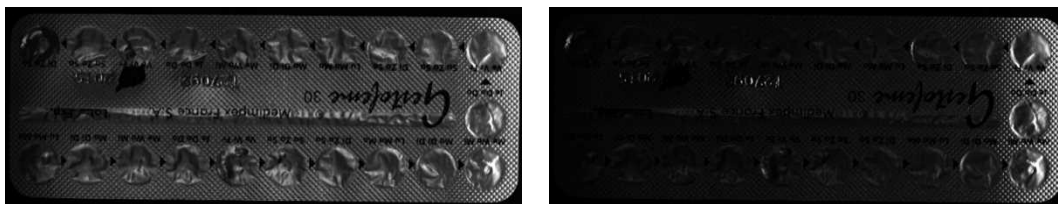
Using a dark image to account for ambient lighting

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;

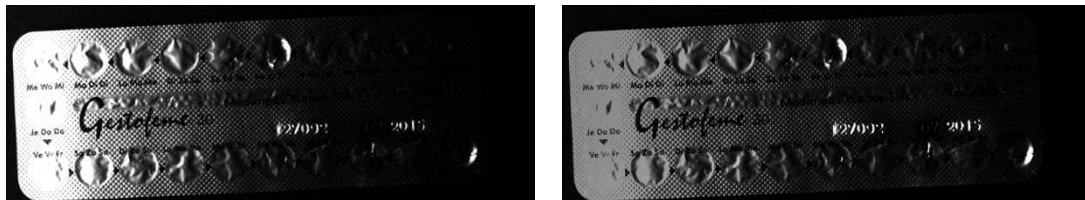
// Load flat images in the same order than the calibration images/angles (Todo)
```

```
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



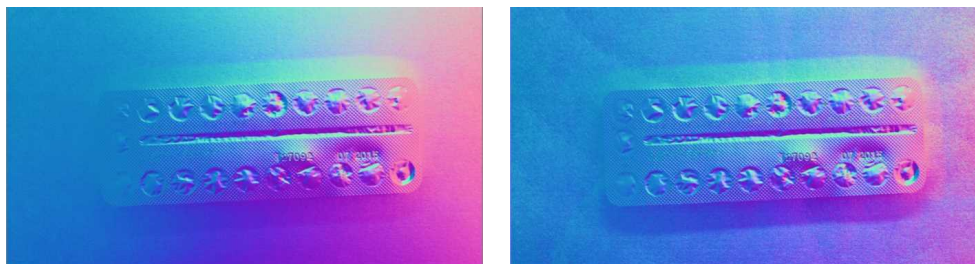
The object image: raw (left) and after correction with the flat image (right)

- 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.



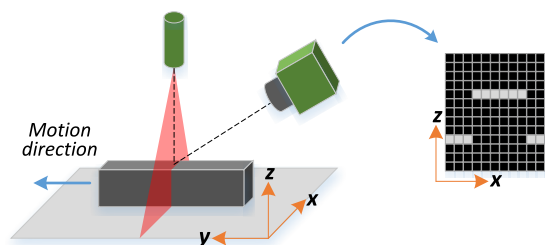
TIP

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.

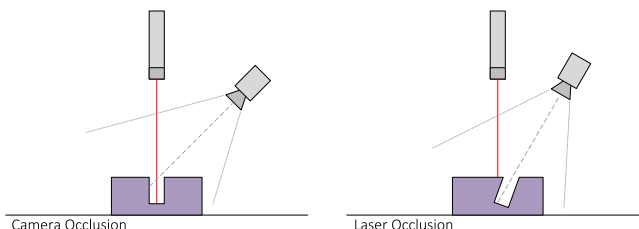
From the scanning you can reconstruct its 3D shape.



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

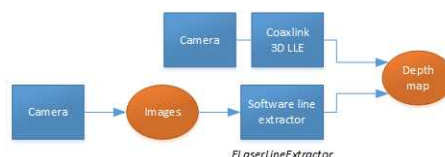
You can limit or avoid occlusions by using advanced scanning methods, for example by using two cameras or two lasers.

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.

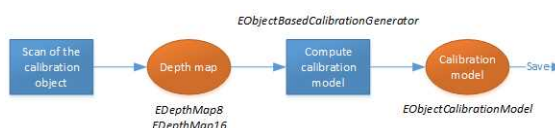
The source material for 3D processing is the depth map, coming from a Coaxlink Quad 3D-LLE or generated from a list of images.

3 types of depth map are available, one for each different pixel coding scheme (8, 16 or 32 bits).



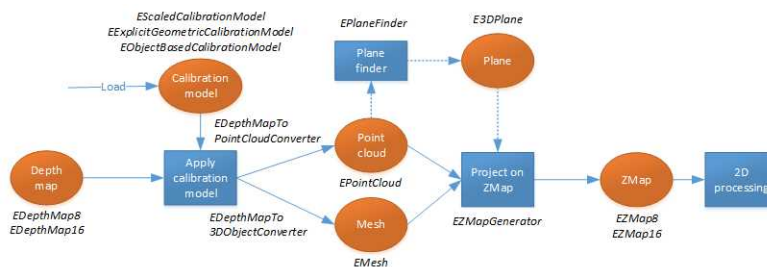
The generation of a depth map, from a hardware or a software source

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.

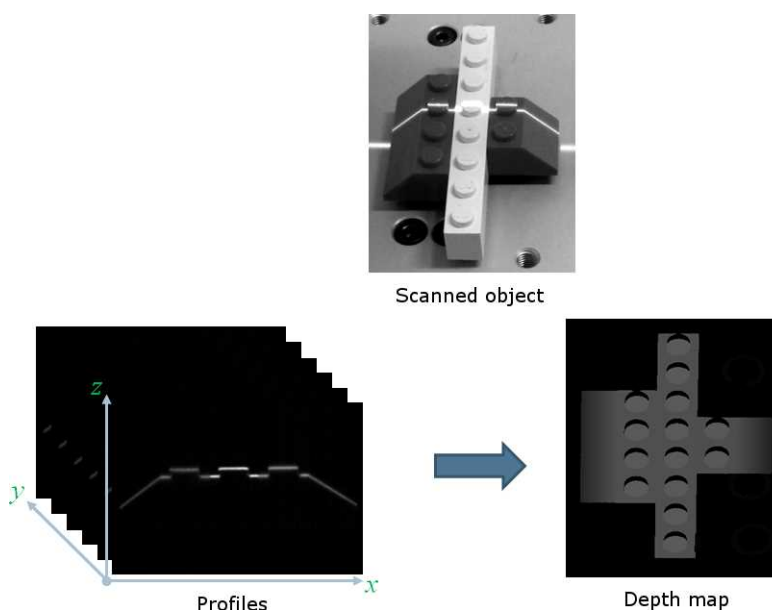
2.3. Laser Line Extraction

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 class `ELaserLineExtractor` provides the laser line extraction functionality in **Open eVision**.

Uses the method `ELaserLineExtractor.AnalysisMode` to select one of the following algorithms to extract the laser line (see below for more details):

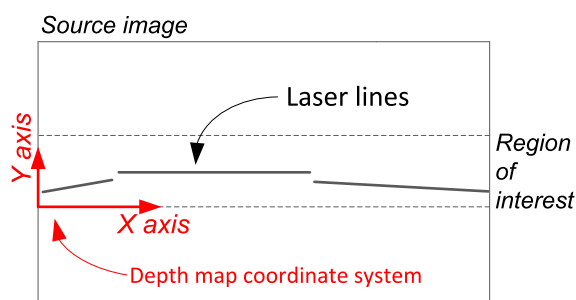
- **Maximum detection** returns the position of the pixel of maximum intensity. It's the fastest method but it doesn't support sub-pixel precision.
- **Peak detection** approach detects local maxima. If several maxima are detected, the one with the highest intensity is returned. The position is returned 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.

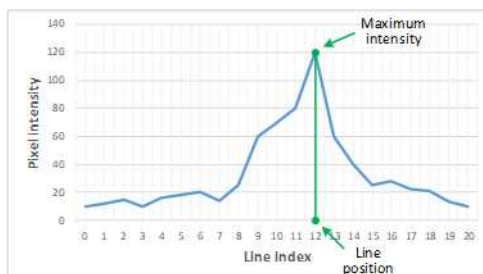
The line position returned by the laser line extraction algorithms is relative to the bottom of the region of interest. So, values in the depth map range from 0 (bottom of the ROI) to the height of the ROI.



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:

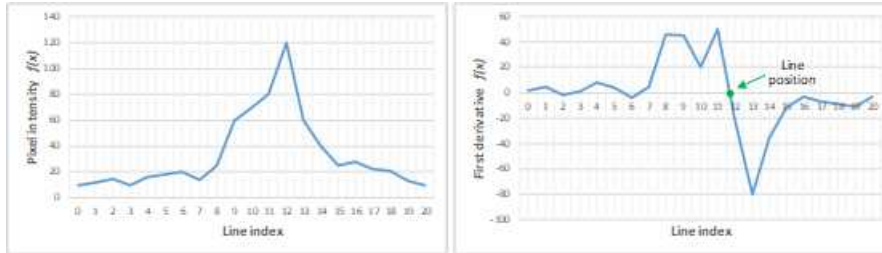
- A low-pass filter to reduce the high frequency variations in the image.
- A threshold to eliminate the background noise from the sensor.

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.

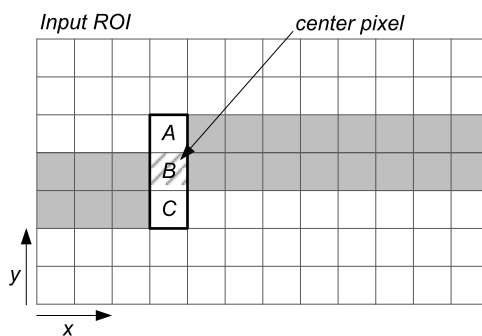


Center of gravity on a ROI profile

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)`.

Depth map Z resolution

- As explained above, the laser line extraction computes the sub-pixel position of the laser line profile for each column of the region of interest.
 - This position is encoded in a 16-bit depth map ([EDepthMap16](#)).
 - The resolution controls the way the sub-pixel positions are converted to 16-bit fixed point values (that is how many bits are allocated to represent the fractional part of the sub-pixel position).
 - On a [EDepthMap16](#), use the method [EDepthMap16.ZResolution](#) to retrieve the resolution. It is a floating-point value used to convert the integer pixel value to a real depth value.
- By default, [ELaserLineExtractor](#) computes the best resolution depending on the height of the region of interest. The following table lists typical resolution values depending on the height of the processed frame or region of interest.

Frame or ROI height	Bits used for the fractional part	Z resolution
100	8	1/256 = 0.00390625
200	8	1/256 = 0.00390625
400	7	1/128 = 0.0078125
800	6	1/64 = 0.015625
2500	4	1/16 = 0.0625

- Use the optional parameter `zResolution` of the constructor [ELaserLineExtractor](#) to fix the Z resolution.
 - If you do not set the parameter `zResolution`, an algorithm computes the optimal value. Use [ELaserLineExtractor.DepthMap](#) to query the returned depth map and get the effective resolution.
- If you are using the hardware laser line extraction with the **Coaxlink Quad 3D-LLE**, the fixed point format is set by the configuration: `8_8` or `11_5` for 8- or 5-bit fractional part.

 See documentation.euresys.com/Products/COAXLINK/COAXLINK/en-us/Content/03_Interfaces/functional-guide/lle/LLE_Processing_Core_Characteristics.htm

2.4. Software vs Hardware Line Extraction

Hardware line extraction on a Coaxlink

- The **Coaxlink Quad 3D-LLE** frame grabber features hardware line extraction. The performances are as follows, for all methods - Maximum detection (Max), Center of gravity detection (CoG) and Peak detection (Peaks):
 - 9500 profiles/s for a 2048×256 or a 4096×128 region
 - 19000 profiles/s for a 2048×128 region
 - 38000 profiles/s for a 1024×128 region
 - 76000 profiles/s for a 1024×64 region
- As a result, the hardware-based line extraction is $2\times$ to $15\times$ faster than the software implementation in **Open eVision** running on an **Intel Core i7-10850H** CPU at 2.70 GHz (see below).

Software line extraction on an Intel CPU

- The tables below present benchmarks for software line extraction for various region sizes (in pixels) and configurations. The extraction speed is expressed in profiles per second.

The measurement setup is:

- Intel Core i7-10850H CPU at 2.70 GHz
- Generation of 16-bit depth maps
- Methods used: Maximum detection (Max), Center of gravity detection (CoG) and Peak detection (Peaks)

Using 1 thread, without low pass filter

Region size	Max	COG	Peaks
3072 × 512	368	298	248
3072 × 256	752	593	497
3072 × 128	1488	1179	966
3072 × 64	3764	2723	2169
1024 × 512	1142	891	771
1024 × 256	2226	1777	1479
1024 × 128	4266	3324	2752
1024 × 64	11130	8533	6400

Using 1 thread, with low-pass filter

Region size	Max	COG	Peaks
3072 × 512	241	200	183
3072 × 256	468	403	357
3072 × 128	920	790	703
3072 × 64	2169	1765	1610
1024 × 512	707	600	561
1024 × 256	1406	1219	1057
1024 × 128	2723	2370	2048
1024 × 64	6400	5333	4830

Using 2 threads, without low-pass filter

Region size	Max	COG	Peaks
3072 × 512	731	576	474
3072 × 256	1406	1094	941
3072 × 128	2813	2226	1868
3072 × 64	7314	5224	3240
1024 × 512	2000	1718	1446
1024 × 256	4129	3368	2844
1024 × 128	8533	6564	4654
1024 × 64	21333	15058	10666

Using 4 threads, without low-pass filter

Region size	Max	COG	Peaks
3072 × 512	1108	885	703
3072 × 256	2438	1765	1273
3072 × 128	4740	3459	2639
3072 × 64	9481	7757	6564
1024 × 512	3605	2694	2031
1024 × 256	6400	5446	4063
1024 × 128	13473	11636	9846
1024 × 64	36571	19692	19692

Software line extraction on a Jetson Nano

- The tables below present benchmarks for software line extraction for various region sizes (in pixels) and configurations. The extraction speed is expressed in profiles per second.

The measurement setup is:

- Jetson Nano (4 available cores)
- See developer.nvidia.com/embedded/jetson-nano
- Generation of 16-bit depth maps
- Methods used: Maximum detection (Max), Center of gravity detection (CoG) and Peak detection (Peaks)

Using 1 thread, without low pass filter

Using 1 thread, with low-pass filter

Region size	Max	COG	Peaks
3072 × 512	34	30	30
3072 × 256	126	106	102
3072 × 128	248	210	199
3072 × 64	479	409	374
1024 × 512	110	98	97
1024 × 256	383	321	308
1024 × 128	736	623	587
1024 × 64	1426	1229	1114

Region size	Max	COG	Peaks
3072 × 512	30	28	27
3072 × 256	112	94	93
3072 × 128	220	188	183
3072 × 64	413	356	333
1024 × 512	90	83	82
1024 × 256	337	285	280
1024 × 128	658	567	543
1024 × 64	1226	1065	987

Using 2 threads, without low-pass filter

Region size	Max	COG	Peaks
3072 × 512	61	56	56
3072 × 256	206	180	181
3072 × 128	423	367	365
3072 × 64	864	740	711
1024 × 512	185	170	168
1024 × 256	596	530	517
1024 × 128	1203	1065	1035
1024 × 64	2403	2106	2000

Using 4 threads, without low-pass filter

Region size	Max	COG	Peaks
3072 × 512	95	91	90
3072 × 256	257	234	240
3072 × 128	515	470	489
3072 × 64	1092	992	1029
1024 × 512	269	260	259
1024 × 256	689	678	675
1024 × 128	1383	1352	1356
1024 × 64	2820	2716	2745

Software line extraction on a Jetson AGX Orin

- The tables below present benchmarks for software line extraction for various region sizes (in pixels) and configurations. The extraction speed is expressed in profiles per second.

The measurement setup is:

- **Jetson AGX Orin** (12 available cores)
- 📄 See www.nvidia.com/en-il/autonomous-machines/embedded-systems/jetson-orin
- Generation of 16-bit depth maps
- Methods used: Maximum detection (Max), Center of gravity detection (CoG) and Peak detection (Peaks)

Using 1 thread, without low pass filter

Region size	Max	COG	Peaks
3072 × 512	222	192	181
3072 × 256	449	386	362
3072 × 128	898	766	710
3072 × 64	1796	1521	1391
1024 × 512	671	579	546
1024 × 256	1336	1155	1075
1024 × 128	2680	2301	2106
1024 × 64	5278	4511	4079

Using 1 thread, with low-pass filter

Region size	Max	COG	Peaks
3072 × 512	198	174	164
3072 × 256	401	351	332
3072 × 128	801	697	653
3072 × 64	1592	1380	1261
1024 × 512	600	525	498
1024 × 256	1197	1049	987
1024 × 128	2386	2085	1935
1024 × 64	4697	4096	3696

Using 2 threads, without low-pass filter

Region size	Max	COG	Peaks
3072 × 512	444	382	362
3072 × 256	897	772	723
3072 × 128	1790	1528	1416
3072 × 64	3543	3002	2745
1024 × 512	1338	1158	1088
1024 × 256	2659	2311	2142
1024 × 128	5224	4471	4112
1024 × 64	9941	8533	7529

Using 4 threads, without low-pass filter

Region size	Max	COG	Peaks
3072 × 512	876	763	720
3072 × 256	1790	1530	1436
3072 × 128	3447	3011	2745
3072 × 64	6649	5752	5069
1024 × 512	2687	2321	2178
1024 × 256	5251	4571	4213
1024 × 128	10138	8752	8000
1024 × 64	18618	16516	14422

2.5. 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 ([EExplicitGeometricCalibrationModel](#) class)
- Use the object-based calibration approach ([EObjectBasedCalibrationModel](#) class)

These models share the same base class [ECalibrationModel](#) and exposes the method [Apply\(\)](#), which is used to apply the conversion between a depth map pixel and a 3D point. It takes as input the coordinates of one point in a depth map and it returns the coordinates of the corresponding point in the 3D space.

The method [Apply](#) is not aware of the possible mirroring of the corresponding depth map and cannot make use of [EDepthMap::AxisSystemType](#) (see below). If necessary (when the corresponding depth map is vertically mirrored) the y coordinates should be flipped before calling the [Apply](#) method.

- The class [EDepthMapToPointCloudConverter](#) generates a point cloud from a depth map, using one of the calibration models.
- The class [EDepthMapToMeshConverter](#) generates a mesh from a depth map, using one of the calibration models.

By convention:

- The origin of the referential is the lower-left corner of the depth map.
- The center of the first pixel at the lower-left corner is at $x = 0.5$ and $y = 0.5$.
- The center of the pixel at the upper-right corner is at $x = \text{width} - 0.5$ 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

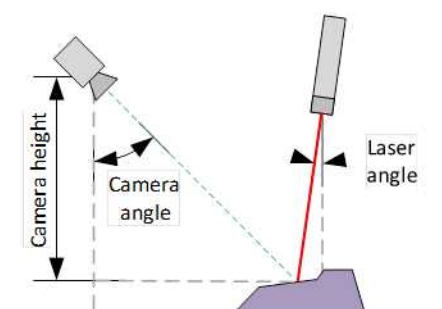
The explicit geometric model ([EExplicitGeometricCalibrationModel](#)) defines a simple and ideal laser triangulation setup. The explicit calibration makes some strong assumptions on the setup geometry and can only be used when a minimum set of parameters are known:

- 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).



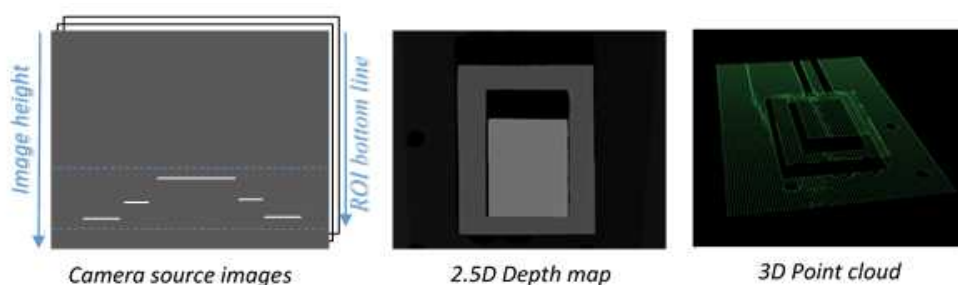
TIP

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 457 section.

2.6. 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.

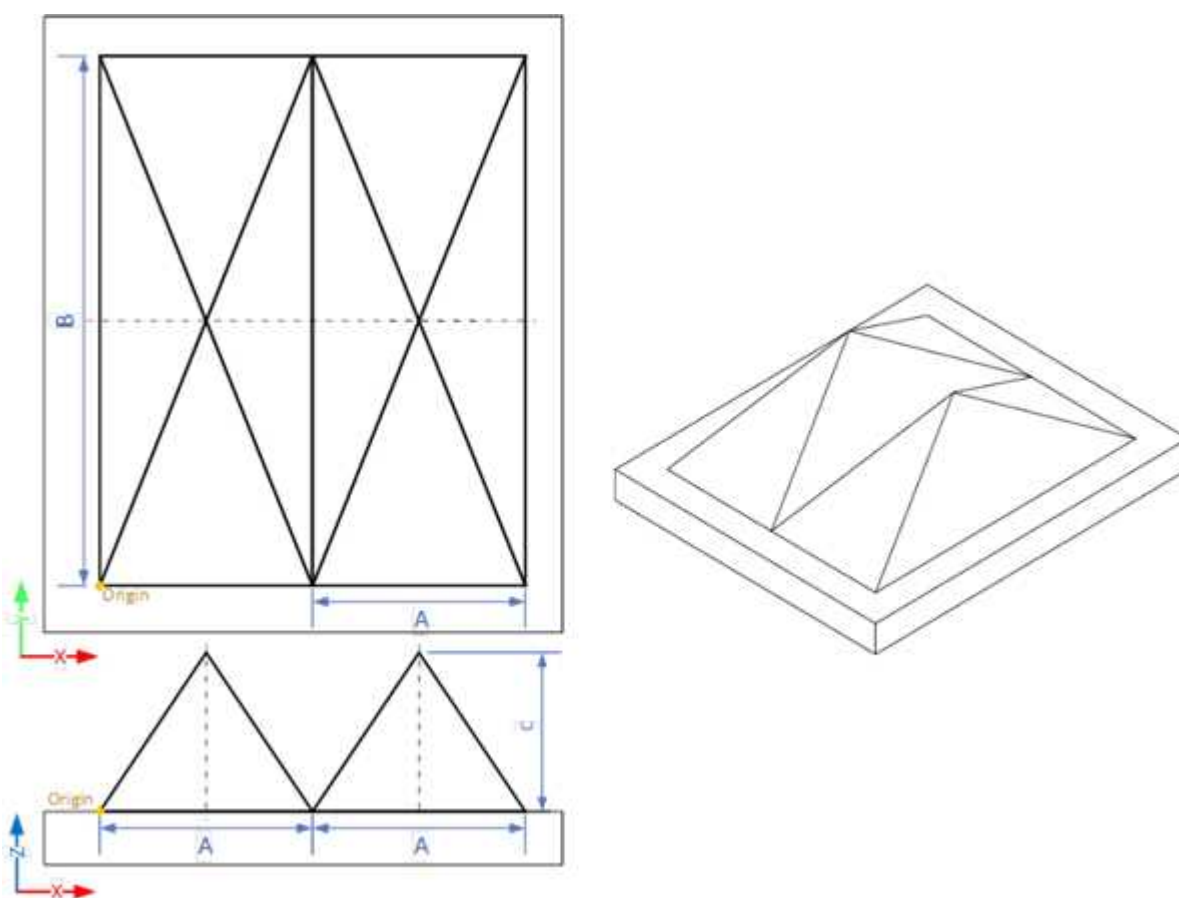
The double pyramid



TIP

In **Open eVision 2.7** the “double truncated pyramid” calibration object is recommended over the “double pyramid” model.

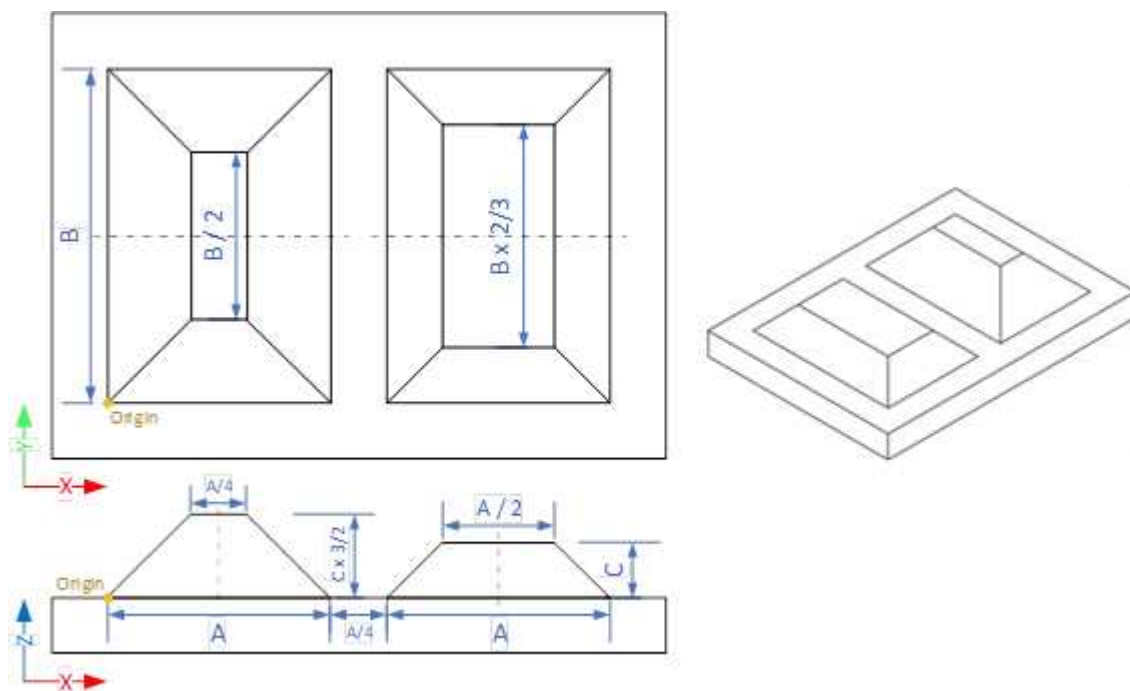
The dimensions of the “double pyramid” calibration object along the X-, Y- and Z-axes are named A, B and C respectively.



The "double pyramid" calibration model

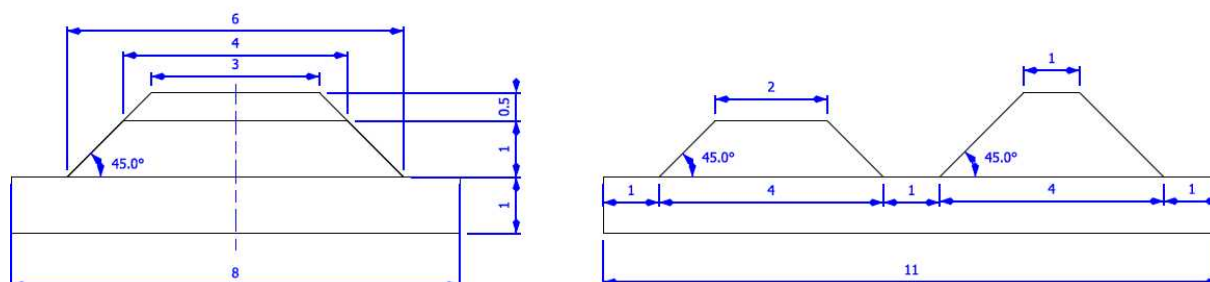
The truncated double pyramid

- 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

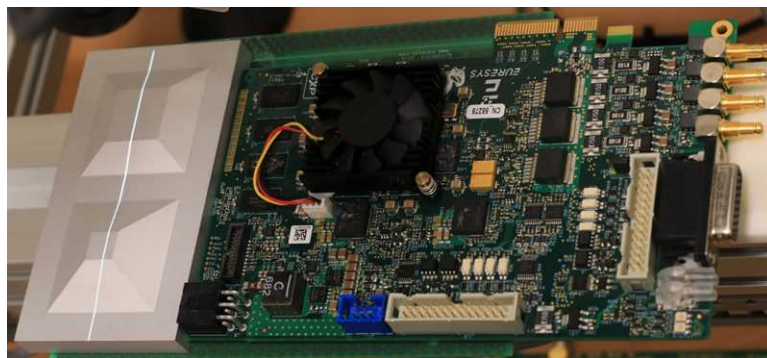
Overall dimensions

- 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.



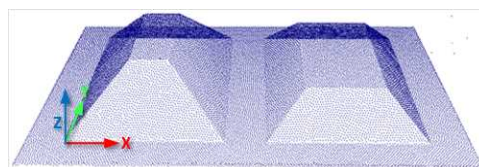
A PCB scanning setup with the associated calibration object
The calibration object dimensions (A, B and C) match the width and the height of the PCB



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.

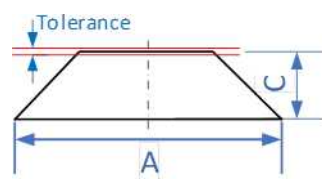


A calibrated point cloud with the origin and the axis of the coordinates system
The 3D origin is located at the external corner of the higher pyramid

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.

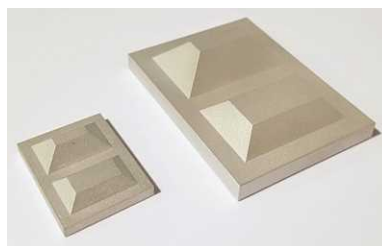
The build material and the surface finishing are also important and must have:

- 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 formats like STEP, OBJ and STL.

Download these files from the **Open eVision** download area in the **Additional Resources** section (www.euresys.com/Support).

OPEN EVISION		2.17	
	Download	File size	Operating system
Release Notes	open_evision_release_notes-2.17.2.1161.pdf Intermedica versions	1.1 MB	Windows
Documentation	View Open eVision 2.17 online documentation (including PDFs) open_evision-win-offline-documentation-en-2.17.1.1160.exe open_evision-linux-offline-documentation-en-2.17.1.1160.tar.gz open_evision-win-offline-documentation-en-cn-2.17.1.1160.exe open_evision-linux-offline-documentation-en-cn-2.17.1.1160.tar.gz open_evision-win-offline-documentation-en-jp-2.17.1.1160.exe open_evision-linux-offline-documentation-en-jp-2.17.1.1160.tar.gz open_evision-win-offline-documentation-en-ko-2.17.1.1160.exe open_evision-linux-offline-documentation-en-ko-2.17.1.1160.tar.gz	0.1 GB 0.2 GB 0.2 GB 0.3 GB 0.3 GB 0.3 GB 0.2 GB 0.3 GB	Windows Windows Linux Windows Linux Windows Linux Windows Linux
Setup Files	open_evision-win-2.17.2.13747.exe open_evision-linux-64-2.17.2.13749.deb.tar.gz open_evision-linux-64-2.17.2.13749.rpm.tar.gz open_evision-win-studio-2.17.2.13747.msi open_evision-win-deep-learning-studio-2.17.2.13747.msi open_evision-win-3d-studio-2.17.2.13747.msi open_evision-win-license-manager-2.17.2.13747.msi med-vela-license-manager-2.17.2.13747.exe reee-linux-license-manager-64-2.17.2.13749.deb.tar.gz reee-linux-license-manager-64-2.17.2.13749.rpm.tar.gz Intermedica versions	0.6 GB 0.5 GB 0.5 GB 0.2 GB 0.2 GB 0.1 GB 43 MB 45 MB 34 MB 41 MB	Windows Linux Linux Windows Windows Windows Windows Windows Linux Linux
Additional Resources	Deep Learning Additional Resources 2.17.2.13747.zip Easy 3D Calibration Models 2.17.0.13019.zip Easy 3D Sensors Compatibility 2.17.2.13747.zip Intermedica versions	1.3 GB 0.4 MB 14 MB	Windows Windows Windows

Download the calibration object models

Scanning the calibration object

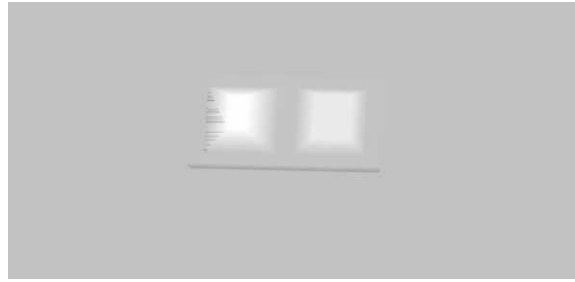
- The scan of the calibration object produces a depth map.
- To ensure a correct detection of the calibration object and a precise calibration model, you must fulfill the following criteria:
 - All faces of the calibration object must be visible on the depth map (this affects the orientation of both 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

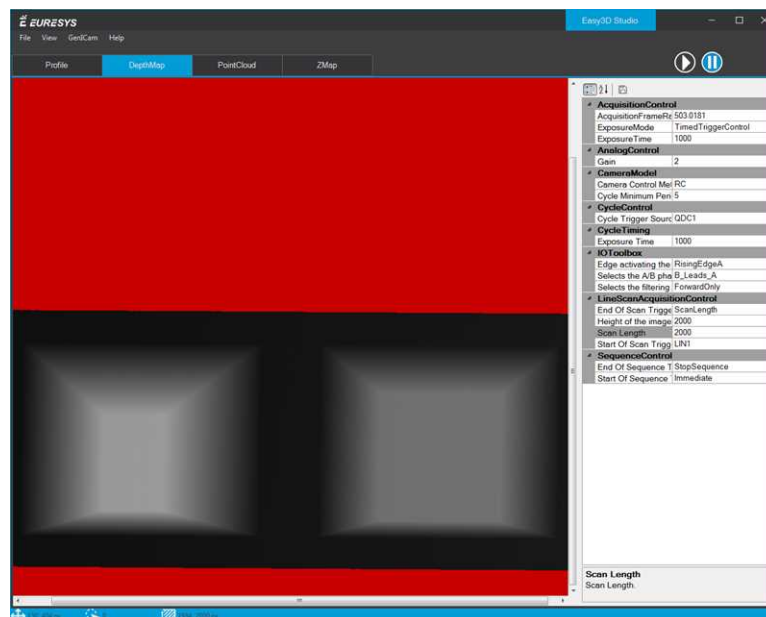
Calibration with Easy3D Studio

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:

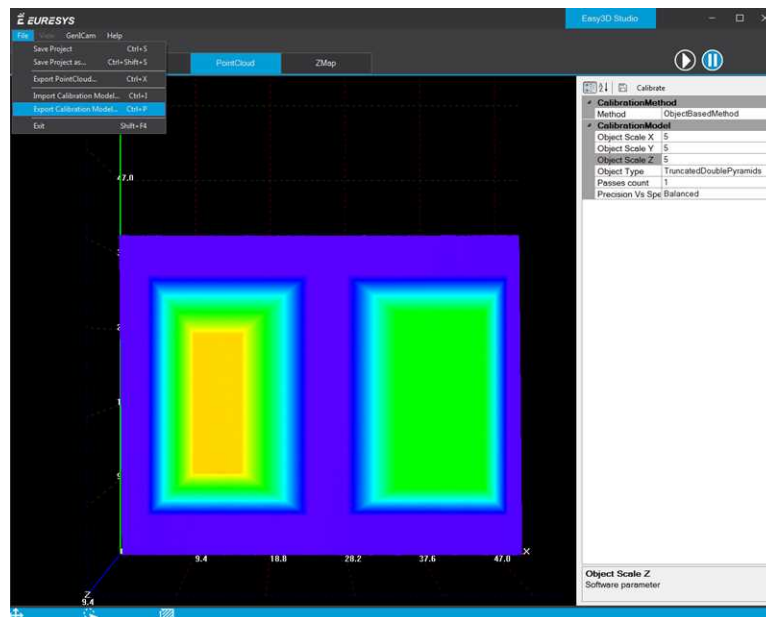
- The scanned image.
- The acquisition parameters on the right side.



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
//
// EDepthMap16 depth_map;
depth_map.LoadImage("ctx1 calibration object.png"); // from Easy3D sample images
depth_map.SetZResolution(1.f / (1 << 5)); // 11.5 fixed point pixel format

// Initialize the calibration generator
EObjectBasedCalibrationGenerator calib_generator;
calib_generator.SetCalibrationObjectType(EObjectBasedCalibrationType_TruncatedDoublePyramid, 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)
EDepthMap16 depth_map;
depth_map.LoadImage("ctx1 shapes.png");
depth_map.SetZResolution(1.f / (1 << 5));

// Initialize a converter, use the loaded model
EDepthMapToPointCloudConverter converter;
converter.SetCalibrationModel(calib_model);

// Convert the depth map to a metric point cloud and save it
EPointCloud point_cloud;
converter.Convert(depth_map, point_cloud);
point_cloud.SavePCD("point_cloud.pcd");

```

To experiment and learn about the **Easy3D** calibration, a C++ sample called `Easy3DLaserLineCalibration` 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 `ED3DObject` 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.

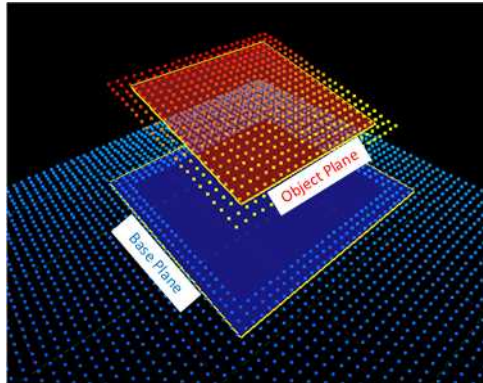
Object plane and base plane

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`

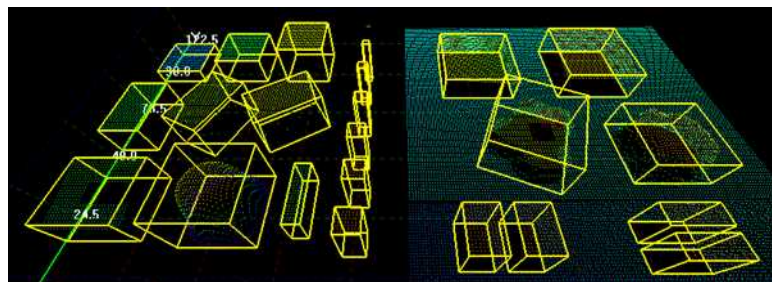
- This plane is called the *base plane*.
- It is an estimation of the local background around the object.
- If there are too many undefined pixels in this area, the base plane is equal to the reference plane of the input `EZMap`.



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.

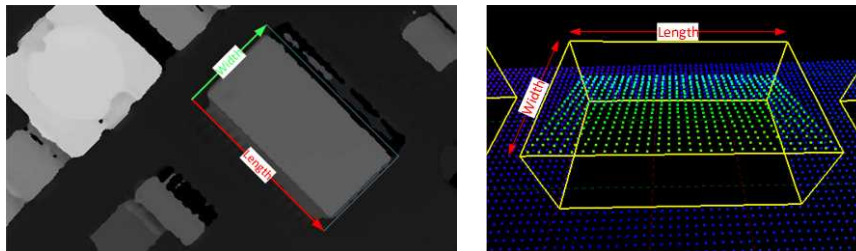


Length and width

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.



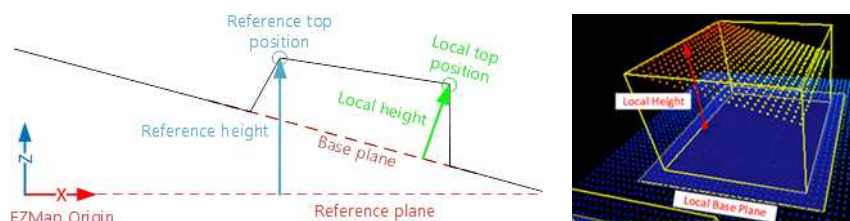
Local and reference top positions and heights

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.

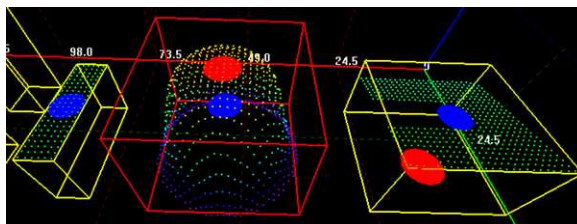
Use the `E3DObjectExtractor.LocalHeightRange` and the `E3DObjectExtractor.ReferenceHeightRange` accessors to set the ranges of allowed values for the local and the reference height.

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.

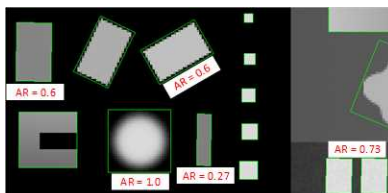


Aspect ratio

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.

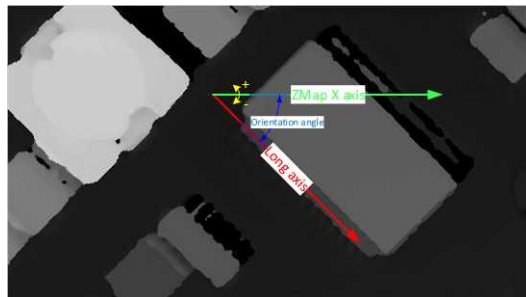


Orientation angle

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

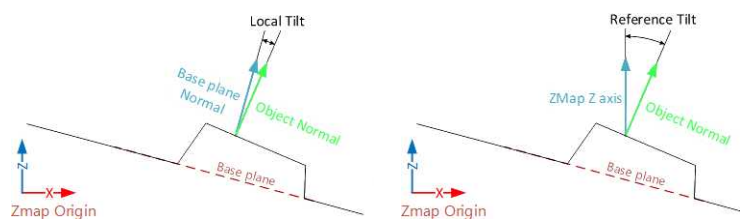
The *local tilt angle* is the angle between the base plane and the object plane.

- 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.

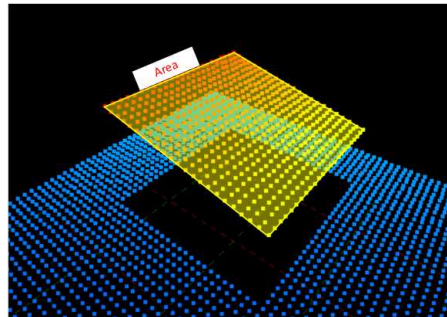


Area

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](#)].

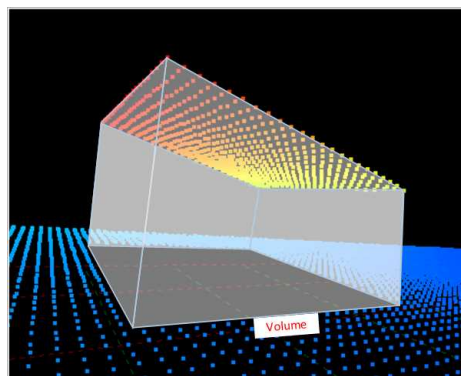
Use the [E3DObjectExtractor.AreaRange](#) accessor to set the range of allowed values for the area.



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.



Sphere

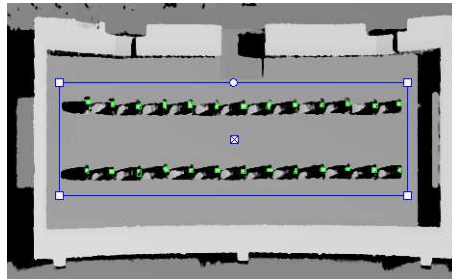
This feature is disabled by default.

When enabled, a sphere is an object [E3DSphere](#) that represents the sphere that best fits the object (calculated according to the least squares).

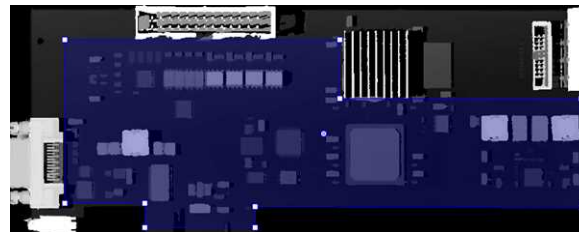
3.3. Extracting and Using Objects

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.
- The computer features for each segmented object.



TIP

To speed up the extraction process:

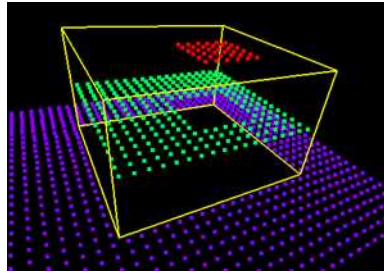
- Adjust the extraction ranges to reduce the number of objects.
- Disable the features you do not need.

Overlapped objects

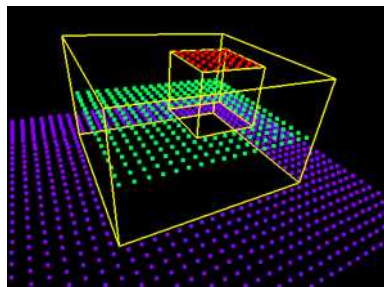
By default, the extraction does not produce objects that overlap on the ZMap. You must enable the `SetOverlappedObject` option to extract “stacked” objects.

The *area ratio* and the *height difference* parameters control how overlapped objects are extracted:

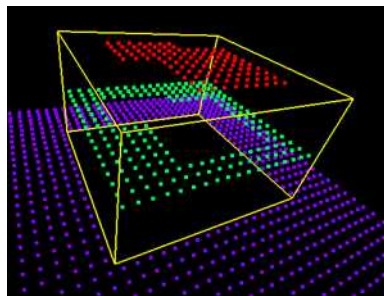
- The *area ratio* is configured by `SetOverlappedAreaRatio`. The area of the bottom object divided by the area of the top object must be larger or equal than that ratio.



Overlapped extraction is disabled.

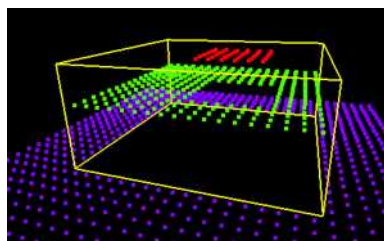


Overlapped extraction is enabled, with `OverlappedAreaRatio = 4`.

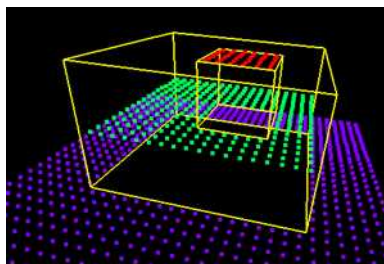


Overlapped extraction is enabled, with `OverlappedAreaRatio = 4`.
The top object is too large to be extracted, the ratio of the areas is lower than 4.

- 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.

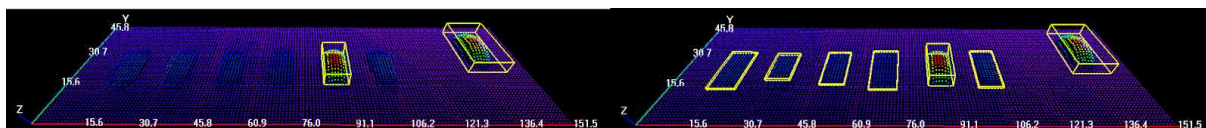


Overlapped extraction is enabled, with `OverlappedHeightDifference = 2`.
 The height of the top object from the bottom object is larger than 2, the object is extracted.

Controlling the object detection

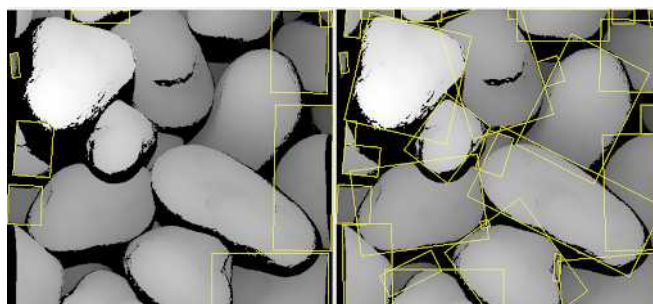
Two optional parameters affect the detection of the object:

- **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.
- Use `GetObjectsMask` to get the mask of all the extracted `E3DObjects`.

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`.

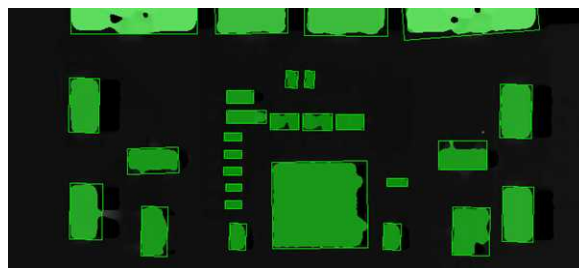


TIP

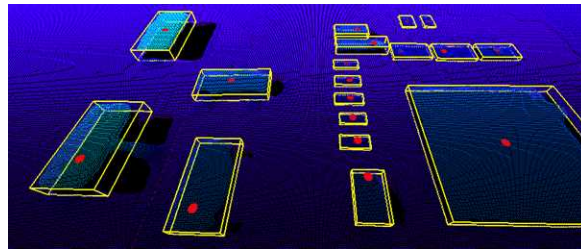
In an `E3DViewer`, use the `ERenderStyle` structure to choose your rendering style.

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

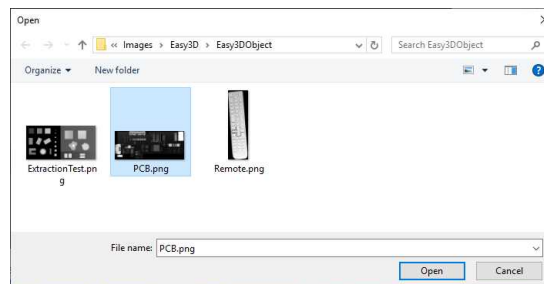
The purpose of this use case is to test if all the components are present and correctly placed on the PCB.



TIP

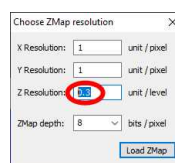
This example uses the sample image `Sample Images\Easy3D\Easy3DObject\PCB.png` and the illustrations are based on the **Easy3DObject** demo application.

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.



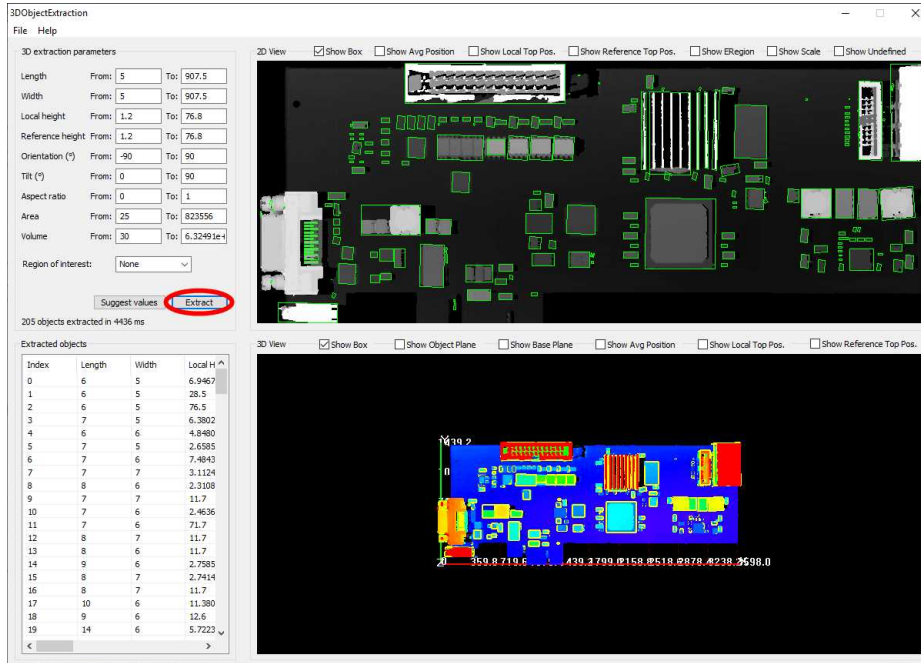
3. Keep the suggested parameters for a first extraction.

- The suggested parameters are set from the ZMap width, height and resolution.

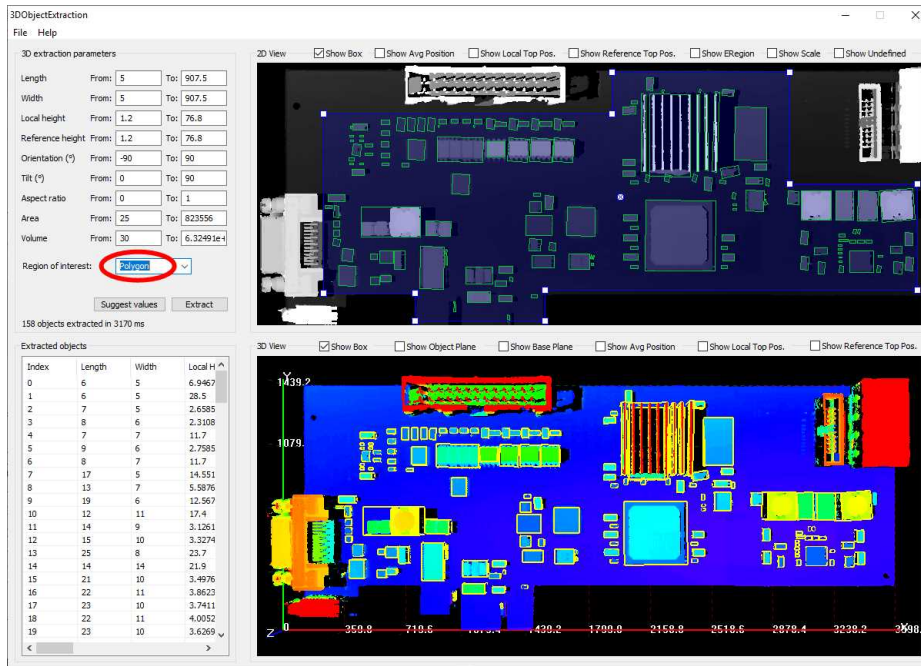
4. Click on the **Extract** button to perform the extraction.

When the extraction is done:

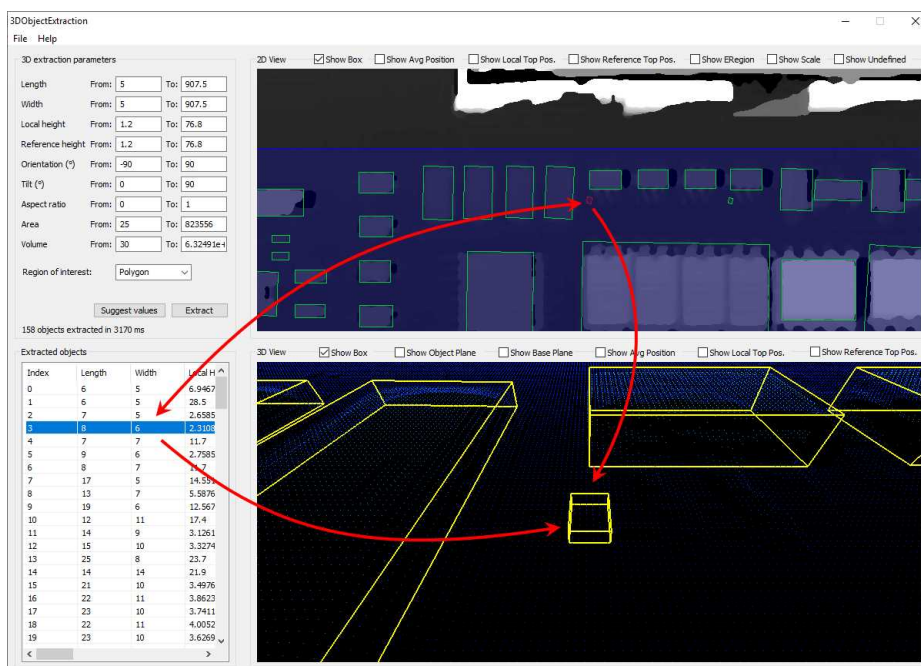
- The object list is filled.
- Click on a column title to sort the object list.
- The various measures are displayed.
- The 2D View and the 3D View show the extracted object bounding boxes.



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.

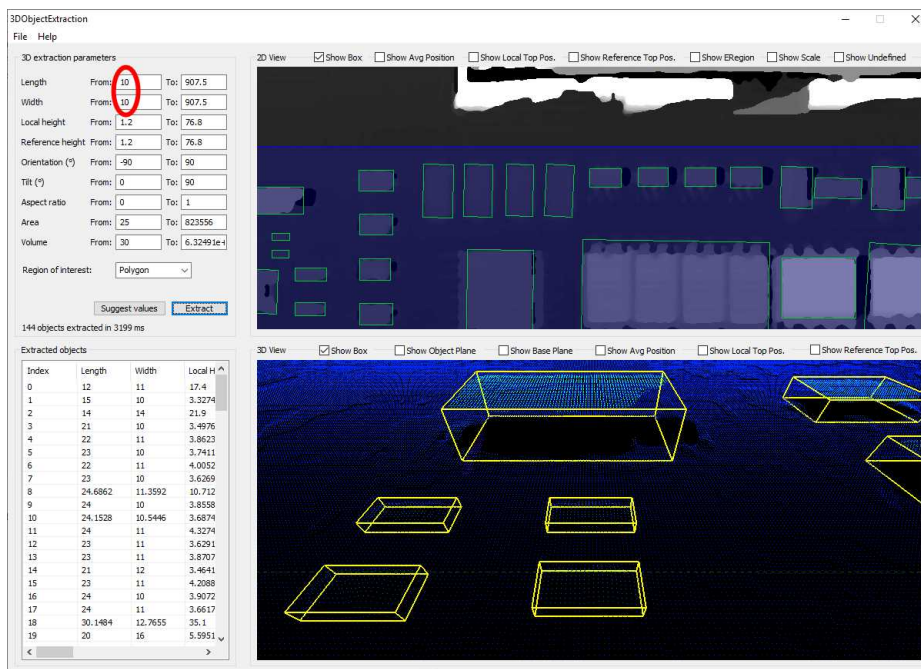
To add or remove objects:

- ❑ Change the extraction parameters, like the length and width ranges.
- ❑ In the illustration below, objects smaller than 10x10 metric unit are not extracted.



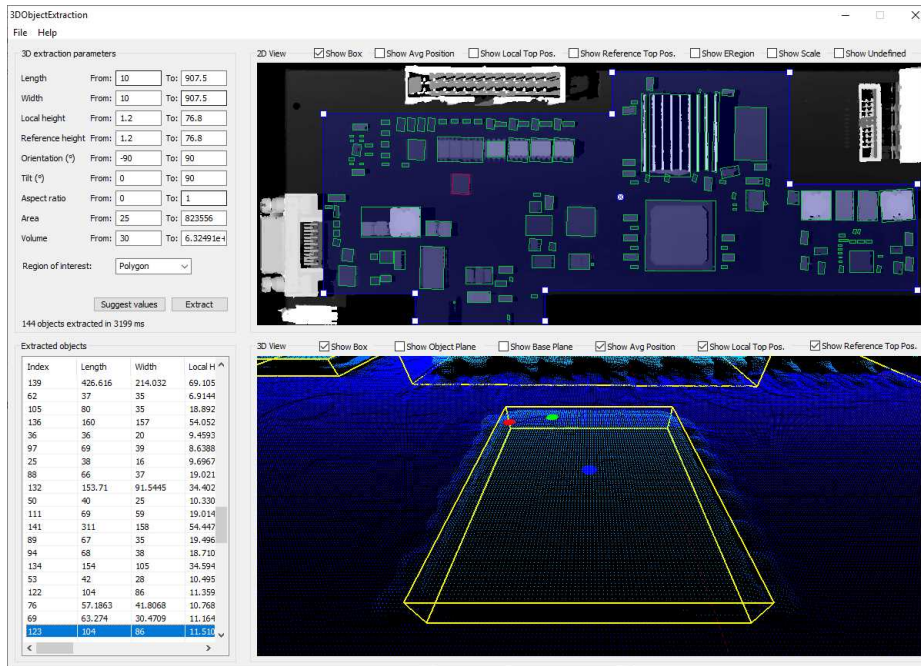
NOTE

After changing a parameter, press the **Extract** button to perform a new extraction.

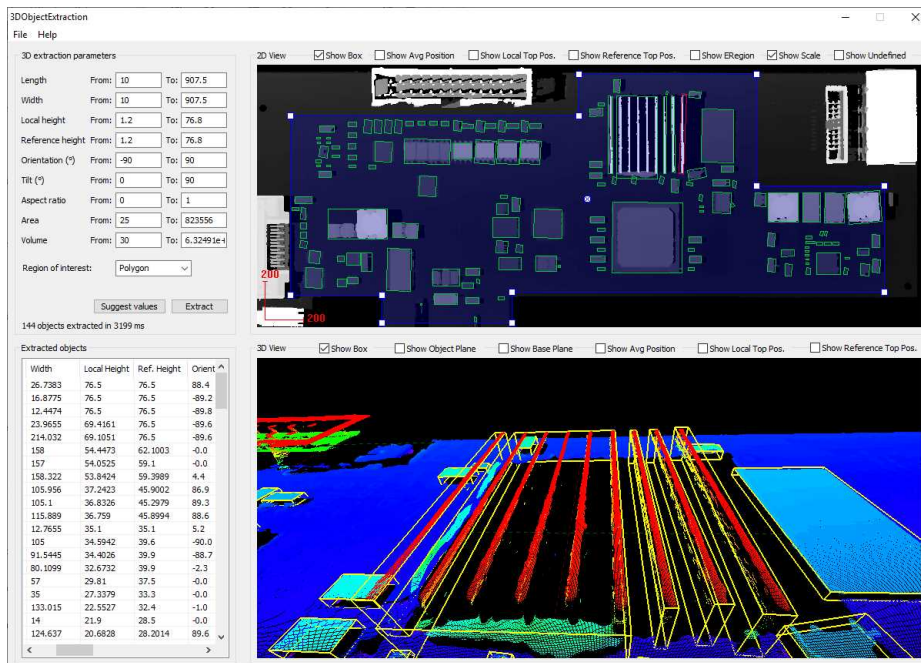


9. Check or uncheck the boxes at the top of the views to toggle the display of most of the object features, either in the 2D View or the 3D View.

- In the illustration below, the object list is sorted by local height.
- The first object is selected and displayed in both views.



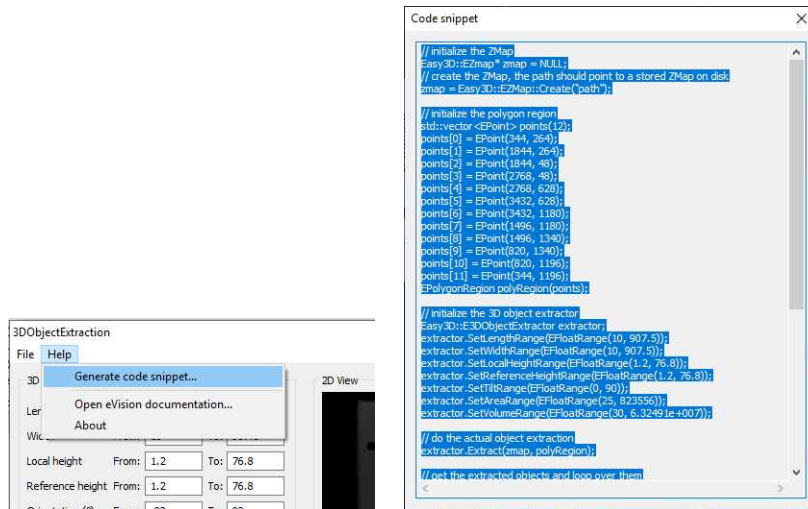
10. Adjust the extraction parameters to accept or reject objects based on the results.



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.



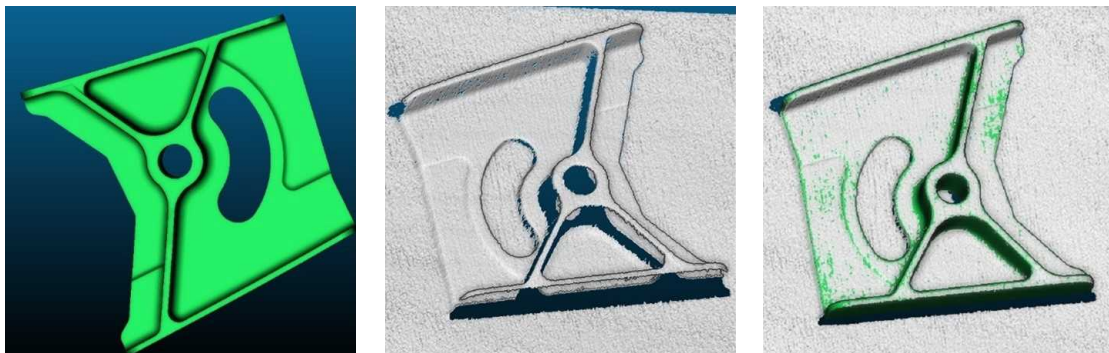
4. Easy3DMatch - 3D Alignment and Comparison

4.1. Purpose and Workflow

Purpose

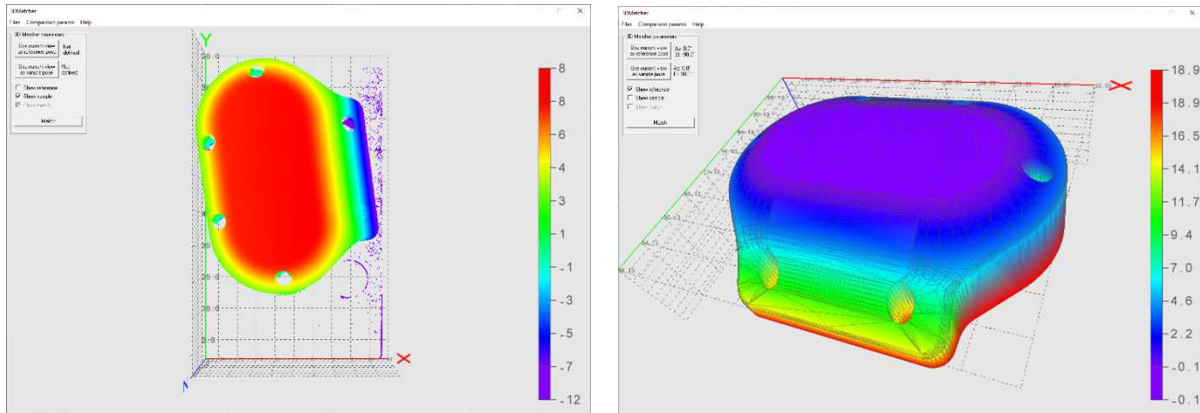
Easy3DMatch allows you to:

- 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 (using the stl file format).

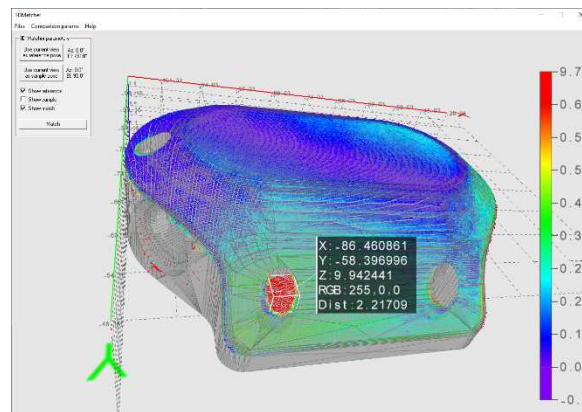


CAD model — sample point cloud — model and sample aligned
(3D models courtesy of Direct Dimensions)

- 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\3D Processing\Easy3DMatchMatch`.

NOTE: 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)

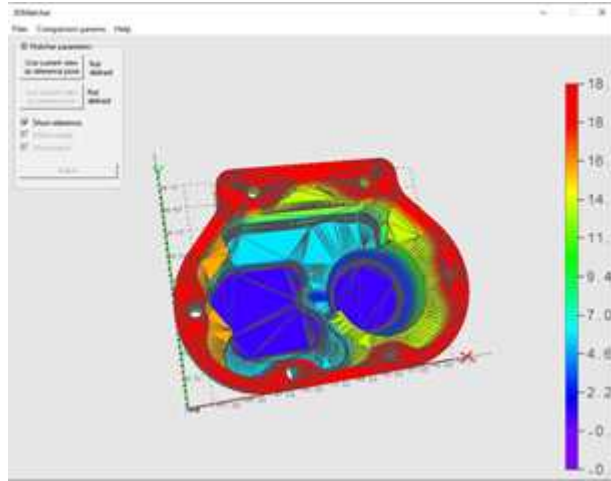
Base case

Use the class [E3DAligner](#) to load a reference and a sample and to find the transformation from the sample to the reference.

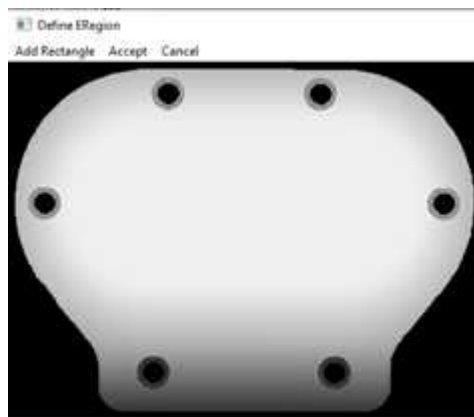
1. The first step is to set the reference using the method `SetReference`.
 - In addition to the mesh or the cloud, this method also takes one or several [E3DPlane](#) or azimuth and elevation angles (see "[Calibration](#)" on [page 433](#) for a definition of the azimuth and the elevation).
 - The angles 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.

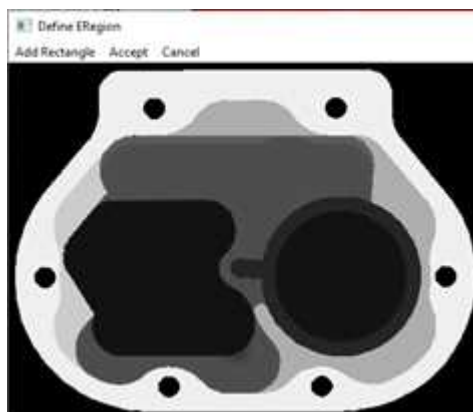
Here is an illustration of the process:



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, 1), 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, -1), -15);
    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	780 ms
2	520 ms
4	383 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 `SetMeshReference` or `SetPointCloudReference` to specify the reference with either an `EMesh` or an `EPointCloud`.
 - Set the options of the comparison (ROIs, mode 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: The `E3DAnomaly` represents points on the reference (except on `NoExtraMaterial` regions and when the distance mode is set to `EComparisonDistanceMode_Euclidian_Advanced` or `EComparisonDistanceMode_Normals_Advanced`).

- 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](#).

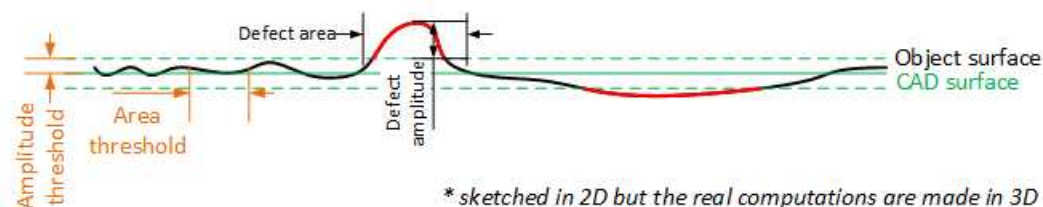


Illustration of the thresholds

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.

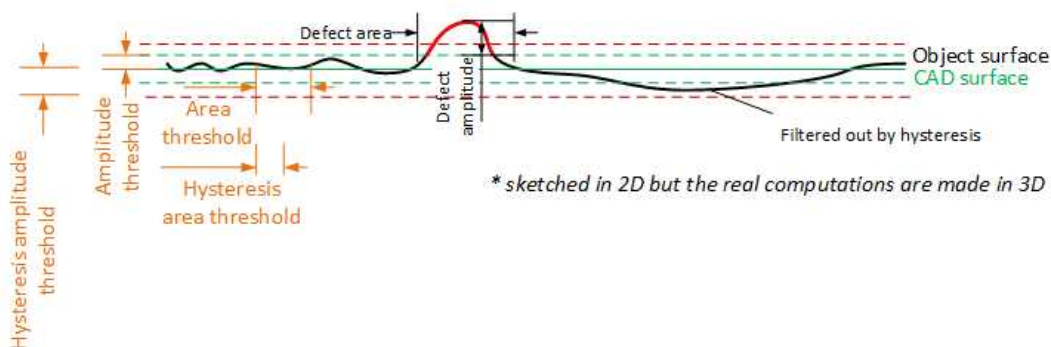
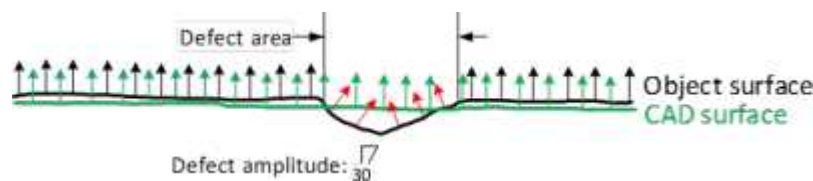


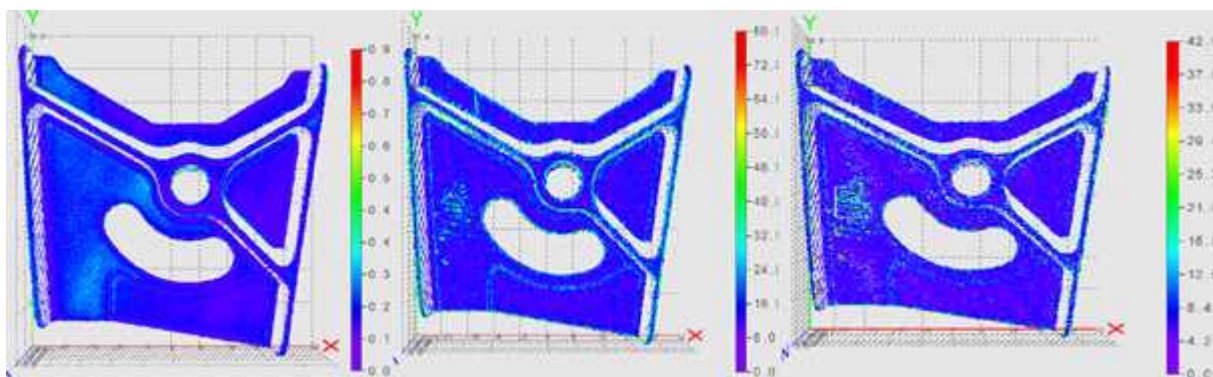
Illustration of the hysteresis thresholds

SetComparisonDistanceMode

- By default, the aligned scan and the reference comparison is based on the euclidean distance between their corresponding points.
 - Use `SetComparisonDistanceMode(EComparisonDistanceMode_Euclidean)` to enable it.
- Another possibility is to base the comparison on the angular distance between the corresponding normals of the scan and of the reference.
 - This works better to detect scratches.
 - Use `SetComparisonDistanceMode(EComparisonDistanceMode_Normals)` to enable it.
- There is a variant that is more robust towards false positives near the edges.
 - Use `SetComparisonDistanceMode(EComparisonDistanceMode_Normals_Advanced)` to enable it.



The Normals comparison mode



Results when using normals to detect scratches.
 (left) scratch is not visible using the euclidean distance mode
 (center) scratch and false positives on the edges are visible using normals distance mode
 (right) only scratch is detected using the advanced normals distance mode

SetEnableAutomaticEdgeCropping and SetEdgeCroppingParameters

- Sharp edges in the reference can lead to false positives, especially when using normals (see `SetComparisonDistanceMode`).

A solution is to remove them automatically:

- Use `SetEnableAutomaticEdgeCropping(true)` to enable this.
- Use `SetEdgeCroppingParameters` to adjust this function.
- By default, the edge cropping is disabled.

SetAutomaticCropFactor

- By default, the sample clouds are automatically cropped around the reference plus a margin to avoid computing distance for points that are not on the object (for example, the plane on which the object lays).
- The margin is obtained by multiplying the anomaly distance threshold by the automatic crop factor (set with [SetAutomaticCropFactor](#)).
 - By default, the factor is 1.

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.SetPointCloudReference(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.SetPointCloudReference(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	503 ms
2	454 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 485) and implements an API close to the one of [E3DComparer](#) (see "[Comparison \(E3DComparer\)](#)" on page 489) with some extra capabilities due to the usage of the reference points of view used in [E3DAligner](#).

To use the [E3DMatcher](#):

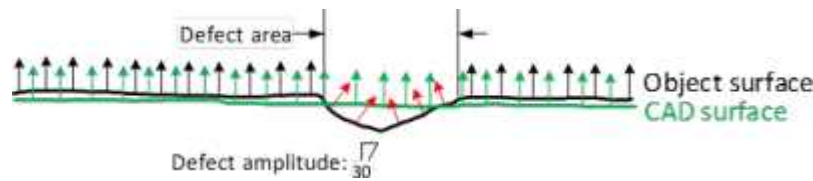
1. The first step is to set the reference using the method `SetReference`.
 - In addition to the mesh or the cloud, this method also takes one or several [E3DPlane](#) or azimuth and elevation angles (see "[Calibration](#)" on page 433 for a definition of the azimuth and the elevation).
 - The angles 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 `Easy3DMatchMatch` 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](#)).

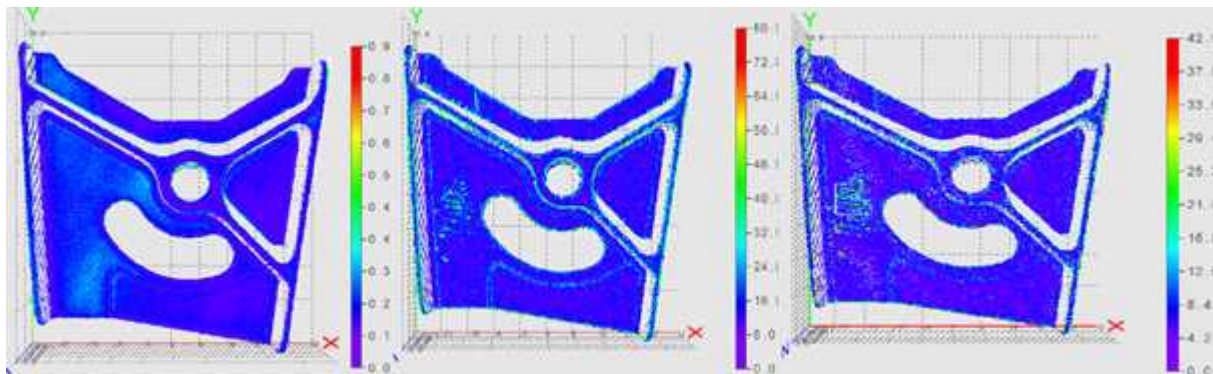
SetComparisonDistanceMode

Use [SetComparisonDistanceMode](#) to select one of the methods available in the class [E3DMatcher](#) to compute the distances:

- [EcomparisonDistanceMode_Euclidean_Fast](#): the fastest method based on the euclidean distance between points. It is less precise on the edges.
- [EcomparisonDistanceMode_Euclidean](#): the default method. It is based on the euclidean distance between corresponding points.
- [EcomparisonDistanceMode_Euclidean_Advanced](#): the slowest method based on the euclidean distances. It penalizes more the bumps.
- [EcomparisonDistanceMode_Normals](#): the same as [EcomparisonDistanceMode_Euclidean](#) except that the comparison is based on the angular distance between the points normals instead of the euclidean distance between the points. It works better to detect scratches.
- [EcomparisonDistanceMode_Normals_Advanced](#): a variant of [EcomparisonDistanceMode_Normals](#) that is more robust towards false positives near the edges.



The Normals comparison mode



Results when using normals to detect scratches.
 (left) scratch is not visible using the euclidean distance mode
 (center) scratch and false positives on the edges are visible using normals distance mode
 (right) only scratch is detected using the advanced normals distance mode

SetEnableAutomaticEdgeCropping and SetEdgeCroppingParameters

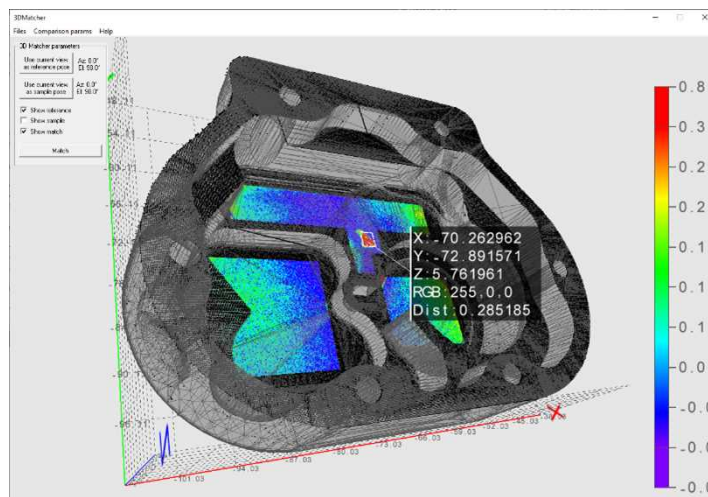
- Sharp edges in the reference can lead to false positives, especially when using normals (see [SetComparisonDistanceMode](#)).

A solution is to remove them automatically:

- Use [SetEnableAutomaticEdgeCropping\(true\)](#) to enable this.
- Use [SetEdgeCroppingParameters](#) to adjust this function.
- By default, the edge cropping is disabled.

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 or 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

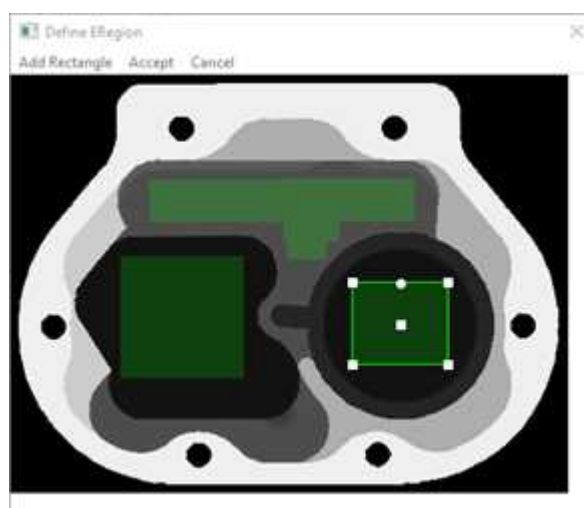
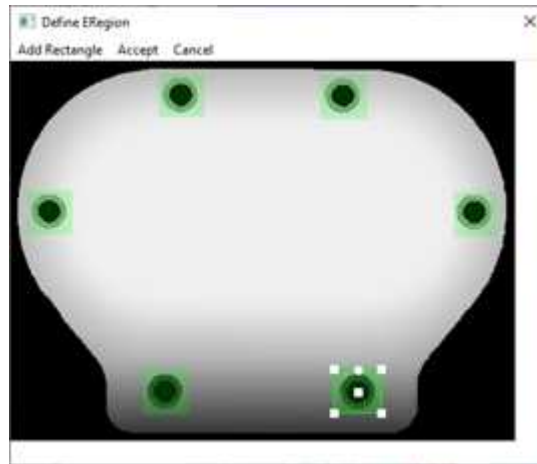


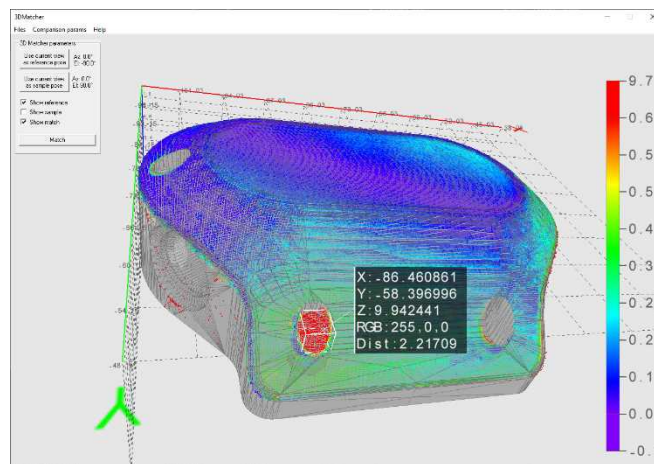
Illustration of setting ROI on a projection of the reference

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.



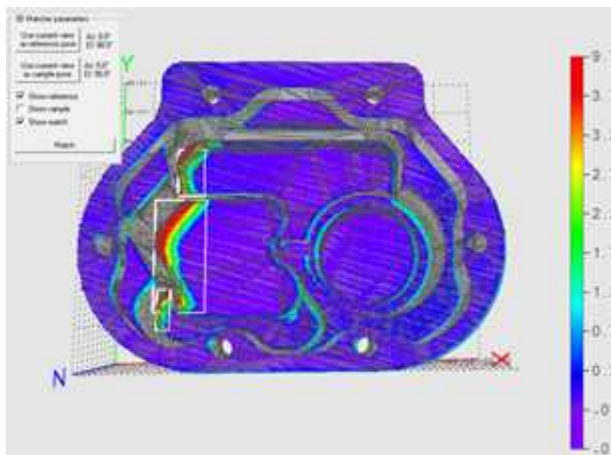
We do not want the holes to be filled



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](#).

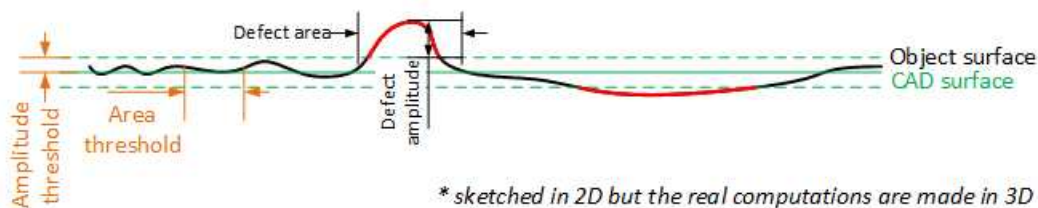


Illustration of the thresholds

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.

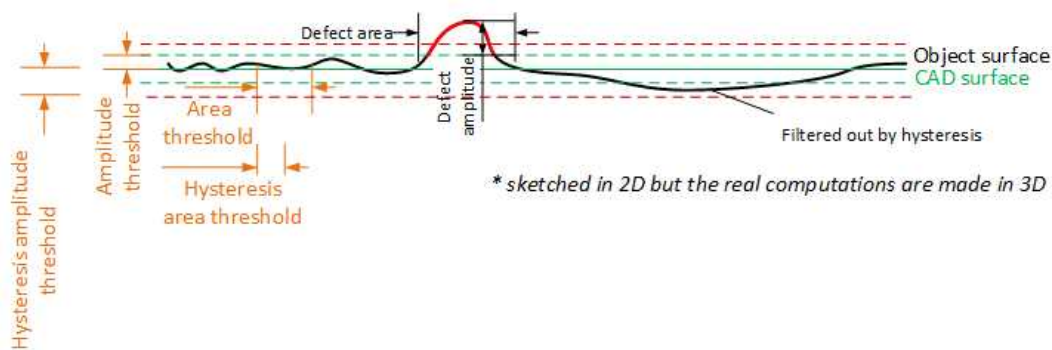


Illustration of the hysteresis thresholds

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.

SetAutomaticCropFactor

- By default, the sample clouds are automatically cropped around the reference plus a margin to avoid computing distance for points that are not on the object (for example, the plane on which the object lays).
- The margin is obtained by multiplying the anomaly distance threshold by the automatic crop factor (set with [SetAutomaticCropFactor](#)).
 - By default, the factor is 1.

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	683 ms
2	642 ms
4	587 ms

4.5. 3D Sensor Fusion (EPointCloudMerger)

NOTE: You need a license for **Easy3DMatch** to use the sensor merging tools.

3D sensor fusion

The 3D sensor fusion is a technique to merge the output of different 3D sensors together. In this case, these are different views of the same object.

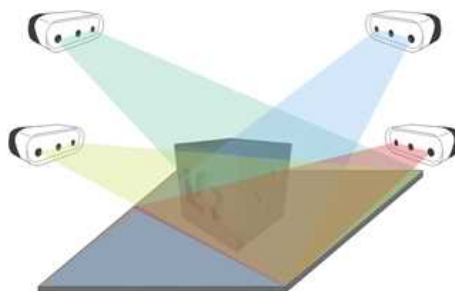
Use the class `EPointCloudMerger`:

1. Choose how many sensors to use and position them.
2. Acquire scans of the calibration cube (that must be 3D printed).
3. Use the method `Calibrate` to perform the calibration.
4. For each new object:
 - a. Acquire the scans of the object.
 - b. Use the method `Merge` to merge the scans together.

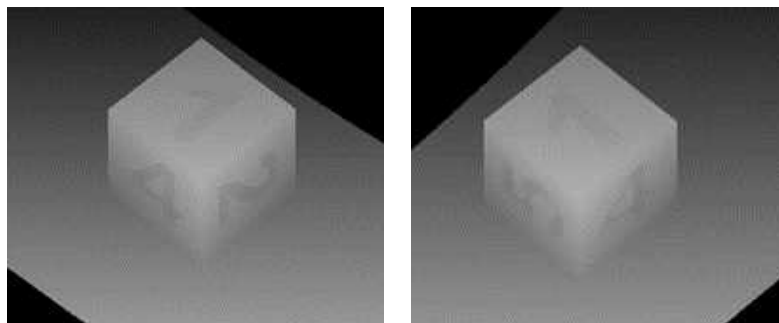


TIP

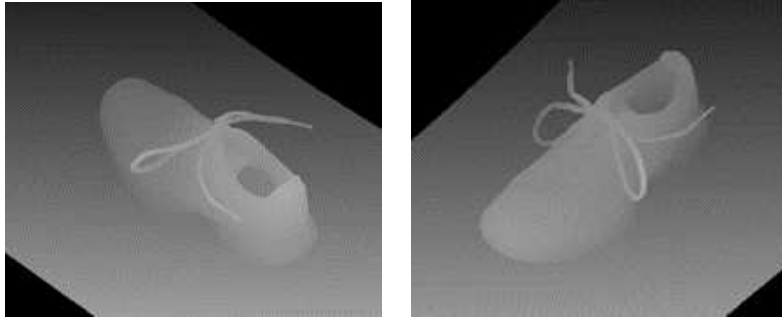
You can perform all these steps interactively in the C++ and C# `Easy3DMatchPointCloudMerger` samples.



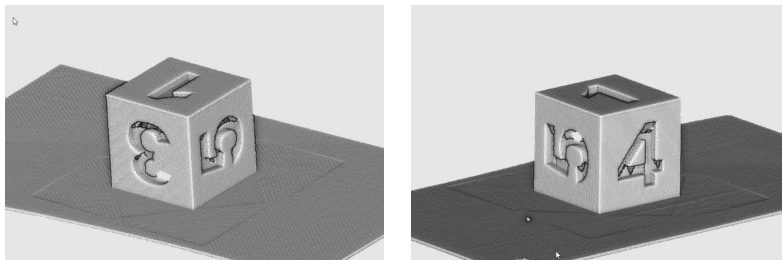
Several sensors acquiring different views of the calibration object



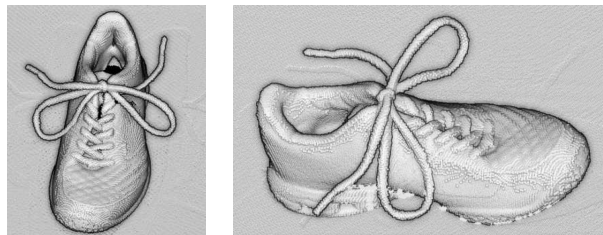
Scans of the calibration cube (displayed as ZMaps)



Scans of the object (displayed as ZMaps)



Merged calibration scans



Merged cloud scans

3D CAD model

- The calibration cube model is available in the STL file format.
- Download this file from the **Open eVision** download area in the **Additional Resources** section (www.euresys.com/Support).

OPEN EVISION			2.17	
	Download	File size	Operating system	
Release Notes	open_evision_release_notes-2.17.2.1161.pdf [Internal file versions]	1.1 MB	Windows	
Documentation	View Open eVision 2.17 online documentation (including PDFs)		Windows	
	open_evision-win-Offline-Documentation-en-2.17.1.1160.exe	0.1 GB	Windows	
	open_evision-linux-Offline-Documentation-en-2.17.1.1160.tar.gz	0.2 GB	Linux	
	open_evision-win-Offline-Documentation-en-cn-2.17.1.1160.exe	0.2 GB	Windows	
	open_evision-linux-Offline-Documentation-en-cn-2.17.1.1160.tar.gz	0.3 GB	Linux	
	open_evision-win-Offline-Documentation-en-isp-2.17.1.1160.exe	0.2 GB	Windows	
	open_evision-linux-Offline-Documentation-en-isp-2.17.1.1160.tar.gz	0.3 GB	Linux	
	open_evision-win-Offline-Documentation-en-ko-2.17.1.1160.exe	0.2 GB	Windows	
open_evision-linux-Offline-Documentation-en-ko-2.17.1.1160.tar.gz	0.3 GB	Linux		
Setup Files	open_evision-win-2.17.2.13747.exe	0.6 GB	Windows	
	open_evision-linux-06_64-2.17.2.13743.deb.tar.gz	0.5 GB	Linux	
	open_evision-linux-06_64-2.17.2.13743.rpm.tar.gz	0.5 GB	Linux	
	open_evision-win-studio-2.17.2.13747.msi	0.2 GB	Windows	
	open_evision-win-deep-learning-studio-2.17.2.13747.msi	0.2 GB	Windows	
	open_evision-win-3d-studio-2.17.2.13747.msi	0.1 GB	Windows	
	open_evision-win-license-manager-2.17.2.13747.msi	43 MB	Windows	
	neo-wiki-license-manager-2.17.2.13747.exe	45 MB	Windows	
	neo-linux-license-manager-06_64-2.17.2.13749.deb.tar.gz	34 MB	Linux	
	neo-linux-license-manager-06_64-2.17.2.13749.rpm.tar.gz	41 MB	Linux	
Additional Resources	Deep Learning Additional Resources 2.17.2.13747.zip	1.3 GB	Windows	
	Easy 3D Calibration Models 2.17.0.13019.zip	0.4 MB	Windows	
	Easy 3D Sensors Compatibility 2.17.2.13747.zip	14 MB	Windows	
	[Internal file versions]			

Download the calibration object model

- Once downloaded, we recommend using a specialized subcontractor to print the 3D calibration cube as 3D filament printing is not really suitable to produce the cube.

For reference, we worked with a company using Selective Laser Sintering and printing a 10×10×10 cm cube using PA12 material costed 220 €.



The calibration cube

Removing duplicate points

A physical point seen by two sensors should not be present two times in the output cloud.

- By default, these points are removed. As this takes most of the processing time, you can disable the process if speed is an issue.
- Set the cloud resolution with the parameter [SetMergedCloudResolution](#) to control this process. The parameter value is computed automatically but you can increase it to reduce the size of the output cloud and speed-up the processing.

Code samples

```

////////////////////////////////////
// This code snippet shows how to perform sensor fusion. //
////////////////////////////////////

// Calibration
Easy3D::EPointCloudMerger merger;
std::vector<EPointCloud> calibrationClouds; // TODO: load or grab
float calibrationObjectSize = 100.f; // size of an edge of the cube in the calibrationClouds
float calibrationScore = merger.Calibrate(calibrationClouds, calibrationObjectSize, true);

// Merging
std::vector<EPointCloud> clouds; // TODO: load or grab, must be in same order as CalibrationClouds
EPointCloud mergedCloud;

merger.Merge(clouds, mergedCloud);
    
```

Computation Time

The following table shows the computation time on 4 clouds of around 200,000 points each.

Nb of threads	Calibration (ms)	Merging @ res (ms)	Merging @ 4 x res (ms)	Merging w/o decimation
1	3386	73	44	15
2	2298	64	46	9
4	1752	52	47	9

Where Res is the resolution computed by the [Calibrate](#) method when its argument `computeMergedCloudResolution` is set to true.

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.

- To enable parallel processing in **Open eVision Studio**:
 - Go to `View > Options`.
 - In the pop-up, tune the number of processing threads enabled.

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	Max. expected improvement for 2 proc. threads
EasyMatrixCode2	<code>EMatrixCodeReader::Read</code>	50%
EasyBarcode2	<code>EBarcodeReader::Read</code>	20%
EasyFind	<code>EPatternFinder::Find</code>	30%
EasyImage	Threshold on ERegion	50%
	Statistics on ERegion	75%
	<code>EasyImage::Median</code>	30%

Library	Method	Max. expected improvement for 2 proc. threads
Easy3D	<code>EPointCloudToZMapConverter::Convert</code>	50%
	<code>EDepthMapToPointCloudConverter::Convert</code>	30%
	<code>EDepthMapToMeshConverter::Convert</code>	30%
	<code>EZMapToPointCloudConverter::Convert</code>	30%
	<code>EPhotometricStereoImager::Compute</code>	35%
	<code>ELaserLineExtractor::ExtractProfileFromFrame</code>	30%
Easy3DMatch	<code>E3DAligner::Align</code>	30%
	<code>E3DComparer::Compare</code>	10%
	<code>E3DMatcher::Match</code>	10%
	<code>EPointCloudMerger::Calibrate</code>	30%
	<code>EPointCloudMerger::Merge</code>	10%
EasyClassify	<code>EClassifier::Classify</code>	2%
EasySegment	<code>EUnsupervisedSegmenter::Apply</code>	4%
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%

**NOTE**

- The speedups are given for 1 additional processing thread (so a total of 2 processing threads). Adding more processing threads could lead to some improvements, but usually the speedups are not linear with the number of processing threads.
- The improvements strongly depend on the parameters of the methods and the size and type of the images.
- The speedups can also vary according to the CPU type.

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 EColor , EPeak , EISH , ELAB , ELCH , ELSH , ELUV , EBW1 , EBW8 , EBW8Path , EBW16 , EBW16Path , EBW32 , EC15 , EC16 , EC24 , EC24A , EC24Path , EPath , ERGB , ERGBColor , EVSH , EXYZ , EYIQ , EYSH , EYUV , EDepth8 , EDepth16 and EDepth32f		No
Pixel collection classes EColorLookup , EPseudoColorLookup , EPeakVector , EBW8Vector , EBWHistogramVector , EBW8PathVector , EBW16PathVector , EBW16Vector , EBW32Vector , EC24Vector , EPathVector , EColorVector , EColorLookup and EC24PathVector	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 EImageBW1 , EImageBW8 , EImageBW16 , EImageBW32 , EImageC15 , EImageC16 , EImageC24 and EImageC24A	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 EDepthMap8 , EDepthMap16 and EDepthMap32f EZMap8 , EZMap16 and EZMap32f	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.
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.
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 EOCR 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 eGrabber Sources

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 **eGrabber** sources, to an **Open eVision** data container.

See [EGrabber documentation](#) for more information about the **eGrabber** library.

EGrabber sources

The possible **eGrabber** sources are the following:

- **Euresys Coaxlink** frame grabbers
- **Gigalink** vision cameras

Prerequisites

- **Open eVision** 2.9 (or newer)
- With a **Coaxlink** frame grabber: **Coaxlink** 11.0.3 (or newer) or **eGrabber** 12.8 (or newer)
- With a **Gigalink** vision camera: **eGrabber** 12.8 (or newer)

Libraries

- 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** assembly 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 [EGrabberBridge](#) code snippets).
- Each class **EGrabberBridge** derives from a specific **Open eVision** data container.

The following classes are implemented:

Base class	EGrabberBridge class	Corresponding GenAPI pixel format
DepthMap16	EGrabberBridge::EGrabberDepthMap16	Coord3D_C16
DepthMap8	EGrabberBridge::EGrabberDepthMap8	Coord3D_C8
EImageBW16	EGrabberBridge::EGrabberImageBW16	Mono16 BayerGR16, BayerRG16 BayerGB16, BayerBG16
EImageBW8	EGrabberBridge::EGrabberImageBW8	Mono8 BayerGR8, BayerRG8 BayerGB8, BayerBG8
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 **GenTL** parameters can be handled through the **EGrabber** Object setters and getters (see [EGrabber documentation](#)).
- You can also test out parameters using the **GenTL** application (see [GenTL documentation](#)).

3. VimbaXBridge - Using Images from VimbaX Sources

See also: code snippets: [VimbaXBridge](#)

VimbaXBridge and VimbaX

VimbaXBridge is a user-friendly namespace of conversion classes. These classes perform the direct conversion from a buffer originating from **VimbaX** to an **Open eVision** data container.

Prerequisites

- **Open eVision** 23.12 (or newer)
- **VimbaX** 1.0 (or newer)

Libraries

- To use the VimbaXBridge in C++, include the main **VimbaX** headers before the **Open eVision** header.

```
#include <VmbC/VmbC.h> or #include <VmbCPP/VmbCPP.h>
#include <VmbImageTransform/VmbTransform.h>
#include "Open_eVision.h" or #include "Easy.h"
```

Data containers

- VimbaXBridge is part of the main **Open eVision** header.
- The VmbImageTransform header is required only if you need to perform a pixel format conversion.
- Each class VimbaXBridge derives from a specific **Open eVision** data container.

The following classes are implemented:

Base class	VimbaXBridge class	Corresponding GenAPI pixel format
EImageBW16	VimbaXBridge::EVimbaXImageBW16	Mono16 BayerGR16, BayerRG16 BayerGB16, BayerBG16
EImageBW8	VimbaXBridge::EVimbaXImageBW8	Mono8 BayerGR8, BayerRG8 BayerGB8, BayerBG8
EImageC24	VimbaXBridge::EVimbaXImageC24	BGR8

- These classes have 2 constructors:
 - 1 that requires only an **VimbaX** C frame descriptor and an optional color transformation directive.
 - 1 that requires only an **VimbaX** C++ frame descriptor and an optional color transformation directive.
- Scope and copy of the buffer:

Open eVision does not perform any copy of the buffer unless:

 - You require a pixel format conversion.
 - The color transformation directive is set to Always, in which case a copy is always performed.

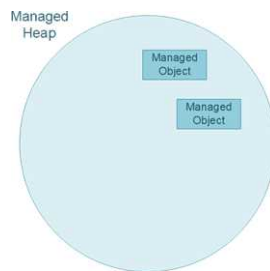
Examples and code snippets

- A sample illustrates the use of the classes `VimbaXBridge` using callbacks. The sample is available in C++ and is present in its corresponding samples solution under the `VimbaXBridge` name.
- A code snippet ([Using VimbaXBridge](#)) is available to show how to use `VimbaXBridge` to perform the inversion of an image acquired using **VimbaX**.

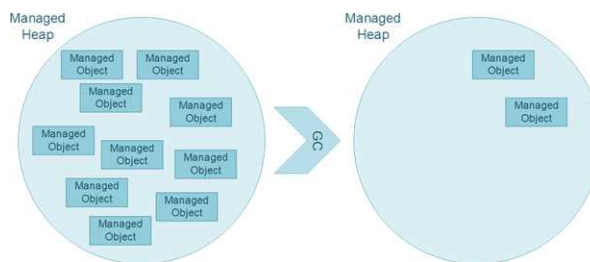
4. Handling the Memory in .NET

The memory management in .NET

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

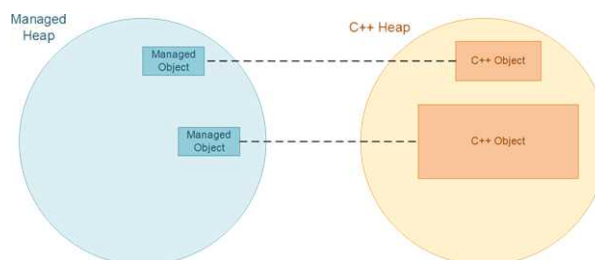


- If that managed heap becomes too full, a specialized process called the *garbage collector* (GC) starts. This process cleans and removes from the memory the objects that are no longer in use.

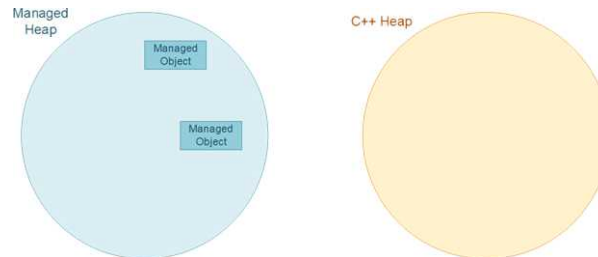


Open eVision specificity

- 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 **EasyObject**:

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

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

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

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

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

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

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

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



NOTE

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

Good practice: use the using statement

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

- Instead of writing your code as follows:

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

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

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

Function scope and limitation

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

In the above code, when the variable is released at the end of the function scope, you can 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(128, 128);
    ...
}

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

- In the above example code:
 - `imageInFunction` and `imageOutOfFunction` are in fact the same image.
 - When you call `.Dispose()` on one image, both are disposed 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(128, 128);
    ...
}

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

Good practice: set the variables to null

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

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

5. 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`.