Open eVision

General Purpose Libraries
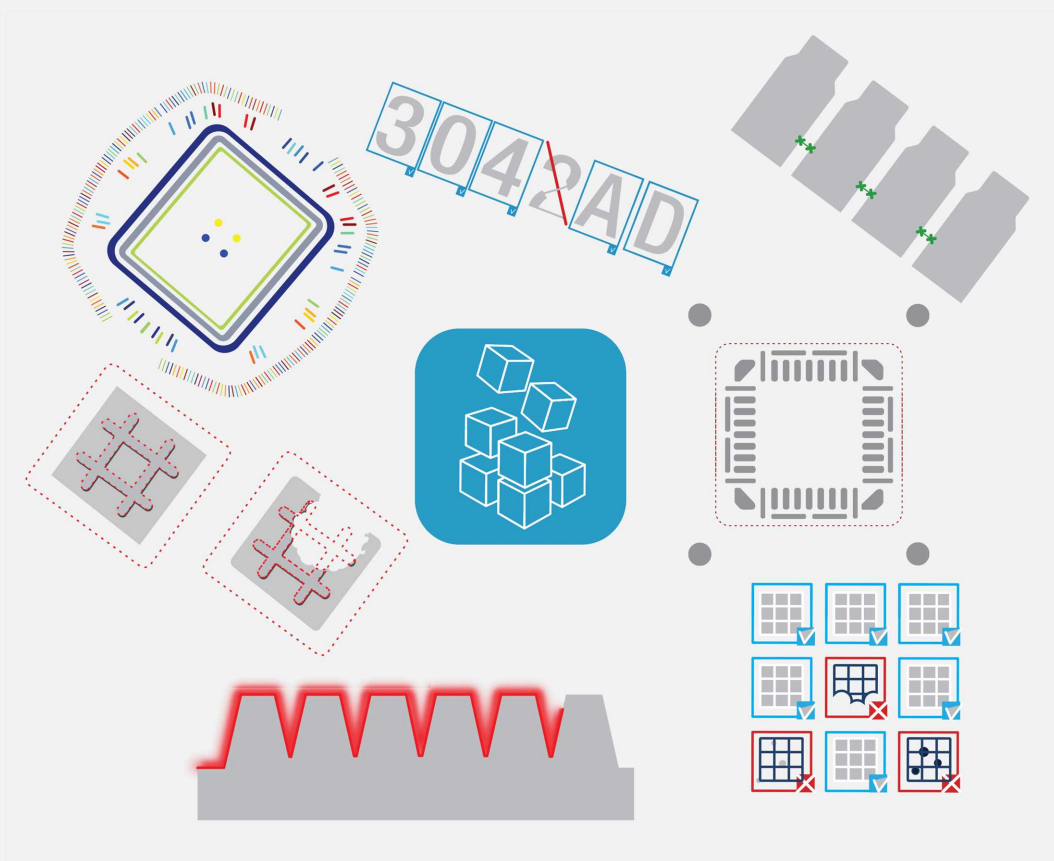
Open eVision   **User Guide**

Contents

1. Dealing with Pixel Containers and Files

1.1. Pixel Container Definition

Images

Open eVision image objects contain image data that represents rectangular images.

Each image object has a data buffer, accessible via a pointer, where pixel values are stored contiguously, row by row.
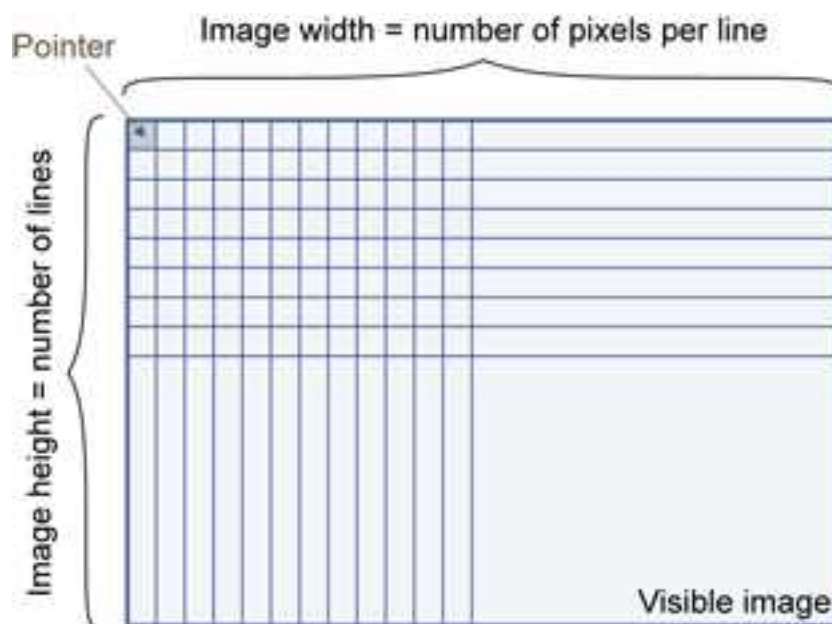
Image main parameters

An Open eVision image object has a rectangular array of pixels characterized by `EBaseROI` parameters .

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

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

Classes

Image and ROI classes derive from abstract class `EBaseROI` and inherit all its properties.
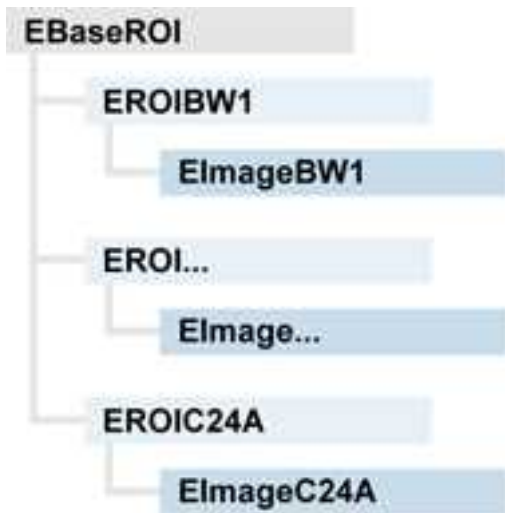
Depth maps

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

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

Point clouds

A point cloud (https://en.wikipedia.org/wiki/Point_cloud) is an unstructured set of 3D points representing discrete positions on the surface of an object.
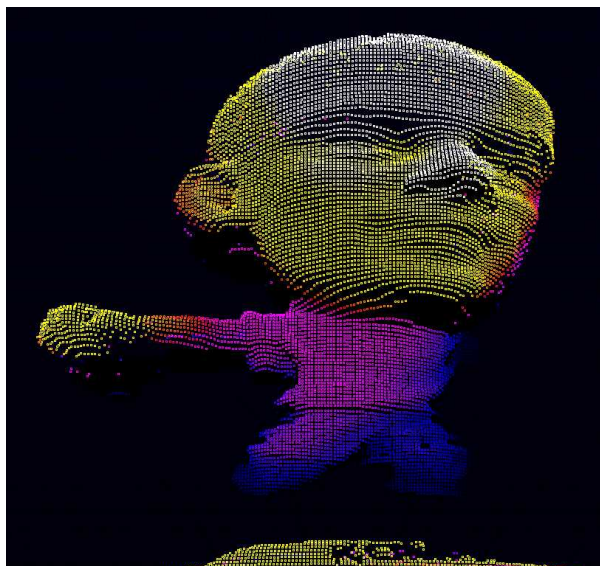
3D point clouds are produced by various 3D scanning techniques, such as Laser Triangulation, Time of Flight or Structured Lighting.

1.2. Pixel Container Types

Reference

Images

Several image types are supported according to their pixel types: black and white, gray levels, color, etc.

Easy.GetBestMatchingImageType returns the best matching image type for a given file on disk.

BW1	1-bit black and white images (8 pixels are stored in 1 byte)	EImageBW▸
BW8	8-bit grayscale images (each pixel is stored in 1 byte)	EImageBW8
BW16	16-bit grayscale images (each pixel is stored in 2 bytes)	EImageBW▸6
BW32	32-bit grayscale images (each pixel is stored in 4 bytes)	EImageBW32
C15	15-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images and MultiCam RGB15 format.	EImageC▸5

7

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

Depth Maps

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

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

Point Clouds

Point Cloud	Set of points coordinates (stored as float)	EPointCloud

1.3. Supported Image File Types

Reference

Type	Description
BMP	Uncompressed image data format (Windows Bitmap Format)
JPEG	Lossy data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 10918-1. Compression irretrievably loses quality.
JFIF	JPEG File Interchange Format
JPEG-2000	Data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 15444-1 and ISO/IEC 15444-2. Open eVision supports only lossy compression format, file format and code stream variants.

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

1.4. Pixel and File Types Compatibility

Depth map to image conversion

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

Pixel and file types compatibility

Pixel access

The recommended method to access pixels is to use `SetImagePtr` and `GetImagePtr` to embed the image buffer access in your own code. See also Image Construction and Memory Allocation and Retrieving Pixel Values.

Use of the following methods should be limited because of the overhead incurred by each function call:

Direct access

`EROIBW8.GetPixel` and `SetPixel` methods are implemented in all images and ROI classes to read and write a pixel value at given coordinates. To scan all pixels of an image, you could run a double loop on the X and Y coordinates and use `GetPixel` or `SetPixel` each iteration, but this is not recommended.

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

Quick Access to BW8 Pixels

In BW8 images, a call to `EBW8PixelAccessor.GetPixel` or `SetPixel` will be faster than a direct `EROIBW8.GetPixel` or `SetPixel`.

Supported structures

- `EBW▸`, `EBW8`, `EBW32`
- `EC▸5` (*), `EC▸6` (*), `EC24` (*)
- `EC24A`
- `EDepth8`, `EDepth▸6`, `EDepth32f`,

(*) These formats support RGB15 (5-5-5 bit packing), RGB16 (5-6-5 bit packing) and RGB32 (RGB + alpha channel) but they must be converted to/from EC24 using `EasyImage.Convert` before any processing.

> 📝 **NOTE**
> Transition with versions prior to eVision 6.5 should be seamless: image pixel types were defined using typedef of integral types, pixel values were treated as unsigned numbers and implicit conversion to/from previous types is provided.

Pixel and File Type compatibility during Load or Save operations

Type	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
BW1	Ok	N/A	N/A	Ok	Ok	Ok
BW8	Ok	Ok	Ok	Ok	Ok	Ok
BW16	N/A	N/A	Ok	Ok	Ok (***)	Ok
BW32	N/A	N/A	N/A	N/A	Ok (***)	Ok
C15	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C16	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C24	Ok	Ok	Ok	Ok	Ok (**)	Ok
C24A	Ok	N/A	N/A	Ok	N/A	Ok
Depth8	Ok	Ok	Ok	Ok	Ok	Ok

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

N/A: Not supported. An exception occurs if you use the combination.

Ok: Image integrity is preserved with no data loss (apart from JPEG and JPEG2000, lossy compression).

(**) C15 and C16 formats are automatically converted into C24 during the save operation.

(***) BW16 and BW32 are not supported by Baseline TIFF readers.

1.5. Color Types

EISH: Intensity, Saturation, Hue color system.

ELAB: CIE Lightness, a*, b* color system.

ELCH: Lightness, Chroma, Hue color system.

ELSH: Lightness, Saturation, Hue color system.

ELUV: CIE Lightness, u*, v* color system.

ERGB: NTSC/PAL/SMPTE Red, Green, Blue color system.

EVSH: Value, Saturation, Hue color system.

EXYZ: CIE XYZ color system.

EYIQ: CCIR Luma, Inphase, Quadrature color system.

EYSH: CCIR Luma, Saturation, Hue color system.

EYUV: CCIR Luma, U Chroma, V Chroma color system.

2. Manipulating Pixels Containers and Files

2.1. Pixel Container File Save

Images and depth maps

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

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

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

`Save` throws an exception when:

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

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

Image file type arguments

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

(*) Default value.

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

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

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

SaveJpeg and SaveJpeg2K specify the compression quality when saving compressed images. They have two arguments:
- □ Path: a string of characters including the path, filename, and file name extension.
- □ Compression quality of the image file, an integer value in range [0: 100].
  SaveJpeg saves image data using JPEG File Interchange Format – JFIF.
  SaveJpeg2K saves image data using JPEG 2000 File format.

JPEG compression values

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

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

Representative JPEG 2000 compression quality values

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

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

Saving point clouds

Use the following methods to save a point cloud in a specific format:
- □ EPointCloud::Save: Open eVision proprietary file format.
- □ EPointCloud::SaveCSV: CSV file.
- □ EPointCloud::SaveOBJ: OBJ file.
- □ EPointCloud::SavePCD: PCD file.
- □ EPointCloud::SavePLY: PLY file.
- □ EPointCloud::SaveXYZ: XYZ file.

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

2.2. Pixel Container File Load

Loading images and depth maps

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

- The Load method throws an exception when:
  - □ File type identification fails
  - □ File type is incompatible with pixel type of the image object

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

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

Loading point clouds

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

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

> ✔ **TIP**
> - The PCD format is supported in ASCII and binary modes.
> - The PLY is supported only in ASCII mode.

2.3. Memory Allocation

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

Internal memory allocation

The image object dynamically allocates and deallocates a buffer.
- □ Memory management is transparent.
- □ When the image size changes, reallocation occurs.
- □ When an image object is destroyed, the buffer is deallocated.

To declare an image with internal memory allocation:

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

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

External memory allocation

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

To declare an image with external memory allocation:

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

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

c. Assign the buffer to the image with `SetImagePtr`.

> 📄 **NOTE**
>
> If your buffer rows are not aligned on 4 bytes, you cannot use `SetImagePtr`. In that case, use `InitializeFromUnalignedBuffer` instead.
>
> Please note, however, that this allocates the memory internally and copies the external buffer into the internal one instead of using the external one directly.

2.4. Image and Depth Map Buffer

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

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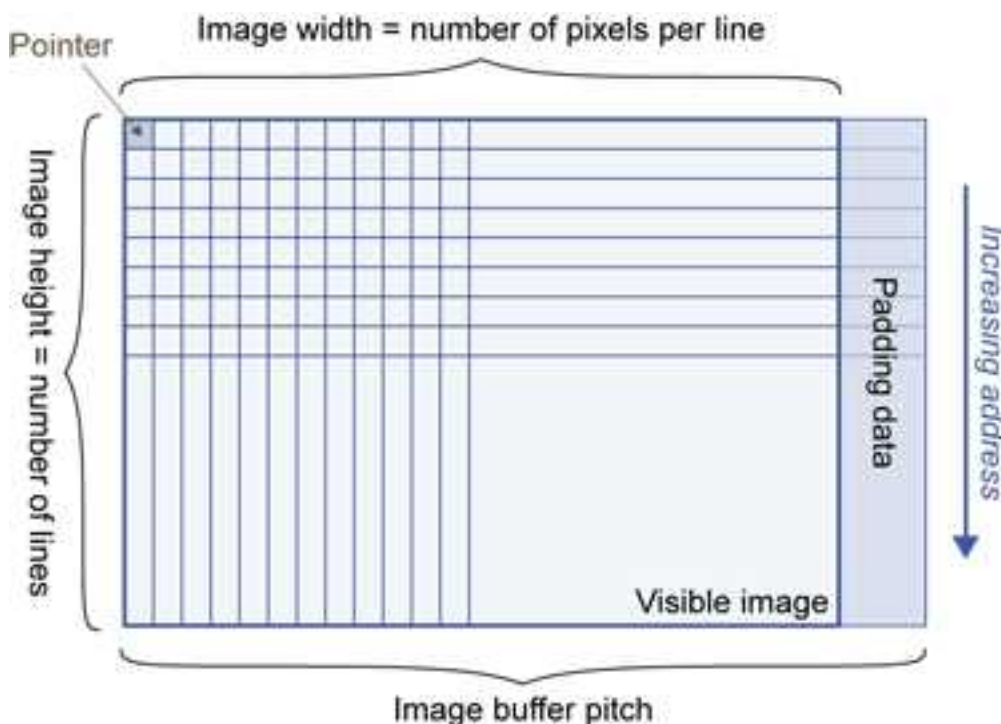
Image buffer pitch

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

Memory layout

- `EImageBW` stores 8 pixels in one byte.

  Example memory layout of the first 2 pixels of a BW1 image buffer:
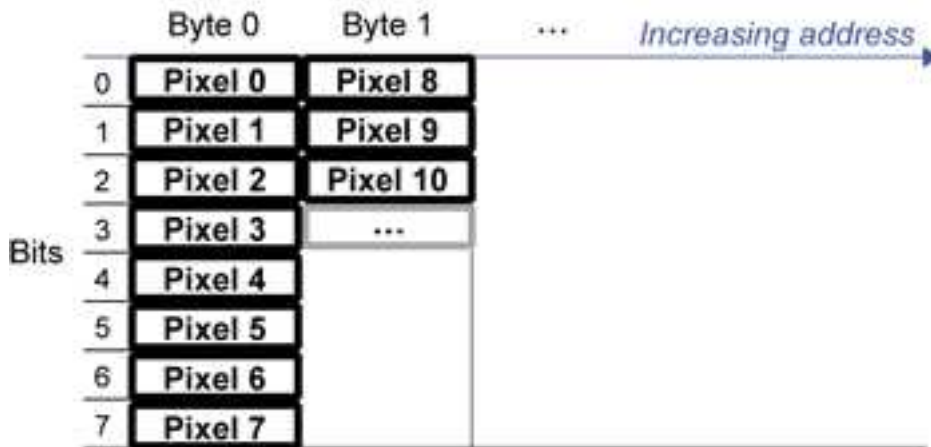
- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

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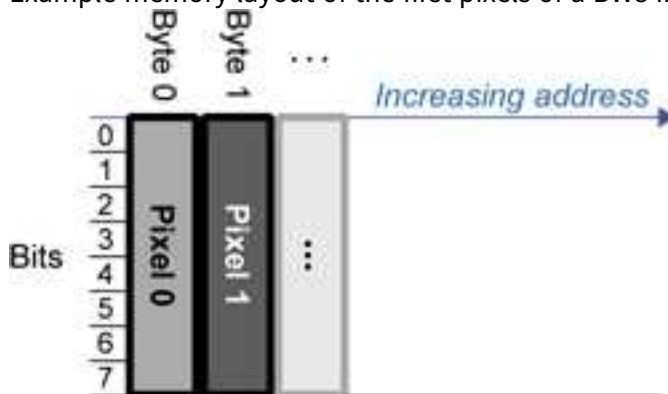
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).

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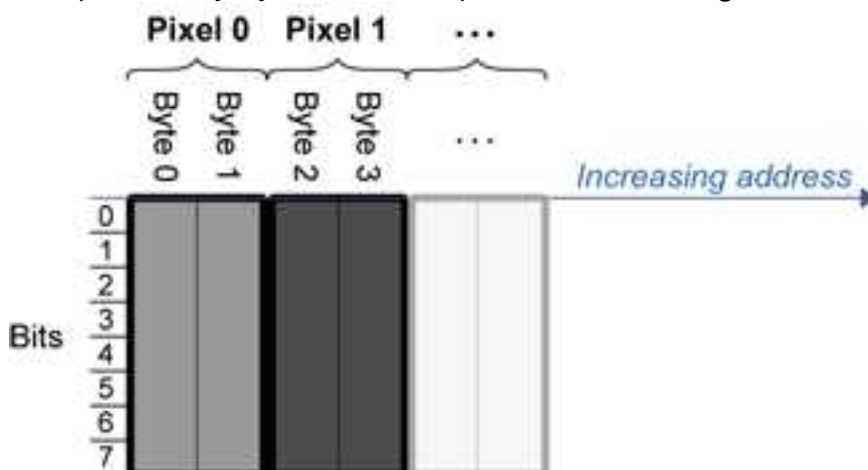
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

Example memory layout of the first pixels of a C15 image buffer:
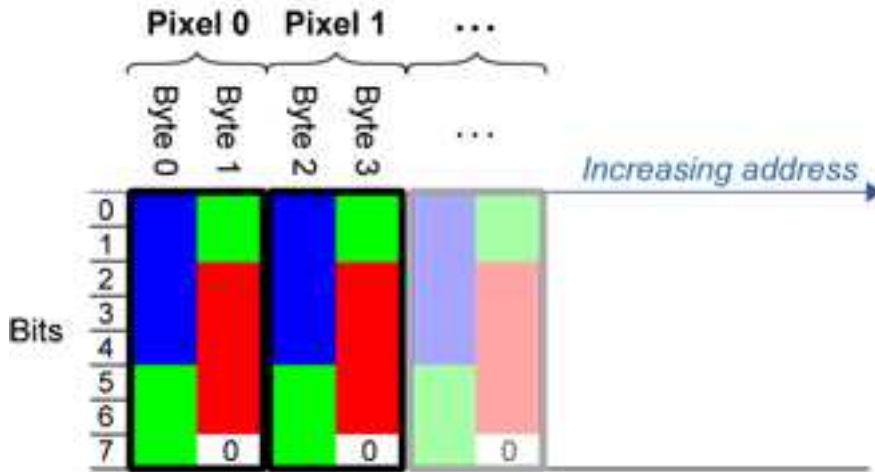
- **EImageC15** stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
  The second color component is coded with 6-bits.

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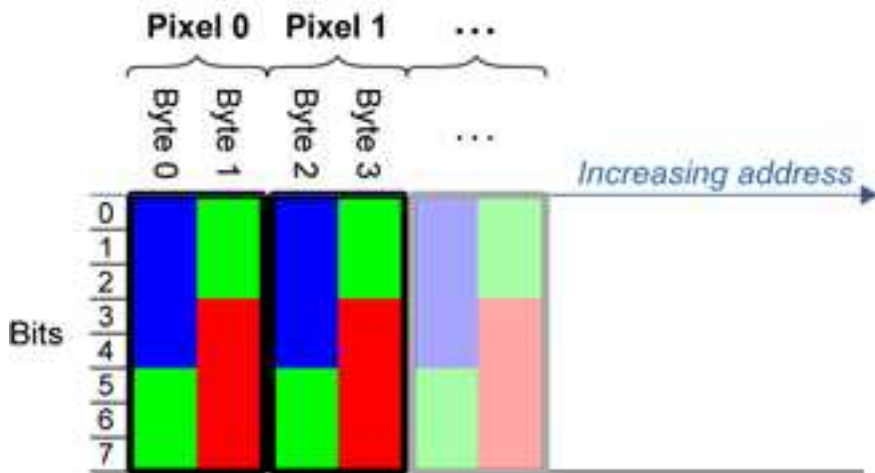
- **EDepthMap16** store each pixel in 2 bytes using a fixed point format.

- **EImageC24** stores each pixel in 3 bytes. Each color component is coded with 8-bits.

Example memory layout of the first pixels of a C24 image buffer:
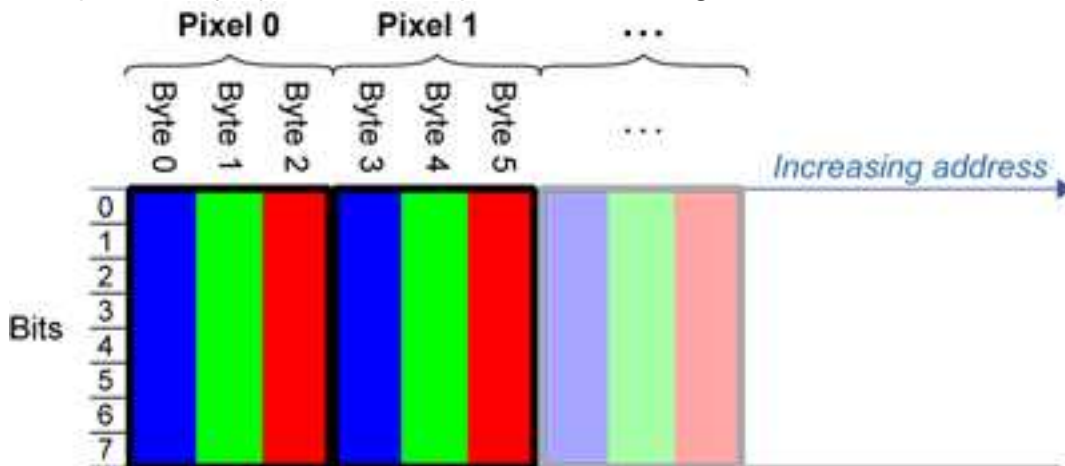
- `EImageC24A` stores each pixel in 4 bytes. Each color component is coded with 8-bits. The alpha channel is also coded with 8-bits.

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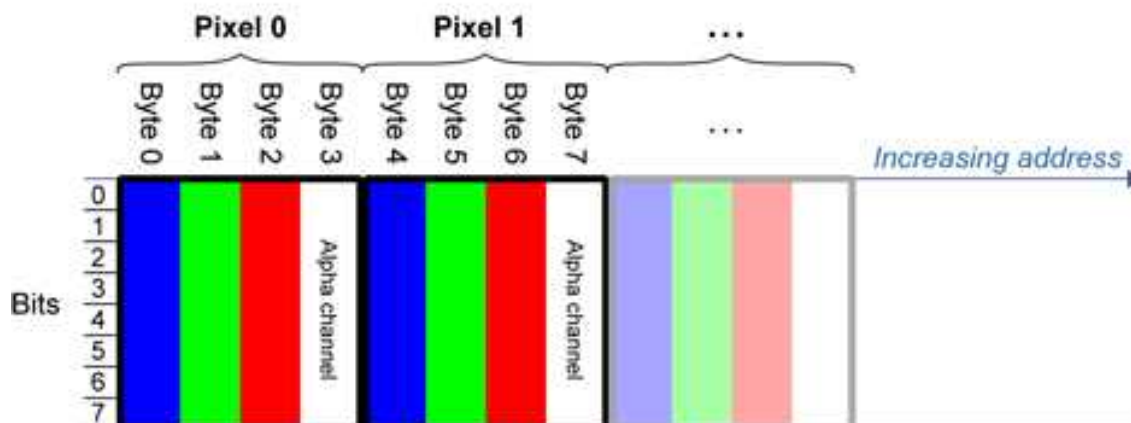
- `EDepthMap32f` store each pixel in 4 bytes using a float format.

2.5. Image Coordinate Systems

The conventions below apply to all Open eVision functions and results.
- □ Pixel coordinates are usually given as integer numbers.
- □ Some results can use subpixel precision with real (floating point) numbers.
- □ Some exceptions apply and are documented per librarie.

Integer coordinates

- The origin (0,0) of the coordinate system is the upper left pixel of the image.

- The lower right pixel is (width-1, height-1).

Buffer/Pixel Coordinates (integer)
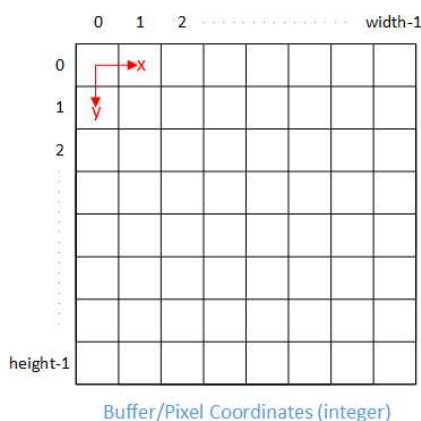
Real coordinates

- 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.

Image Coordinates (float)
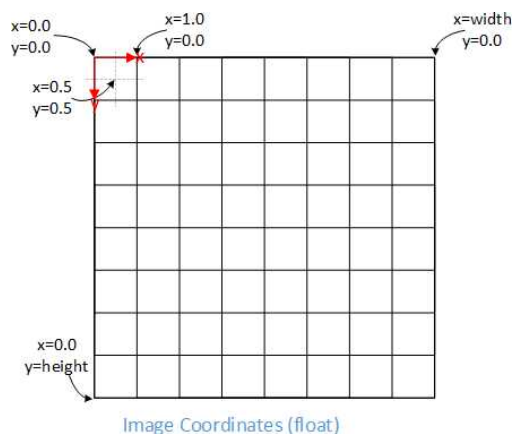
2.6. Image Drawing and Overlay

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

- The color palette in 256-color display mode gives optimal rendering.

- Gray-level images can be improved using LUTs (LookUp Tables) (using histogram stretching techniques or pseudo-coloring).

- The zoom can be different horizontally and vertically.

- `DrawFrameWithCurrentPen` method draws a frame.

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

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

2.7. 3D Rendering of 2D Images

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

Gray 3D rendering

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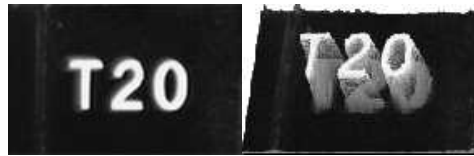
3D rendering

Color histogram 3D rendering

`Easy::RenderColorHistogram` prepares a 3-dimensional rendering of a color image histogram.
The pixels are drawn in the RGB space (not XY-plane) to show clustering and dispersion of RGB
values.
This function can process pixels in other color systems (using EasyColor to convert), but the raw
RGB image is required to display the pixels in their usual colors.

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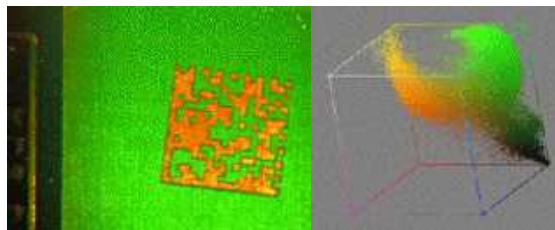
Color histogram rendering

2.8. Vector Types and Main Properties

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

EVector is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.
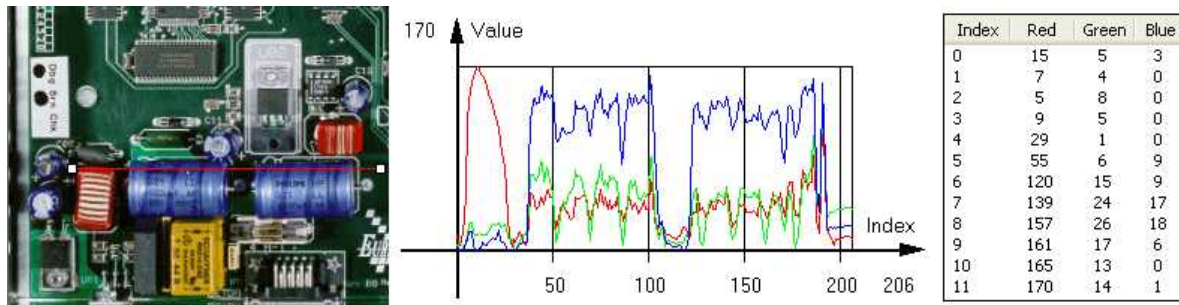
Profile in a C24 image, RGB values plot along profile and RGB values array (EC24Vector**)**

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

The use of vectors is quite straightforward:

- To create a vector of the appropriate type:
  - Use its constructor and preallocate elements if required.

- To fill a vector with values:
  - Call the EVector::Empty member to empty it.
  - Call the EC24Vector::AddElement member to add elements one by one.
  - Use the indexing to access any element.

- To access a vector element, either for reading or writing:
  - Use the brackets operator EC24Vector::operator[].

- To determine the current number of elements:
  - Use the EVector::NumElements member.

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

Vector types

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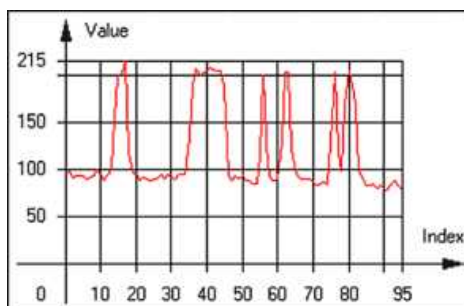
Graphical representation of an `EBW8Vector` (see `Draw` method)

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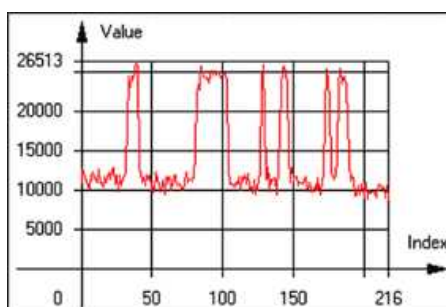
Graphical representation of an `EBW16Vector`

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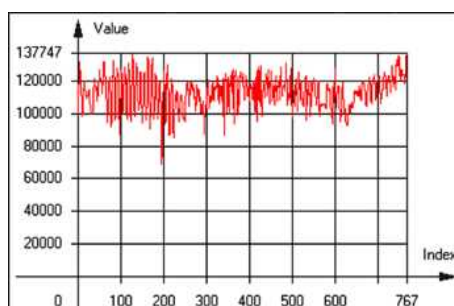
Graphical representation of an `EBW32Vector`

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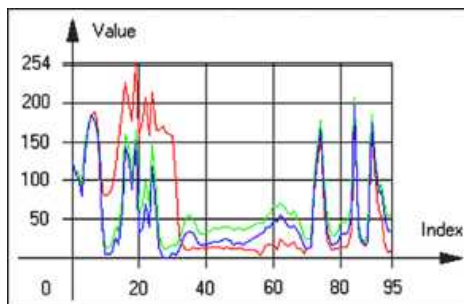
Graphical representation of an `EC24Vector`

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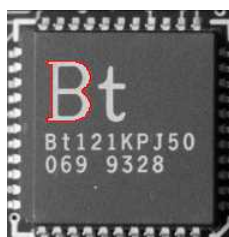
Graphical representation of an `EBW8PathVector` **(see** `Draw` **method)**

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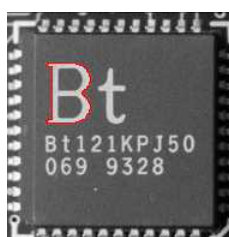
Graphical representation of an `EBW16PathVector` **(see** `Draw` **method)**

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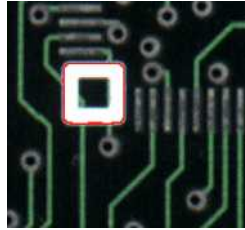
Graphical representation of an `EC24PathVector` (see `Draw` method)

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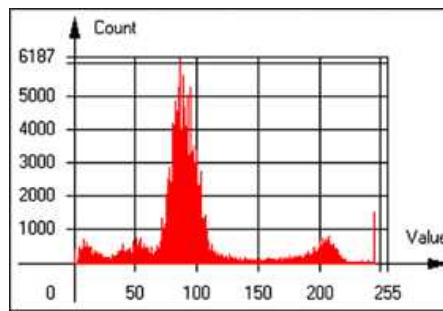
Graphical representation of an `EBWHistogramVector` (see `Draw` method)

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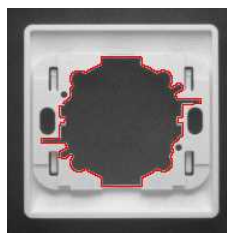
Graphical representation of an `EPathVector` (see `Draw` method)

- `EPeakVector`: peaks found in an image profile
  (used by `EasyImage::GetProfilePeaks`).

- `EColorVector`: a description of colors
  (used by `EasyColor::ClassAverages` and `EasyColor::ClassVariances`).

2.9. ROI Main Properties

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

Save and load

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

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

ROI Classes

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

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

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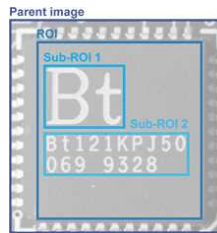
Nested ROIs: Two sub-ROIs attached to an ROI, itself attached to the parent image

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).

2.10. Arbitrarily Shaped ROI (ERegion)

> **See also:** example: Inspecting Pads Using Regions / code snippets: ERegion

Regions or arbitrarily shaped ROI

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

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

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

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

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

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

Creating regions

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

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

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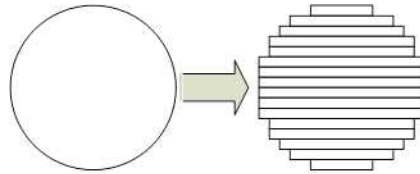
Run-Length Encoding of a circle-shaped region

To create a region, either:

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

Geometry-based regions

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

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

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

Rectangle region separating a bar code from the background

- **ECircleRegion**
  - ☐ The contour of an **ECircleRegion** class is a circle.
  - ☐ Define it using its center and radius or 3 non-aligned points.
  - ☐ Alternatively, use an **ECircle** instance, such as one returned by an **ECircleGauge** instance.
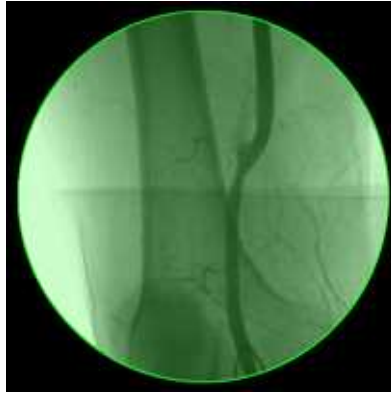
Circle region encompassing the useful part of an X-Ray image

- **EEllipseRegion**
  - ☐ The contour of an **EEllipseRegion** class is an ellipse.
  - ☐ Define it using its center, long and short radius and angle.
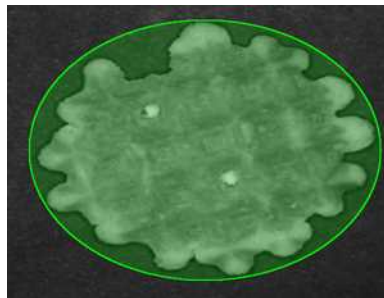
Ellipse region encompassing a waffle

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

Polygon region encompassing a key

Using the result of other tools

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

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

Open eVision provides constructors for the following tools:
- ☐ EasyFind: EFoundPattern
- ☐ EasyMatch: EMatchPosition
- ☐ EasyGauge: ECircle and ERectangle
- ☐ EasyObject: ECodedElement

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

Combining regions

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

- Union
  - ☐ The ERegion::Union(const ERegion&, const ERegion&) method returns the region that is the addition of the two regions passed as arguments.
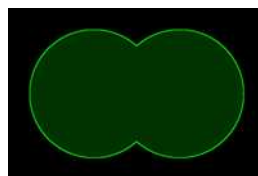
Union of 2 circles

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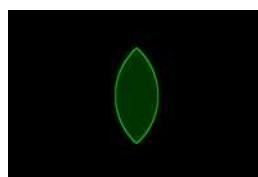
Intersection of 2 circles

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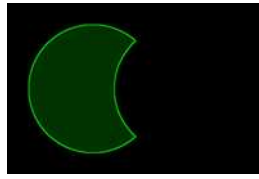
Subtraction of 2 circles

Morphological operations on regions
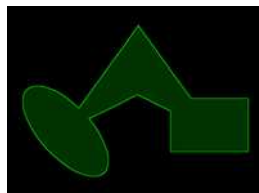
The initial arbitrary region used to illustrate the different morphological operations

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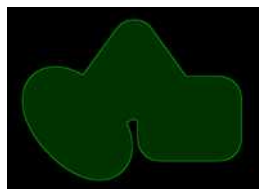
Grow of the arbitrary region

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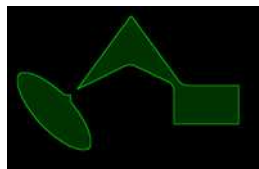
Shrink of the arbitrary region

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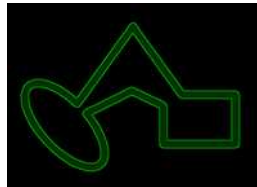
Contour of the arbitrary region

Free-hand drawing a region

- The `ERegionFreeHandPainter` class provides the methods that allow you to create a region by hand, using the mouse or any other user input method.

- The `RegionFreeHand` sample, available both in C++ and C#, shows how to use this class to draw a region on an image.

Using regions

The tools supporting regions provide methods that follow one of these conventions:
  - □ `Method(const EImage& source, const ERegion& region)`
  - □ `Method(const EImage& source, const ERegion& region, EImage& destination)`

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

Preparing the region

- Open eVision automatically prepares the regions when it applies them to an image, but this preparation can take some time.

- If you do not want your first call to a method to take longer than the next ones, you can prepare the region in advance by using the appropriate `Prepare()` method.

- To manually prepare the regions, adapt the internal RLE description to your images.

Drawing regions

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

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

- `ERegion::DrawContour()` draws the region contour in the provided device context.

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

ERegions and EROIs

- The older `EROI` classes of Open eVision are compatible with the new regions.

- Some tools allow the usage of regions with source and/or destinations that are `ERoi` instead of `EImage` follow one of these conventions:
  - ☐ `Method(const ERoi& source, const ERegion& region)`
  - ☐ `Method(const ERoi& source, const ERegion& region, ERoi& destination)`

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

ERegion and 3D

- The new regions are compatible with the 2.5D representations of Easy3D (`EDepthMap` and `EZMap`).

- You can also reduce the domain of processing when using these classes.

2.11. Flexible Masks

ROIs vs flexible masks

ROIs and masks restrict processing to part of an image:
  - ☐ apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
  - ☐ Flexible Masks are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some EasyObject and EasyImage library functions.

Flexible Masks

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

- ☐ All pixels of the flexible mask having a value of 0 define the ignored areas.
- ☐ All pixels of the flexible mask having any other value than 0 define the areas to be processed.
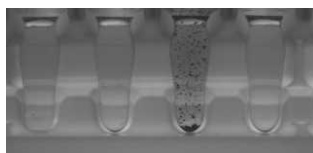
| Source image | Associated mask | Processed masked image |
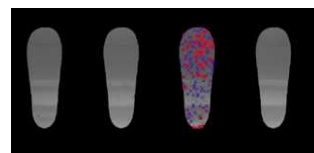
A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.

Flexible Masks in EasyImage
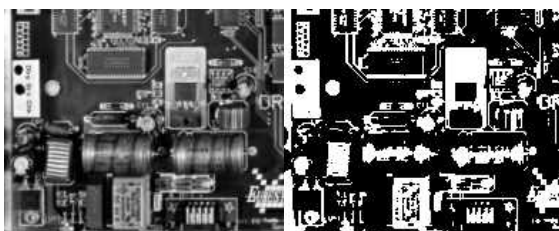
Code Snippets

Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

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

2. **Display the results.**
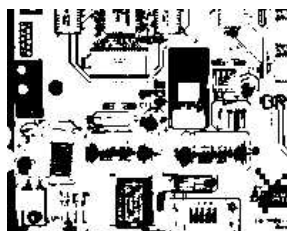
Resulting image

EasyImage Functions that support flexible masks

- `EImageEncoder.Encode` has a flexible mask argument for BW1, BW8, BW16, and C24 source images.

- AutoThreshold.

- Histogram (function `HistogramThreshold` has no overload with mask argument).

- RmsNoise, SignalNoiseRatio.

- Overlay (no overload with mask argument for BW8 source images).

- ProjectOnAColumn, ProjectOnARow (Vector projection).

- ImageToLineSegment, ImageToPath (Vector profile).

Flexible Masks in EasyObject

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

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

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

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

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

EasyObject functions that create flexible masks
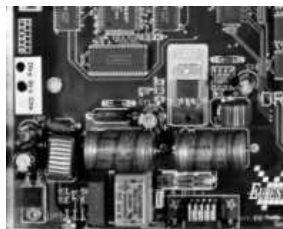
Source image

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

1. To encode and extract a flexible mask, first construct a coded image from the source image.

2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).

3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers using minimum residue thresholding).

4. Make the mask image the desired size using `mask.SetSize(sourceImage.GetWidth(), sourceImage.GetHeight())`.

5. Exploit the flexible mask as an argument to `ECodedImage2.RenderMask`.
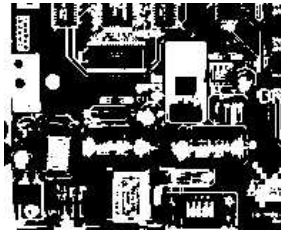
BW8 resulting image that can be used as a flexible mask

2) ECodedElement.RenderMask: from a blob or hole

1. Select the coded elements of interest.

2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement.RenderMask`.

3. Optionally, compute the feature value over each of these selected coded elements.
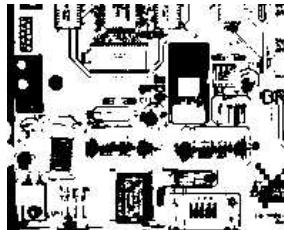
BW8 resulting image that can be used as a flexible mask

3) EObjectSelection.RenderMask: from a selection of blobs

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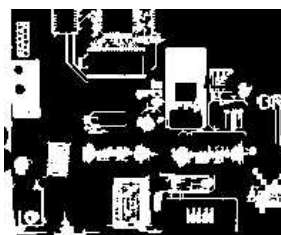
BW8 resulting image that can be used as a flexible mask

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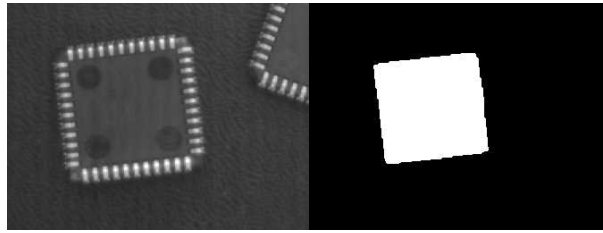
Find four circles (left) Flexible mask can isolate the central chip (right)

1. Declare a new `ECodedImage2` object.

2. Setup variables: first declare source image and flexible mask, then load them.

3. Declare an `EImageEncoder` object and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.

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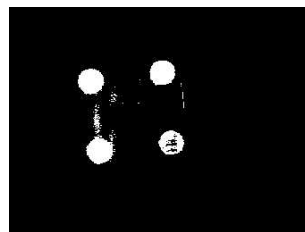
5. Select all objects of the coded image.

6. Select objects of interest by filtering out objects that are too small.

7. Display the blob feature by iterating over the selected objects to display the chosen feature.

2.12. Profile

Code Snippets

Profile Sampling

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

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

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

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

Profile Analysis

The profile can be processed to find peaks or transitions:

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

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

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

Profile Insertion Into an Image

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

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

3. Image Pre-Processing Libraries

3.1. 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.

Intensity Transformation

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

Gain offset

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

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

In this example, the resulting image has better contrast and is brighter than the source image.
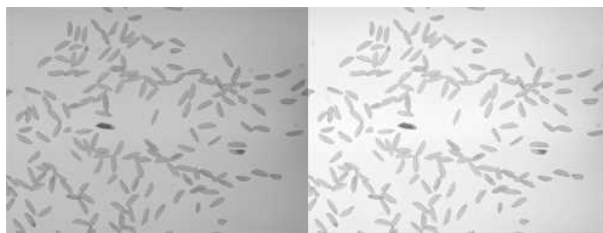
Source and result images (with gain = 1.2 and offset = +12)

Color images have three separate gain and offset values, one per color component (red, green, blue).
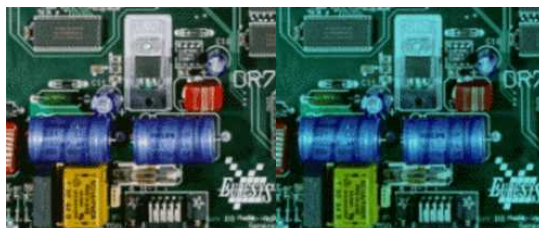
Example of gain/offset applied on a color image

Normalization

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

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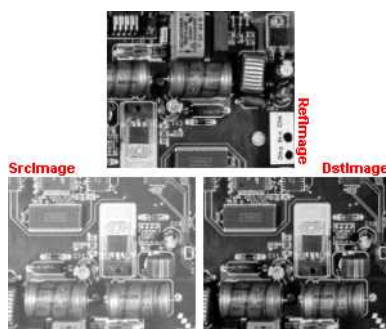
**The reference image (from which the average and standard deviation are computed),
the source image (too bright),
and the normalized image (contrast and brightness are the same as the reference image)**

Uniformization

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

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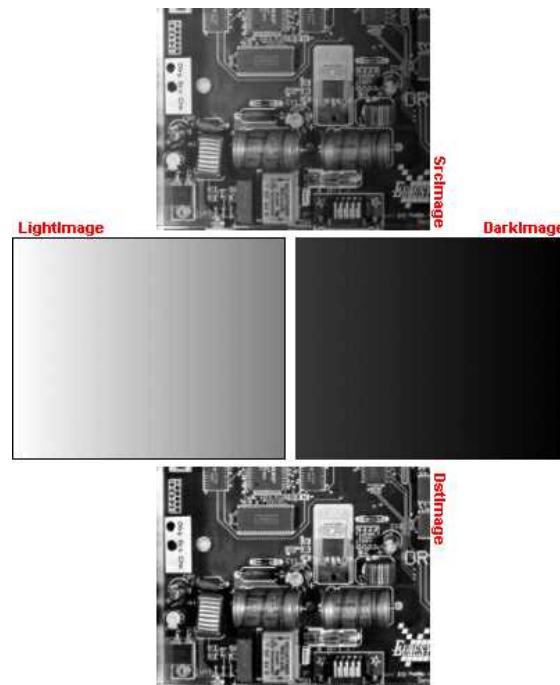
Example of an image uniformized with two reference images

Lookup tables

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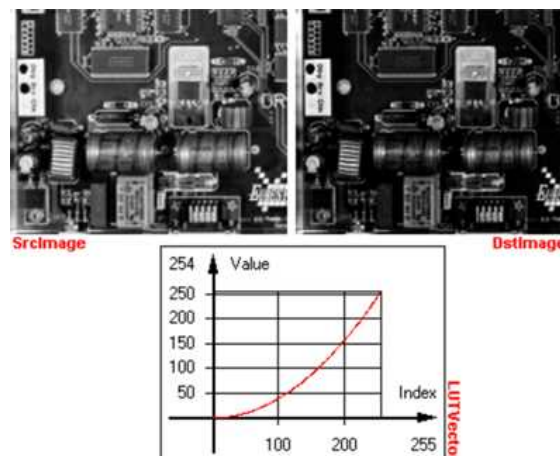
Example of a transform

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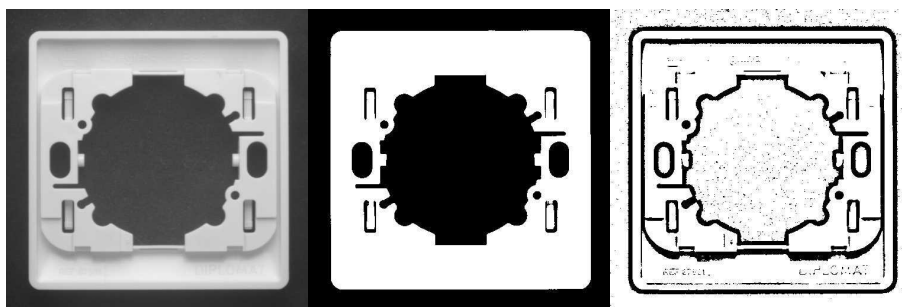
Thresholding transforms an image by classifying the pixel values using these methods:

- (BW8 and BW16 images only)
- (BW8 and BW16 images only)
- using one or two threshold values
- (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).
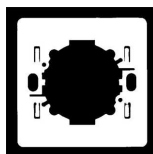
Max Entropy: Maximizes the entropy (that is, the amount of information) between object and background of the resulting image.
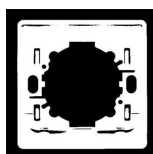
Isodata: Calculates a threshold value that is an average of the gray levels: halfway between the average gray level of pixels below the threshold, and the average gray level of pixels above the threshold.

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.

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

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

Non-Linear Filtering

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

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

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

1. They define the required shape by a "Kernel" on page 48 (usually in a 3x3 matrix).
2. They slide this Kernel over the image to determine the value of the destination pixel when a match is found:
   - Erosion, Dilation: shrinks / grows image regions.
   - Opening, Closing: removes / fills image region boundary pixels.
   - Thinning, Thickening: erodes / dilates using image pattern matching.
   - Top-Hat filters: retains all the tiny image details while removing everything else.
   - Morphological distance: indicates how many erosions are required to make a pixel black.
   - Morphological gradient: indicates the outer and inner edges of the erosion and dilation processes.
   - Median filter: removes impulsive noise.
   - Hit-and-Miss transform: detects patterns of foreground /background pixels, can create skeletons.

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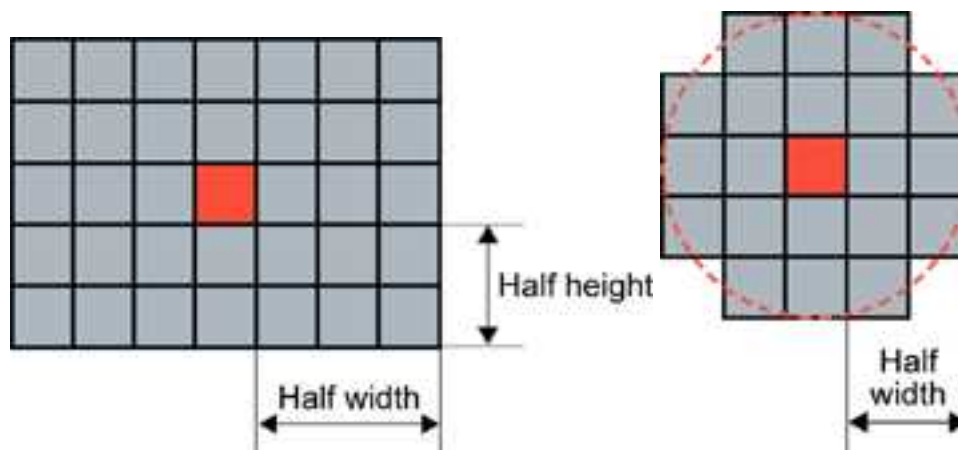
Rectangular kernel of half width = 3 and half height = 2 (left) Circular kernel of half width = 2 (right)

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

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

examples

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

Erosion, Dilation

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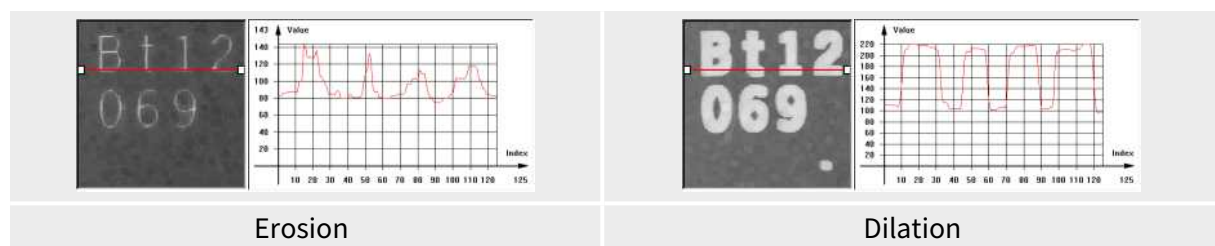
Erosion	Dilation

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

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

Opening, Closing

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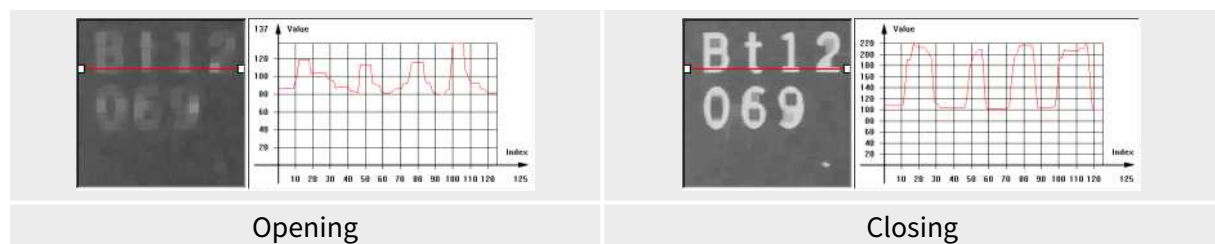
Opening	Closing

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

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

Thinning, Thickening

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

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

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

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

Top-Hat filters

Top-hat filters are excellent for improving non-uniform illumination.
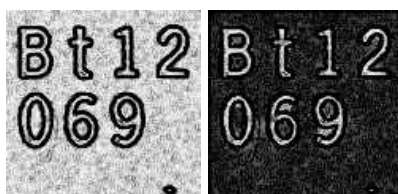
White top-hat filter: source and destination images

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

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

Morphological distance

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

Morphological gradient

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

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

`MorphoGradientBox`, `MorphoGradientDisk`.
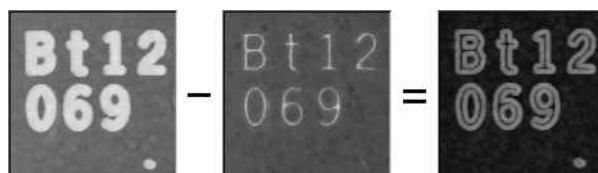
Dilation – Erosion = Gradient

Median

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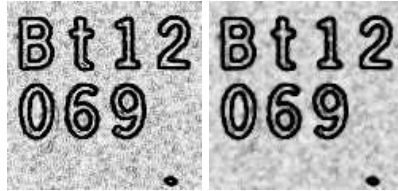
Median filter: source and destination images

Hit-and-Miss transform

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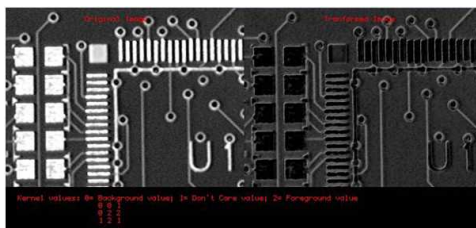
Hit-and-miss transform

The `HitAndMiss` function has three arguments:

- A pointer to the source image of type `EROIBW8`, `EROIBW 6`, or `EROIC24`
- A pointer to the destination image of type corresponding to the type of the source image. The sizes of the source and destination images must be identical.
- A kernel of type `EHitAndMissKernel` Two constructors are available for the kernel object:
  - `EHitAndMissKernel(int startX, int startY, int endX, int endY)` where:

    `startX`, `startY` are coordinates of the top left of the kernel, must be less than or equal to zero.

    `endX`, `endY` are coordinates of the bottom right of the kernel, must be greater than or equal to zero.

    The constructed kernel has no explicit restrictions on its size, and the following characteristics:

    `kernel width = (endX – startX + ⏎)`, `kernel height = (endY – startY + ⏎)`
  - `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) + ⏎)`, `kernel height = ((2 x halfSizeY) + ⏎)`

    `kernel StartX = - halfSizeX`, `kernel StartY = - halfSizeY`

example: detecting corners in a binary image.

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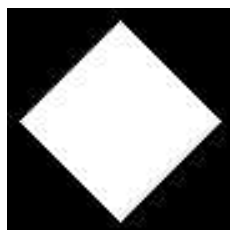
Binary source image

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

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

2. Apply the filter on the source image. Note that the resulting image should be properly sized.
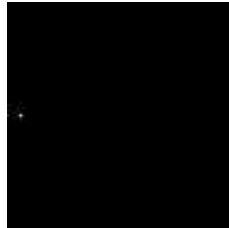
Resulting image, highlighted pixel is located on left corner of rhombus

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

4. Detect the right, top and bottom corners.
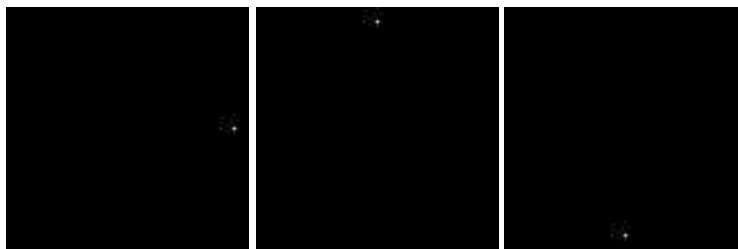
Geometric Transforms

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

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

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

Possible geometric transformations are:

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.
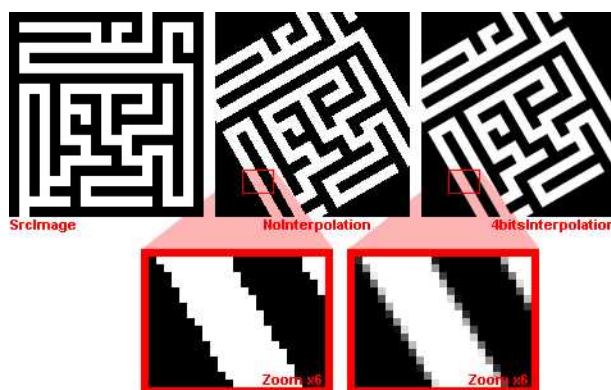
Scale and rotate example

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.

Noise Reduction and Estimation

Code Snippets

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

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

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

Spatial noise reduction (if you only have 1 image)

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

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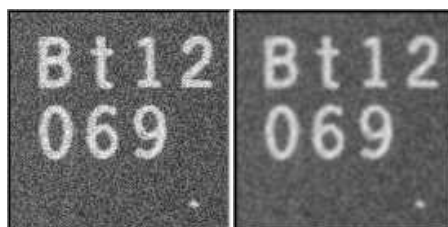
Uniform noise reduction by low-pass filtering

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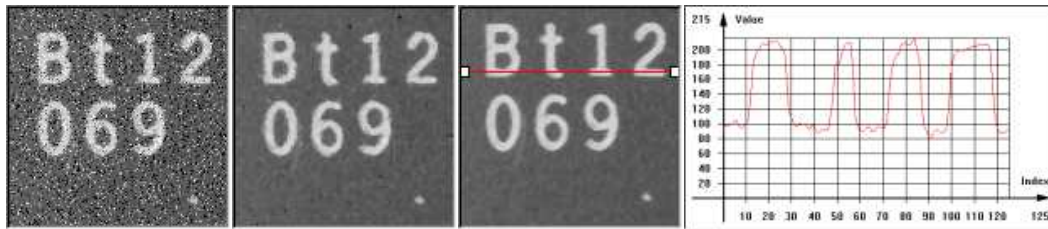
Impulse noise reduction by median filtering

- EasyImage.Median
- EasyImage.BiLevelMedian

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

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

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

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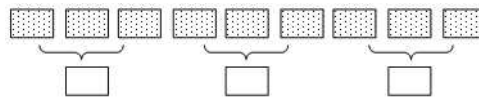
Simple average

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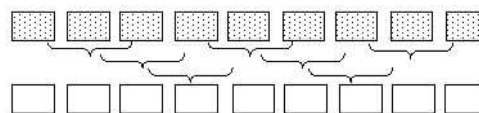
Moving average

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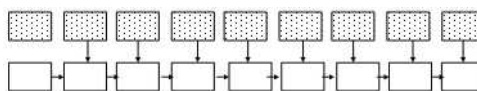
Recursive average

Recursive averaging

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

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

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

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

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

For optimal performance, the non-linearity must be precomputed once for all using function EasyImage.SetRecursiveAverageLUT.

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

Noise estimation (of image compared to reference image):

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

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

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

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

Scalar Gradient

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

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

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

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

Vector Operations

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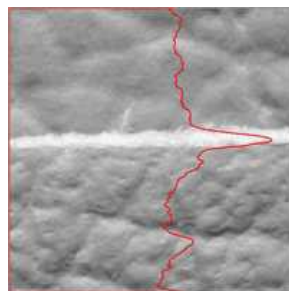
Projects the sum or average of all gray color-level values in a given direction, into various vector types (levels are added when projecting into an `EBW32Vector` and averaged when projecting into an `EBW8Vector`, `EBW16Vector` or `EC24Vector`). These functions support **flexible masks**.

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

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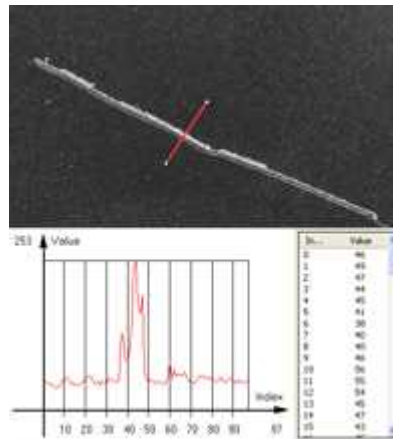
Samples a series of pixel values along a given segment, path or contour, then analyze and modify their Peaks and Transitions to make images clearer:

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.

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
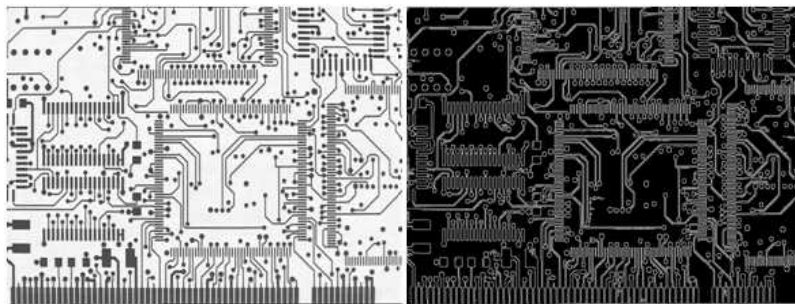
Source image and the result after a Canny edge detection

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.
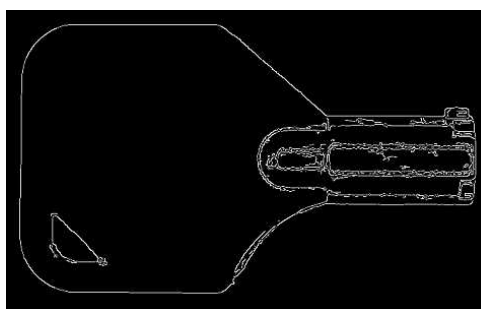
Canny edge detector example

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.

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.
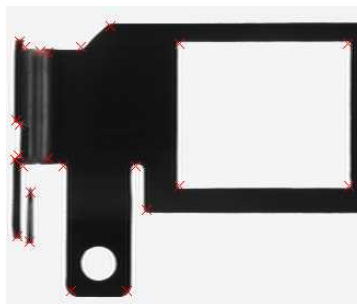
Harris corner detector example

The EasyImage Harris corner detector requires three parameters:

- The integration scale σi: the standard deviation of the Gaussian Filter used for scale analysis.
  σd = 0,7 x σi, where σd is the differentiation scale: the standard deviation of the Gaussian Filter used for noise reduction during computation of the gradient.
- A corner threshold: a fraction ranging from 0 to 1 of the maximum value of the cornerness of the source image.
- A Boolean that toggles sub-pixel detection.

The following characteristics are available for every point of interest:

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

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

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

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

Basic usage of Harris Corner Detector

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

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

Overlay

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

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

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

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

Operations on Interlaced Video Frames

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

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

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

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

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

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

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

The size of the destination image is determined as follows:

$$dstImage\_Width = srcImage\_Width$$
$$dstImage\_Height = (srcImage\_Height + 1 - odd)/2$$
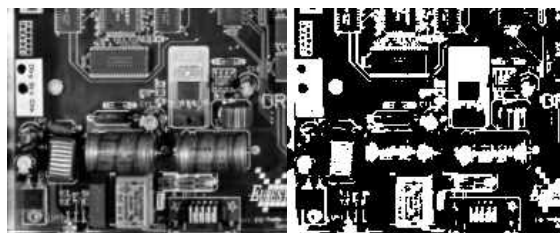
Flexible Masks in EasyImage

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.**
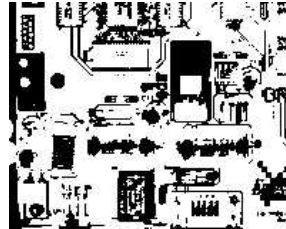
Resulting image

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).

Computing Image Statistics

Code Snippets

EasyObject statistics are related to the objects in an image.

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

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

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

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

The result of the operation is another image.

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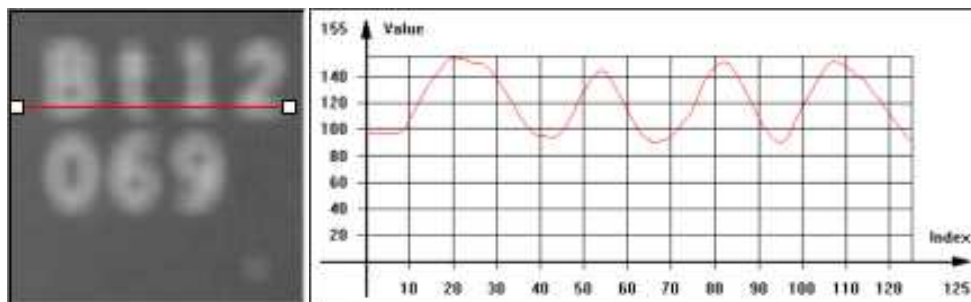
Sliding window average

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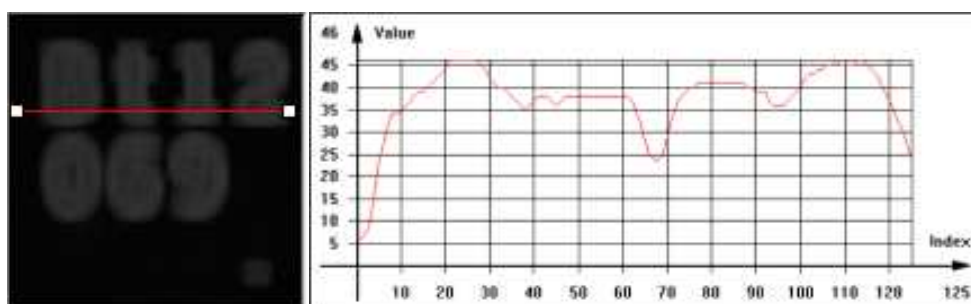
Sliding window standard deviation

Histogram computation and analysis(and LUT creation)

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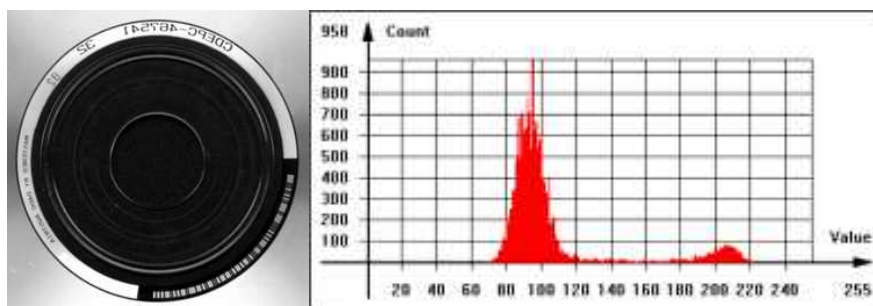
Typical image histogram

Histogram Computation

`EasyImage.Histogram` computes the histogram of an image. It can take a flexible mask as input argument.
BW8, BW16 and BW32 source images are supported.

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

Histogram Analysis

`EasyImage.AnalyseHistogram` and
`EasyImage.AnalyseHistogramBW 6` provide statistics and thresholding values:

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

Histogram equalization

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

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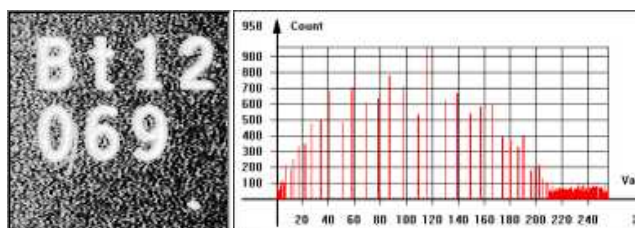
Equalized image and histogram

Setup a lookup table

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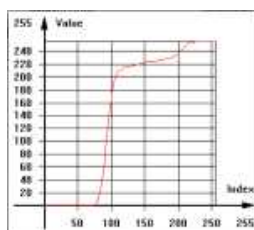
Equalization lookup table

Image focus

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

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

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

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

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

    b. Averages the squared pixel values over both images.

    c. Sums the averages.

    d. Takes the square root of the resulting value.

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

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

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

EasyImage statistics functions

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

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

Binary and weighted moments (object position and extent)

- `EasyImage.BinaryMoments` computes the 0th, 1st or 2nd order moments on a binarized image, i.e. with a unit weight for those pixels with a value above or equal to the threshold, and zero otherwise. It provides information such as object position and extent.

- `EasyImage.WeightedMoments` computes the 0th, 1st, 2nd, 3rd or 4th order weighted moments on a gray-level image. The weight of a pixel is its gray-level value. It provides information such as object position and extent.

Gravity center (average pixel coordinates above/on threshold)

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

Pixel count (between 2 thresholds)

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

Minimum, maximum and average gray-level value

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

Average, variance and standard deviation

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

Number of different pixels by comparing 2 images

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

3.2. 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.
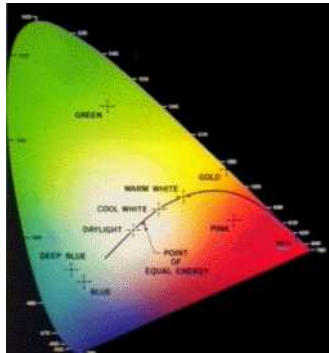
Visible color gamut in the XYZ color space

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 75: `EasyImage::GainOffset`,
- □ "LUT for Color Calibration" on page 76: `EColorLookup::Calibrate`,
- □ "LUT for Color Balance" on page 76: `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.
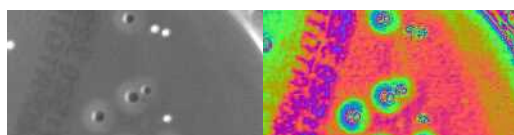
Gray-level and pseudo-colored image

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.
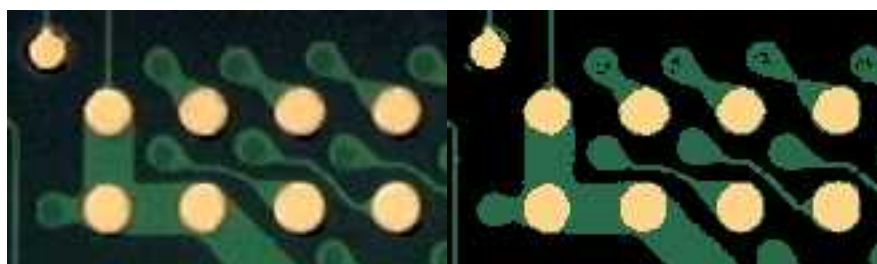
Raw image and segmented image (3 colors)

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:

Red-Green
EBayerConfiguration_RG

Green-Blue
EBayerConfiguration_GB

Green-Red
EBayerConfiguration_GR

Blue-Green
EBayerConfiguration_BG
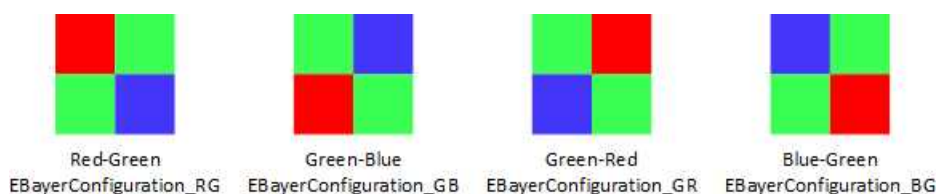
Several methods are available to reconstruct the missing pixels.
- Some are fast but the resulting image may have artifacts, like the zipper effect or color aliasing.
- Some are slower but achieve better interpolation and produce less artifacts.

Interpolation modes

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

- Mode 0
  - No interpolation
  - Frame rate: 943
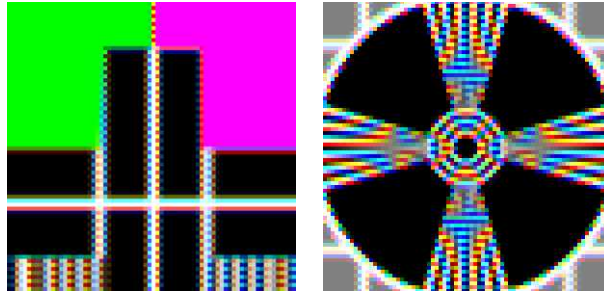
- Mode 1
  - Linear interpolation on a 3x3 kernel
  - Frame rate: 2159
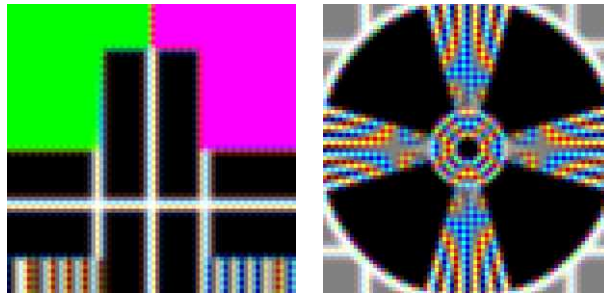
- Mode 2
  - Advanced interpolation on a 3x3 kernel
  - Frame rate: 1303
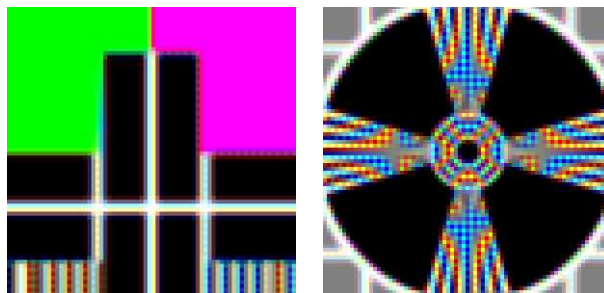
- Mode 3
  - Interpolation on a 5x5 kernel
  - Frame rate: 449
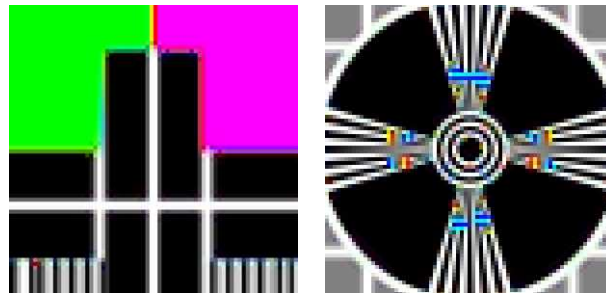
- Mode 4
  - Interpolation on 9x9 kernel
  - Frame rate: 22
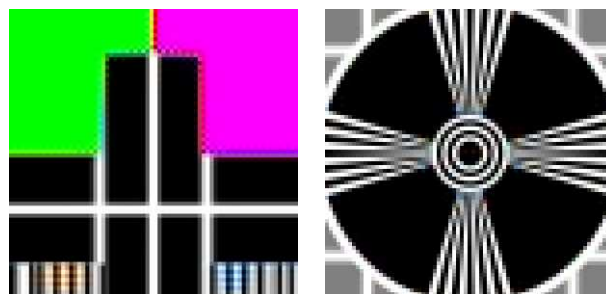
The method `EasyColor::C24ToBayer` is the reciprocal function of `EasyColor::BayerToC24`. It converts RGB color pixel images to Bayer images using the given `EBayerConfiguration`.

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

LUT for Gain/Offset (Color)

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

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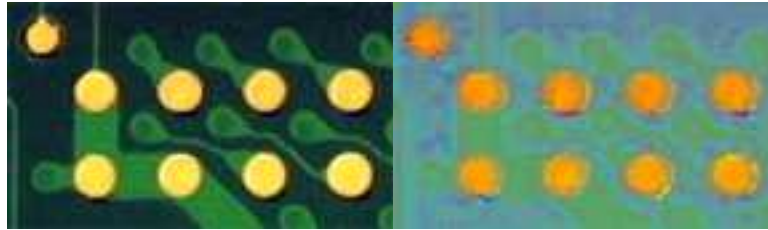
Enhanced saturation / Uniform lightness

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

LUT for Color Calibration

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

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

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

LUT for Color Balance

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

These effects can be corrected efficiently by setting up a lookup table using `WhiteBalance` and applying it on a series of images by means of `Transform`. The LUT 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.
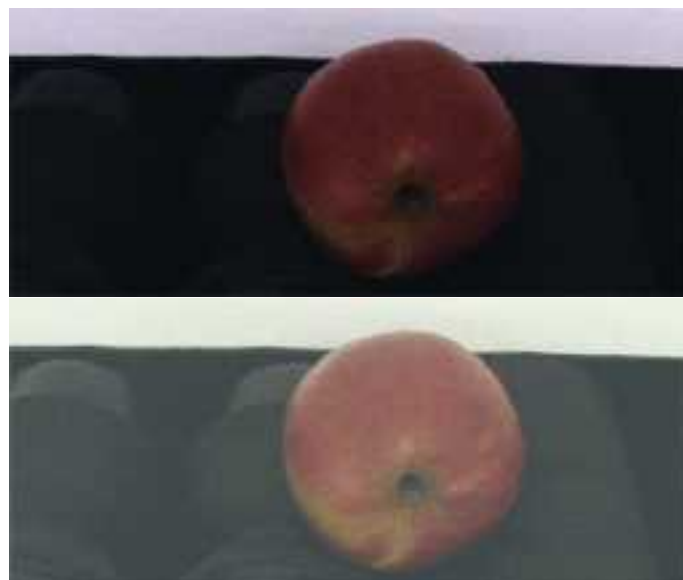
Raw image, and image with white balance and gamma precompensation

4. Using Open eVision Studio

4.1. Selecting your Programming Language

When you start Open eVision Studio for the first time, the following welcome screen is displayed:
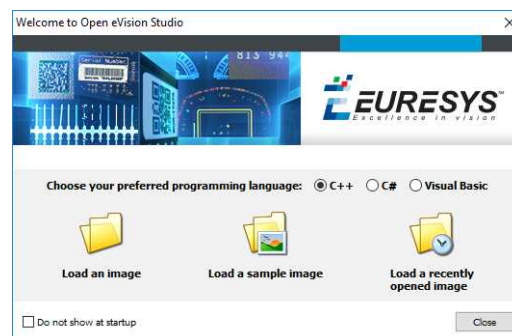
1. Select your programming language.

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

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

2. Click on one of the `Load` buttons to already load one or several images for later processing.

3. Check the `Do not show at startup` box to hide this welcome screen next time you start Open eVision Studio.

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

4.2. Navigating the Interface
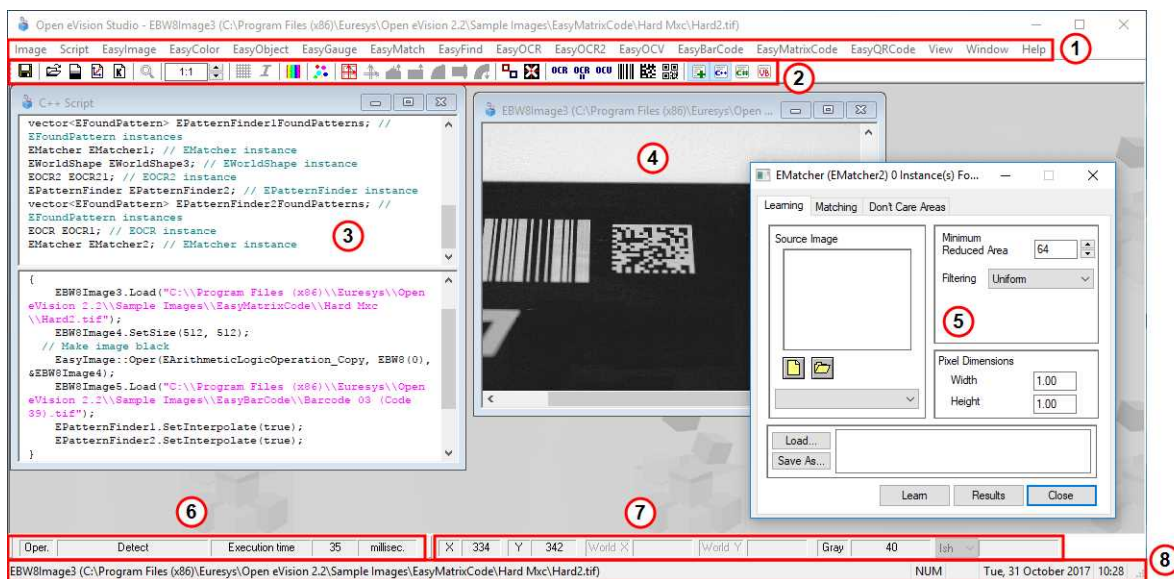
Open eVision Studio graphical user interface (GUI) is organized as follows:

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

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

2. The *main toolbar* gives you quick access to main Open eVision objects such as images, shapes, gauges, bar codes, matrix codes...

3. The *script window* displays the code, in the programming language you selected, corresponding to the actions you perform in Open eVision Studio. You can save or copy this code in your own application at any time.

4. The *image windows* display the open images that you can process using the libraries and tools.

5. The *tool windows* enable you to easily configure all the available tools. The corresponding settings are automatically added in the script window for easy reuse.

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

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

7. The *color toolbar* displays current information such as the X and Y coordinates of the cursor on an image and the corresponding pixel value.

8. The *status bar* displays general information about the application such as the active image file path...

4.3. Running Tools on Images

Step 1: Selecting a Tool

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

To do so:

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

2. Click on the tool you want to use.

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

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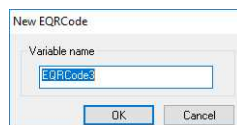
Example of variable creation dialog box for EasyQRCode

4. Click `OK`.

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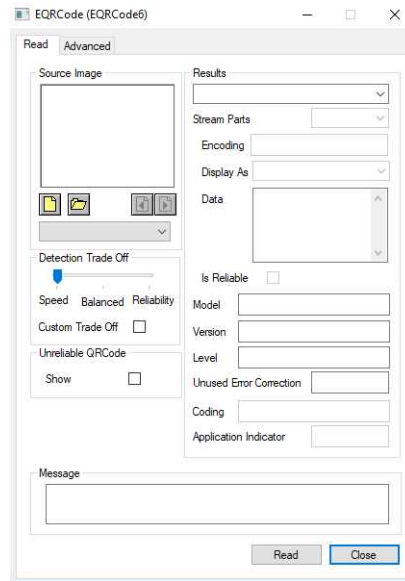
Example of variable creation dialog box for EasyQRCode

The next step is .

Step 2: Opening an Image

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

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

1. Open an image:

  □ Click on the `Open an Image` button and select one or several (using SHIFT and CTRL) images on your computer.

  □ Or select one of the images (or one of the ROIs, if any) already open in the drop-down list.

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

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

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

The next step is "Step 3: Managing ROIs" on page 83.

Step 3: Managing ROIs

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

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

Creating a ROI

1. Open the image:
   - □ If the image is already open, activate the corresponding image window.
   - □ If the image is not open yet, go to the main menu: `Image` > `Open...` to open one.

2. To create an ROI, go to the main menu: `Image` > `ROI Management....`

The `ROI Management` window is displayed as illustrated below.
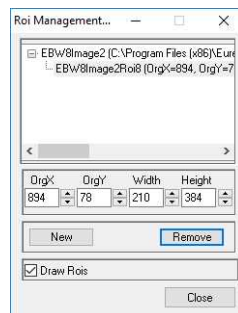
3. Select the image in the tree.

4. Click on the `New` button.

5. In the dialog box, enter a `Variable name` for the new ROI.

The ROI is represented as a color rectangle on your image as illustrated below.
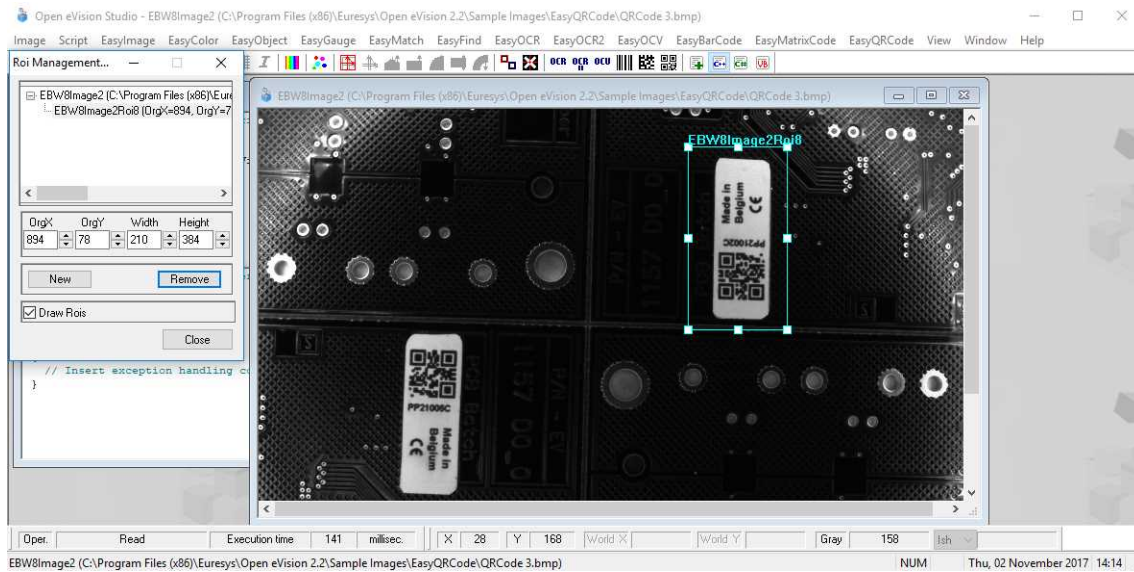
6. Drag the ROI corner and side handles to move it to the required position.

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

The next step is "Step 4: Configuring the Tool" on page 85.

Managing ROIs

You can add, change and remove ROIs.

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

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

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

If the `Draw Rois` box is checked, all ROIs are displayed on the image with a different color.
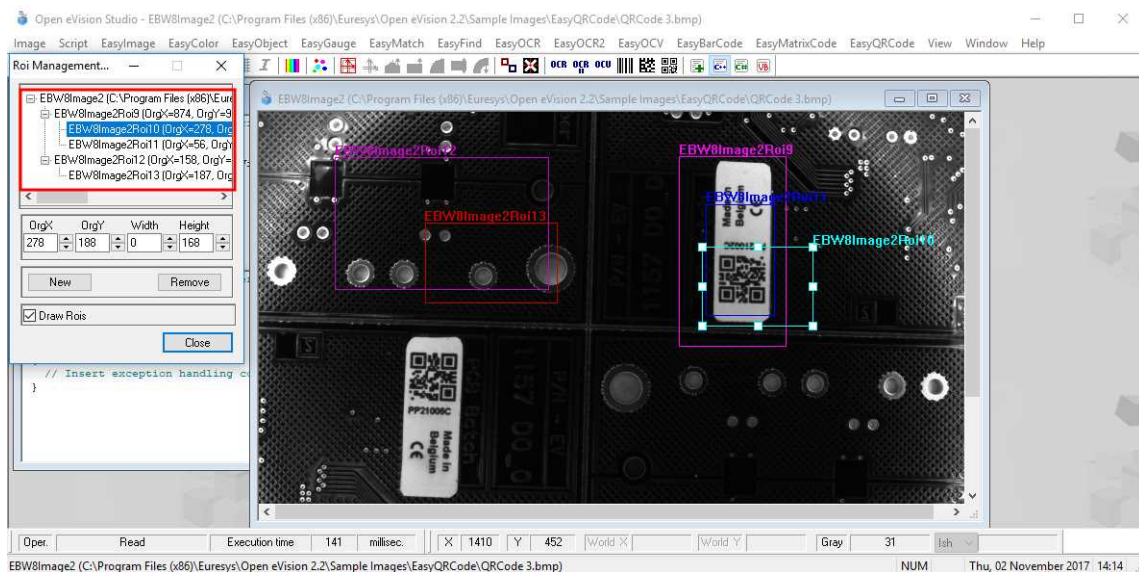
2. Select an ROI in the ROI relation tree.

3. Drag the ROI corner and side handles to change the position and size of the selected ROI (as well as the position of all ROIs attached to it if any).

4. Click on the `New` button to add a new ROI attached to the selected ROI.

> ✓ **TIP**
> Select the image at the top of the ROI relation tree to attach the ROI directly to the image.

5. Click on the `Remove` button to delete the selected ROI (and all ROIs attached to it if any).

6. Click on the `Close` button to close the `ROI Management` window.

Step 4: Configuring the Tool

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

In the tool window:

1. Open the various tabs.

> ✓ **TIP**
> When you create a new tool, all parameters are set with their default value.
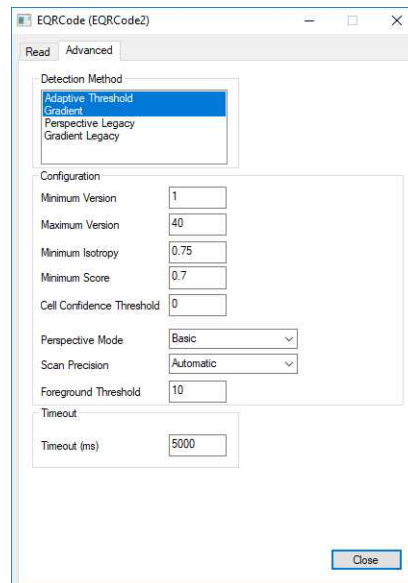
Example of the parameter tab of an EasyQRCode tool

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

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

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

3. Run the tool and analyze the results as described in the next step "Step 5: Running the Tool and Checking Execution Time" on page 86.

Step 5: Running the Tool and Checking Execution Time

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

In the tool window:

1. Click on the Read, Detect, Results or Execute button (depending on the library function), to run the tool on the selected image.

2. Check the results on the image and in the Results field or area as illustrated below.
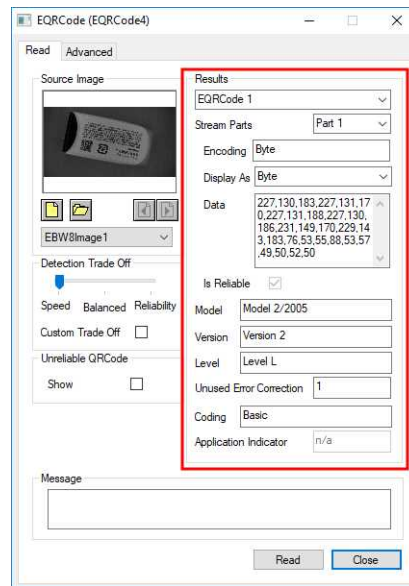
Example of results after reading a QRCode

3. If you do not have the expected results:
   □ Try to change your parameters (start with default values then change one parameter at a time).
   □ If your image is not good enough, try to enhance it as described in .

4. Check the execution time in the execution time bar at the bottom left of the main Open eVision Studio window.
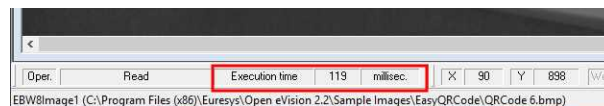
The execution time
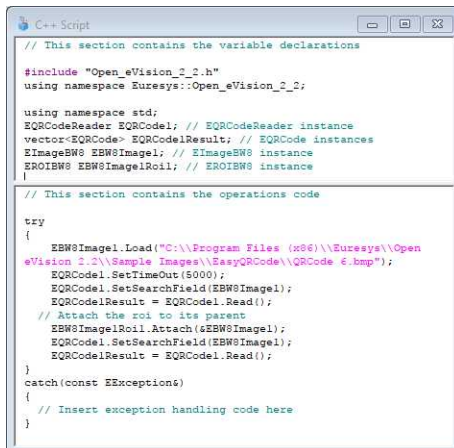
> ✅ **TIP**
>
> The execution time is the actual time that the processing took as measured on your computer. It depends your computer processor, memory, operating system... and, of course, on the processor load at the time of execution. Thus this execution time slightly varies from execution to execution.

5. To get a more representative execution time, click on the Read, Detect, Results or Execute button several times and calculate the mean execution time.

6. If your application requires that you reduce the execution time, try:
   □ To change the tool parameters,
   □ To add one or several ROIs on your image,
   □ To enhance your image.

The next step is "Step 6: Using the Generated Code" on page 88.

Step 6: Using the Generated Code

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

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

Copy and paste the code in your application

In the script window:

1. Select the code section you want to copy.

2. Right click on this code and click `Copy` in the menu.

3. Go to your development environment tool and paste the code in place.

Save the code

1. Go to the **Script** menu.

2. Click on `Save Script As....`

3. Enter a file name and path to save the code as a text file.

Manage the generated code

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

4.4. Pre-Processing and Saving Images

When should you pre-process your images?

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

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

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

Pre-processing images

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

To pre-process an image or an ROI:

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

2. Click on the function you want to use.

Most function dialog boxes are similar to the one illustrated below with 2 image selection areas and a parameter setting area.
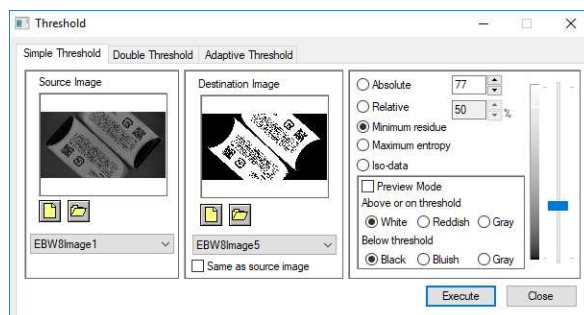
Example of a pre-processing dialog box (Threshold with EasyImage)

3. If there are multiple versions for your selected function, open the corresponding tab.

4. In the `Source Image` area, open the source image (as described in "Step 2: Opening an Image" on page 82).

5. In the `Destination Image` area, open or create a new destination image.

6. Set your parameters.

7. Click on the `Execute` button.

The pre-processed image is available in the destination image as illustrated below.
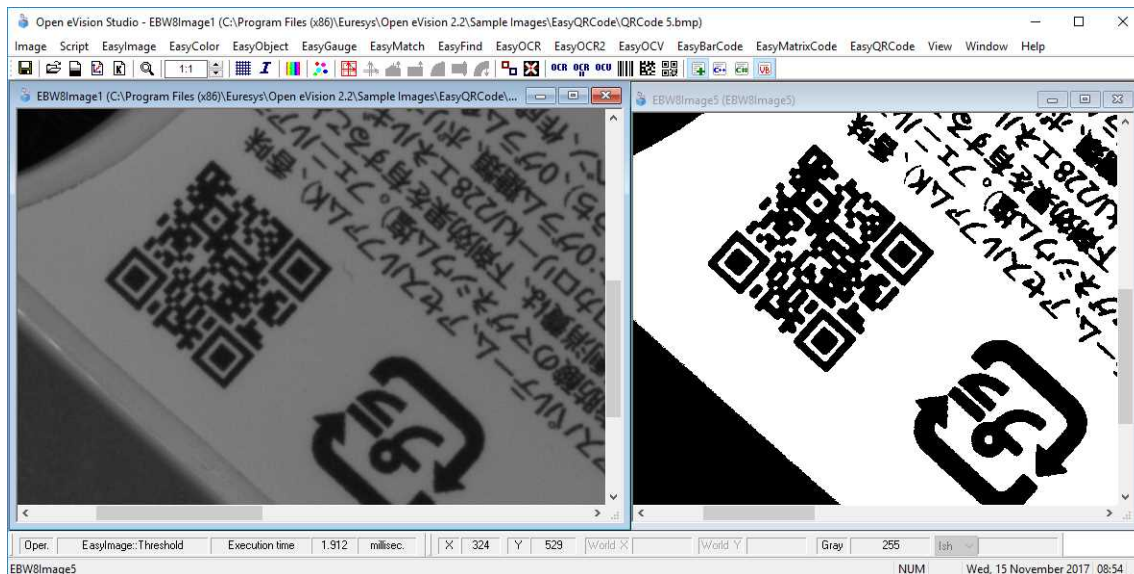
Source and destinations images (Threshold with EasyImage)

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

Saving an image

1. Click on the image you want to save to makes its window active.

2. To open the save menu either:
   □ Right-click in the image
   □ Or open the main menu > Image

3. Click on Save as....

4. Select the file format (JPEG, JPEG2000, PNG, TIFF or Bitmap).

5. Enter a name and select a path.

6. Click on the Save button.

5. Tutorials

5.1. EasyImage

Converting a Gray-Level Image into a Binary Image

Objective

Following this tutorial, you will learn how to use EasyImage to convert a gray-level source image into a binary destination image. Thresholding an image transforms all the gray pixels into black or white pixels, depending on whether they are below or above a specified threshold. Thresholding an image makes further analysis easier.

You'll need first to load an image (step 1). Then you'll set the thresholding parameters (step 2), and perform the conversion (step 3).

Gray-level source image
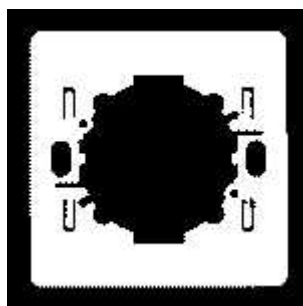
Black and white destination image, after thresholding

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Threshold`.

2. In the `Simple Threshold` tab, click the `Open` icon of the Source Image area, and load the image file `EasyMatch\Switch▸.tif`.

3. Keep the default variable name for the new Image object, and click `OK`.

Step 2: Set the thresholding parameters

1. In the right area of the `Threshold` dialog box, move the slider to change the threshold, and see directly in the source image a preview of the result.

2. Select the `Minimum residue` option to set a pre-defined algorithm that finds automatically the right threshold.

Step 3: Perform the conversion

1. Click the `New` icon in the `Destination Image` area to create a new destination image.

2. Keep the default settings for the new Image object, and click `OK`.

3. In the `Threshold` dialog box, click `Execute` to perform the thresholding in the destination image.

Extracting an Object Contour

Objective

Following this tutorial, you will learn how to use EasyImage to trace an object outline in a gray-level image. The contour extraction allows you to get in a path vector all the points that constitute an object contour, just by clicking an edge of this object.

You'll need first to load an image (step 1) and set a vector that will contain all the contour points (step 2). Then you'll click an object edge, and the contour will be extracted automatically (step 3).
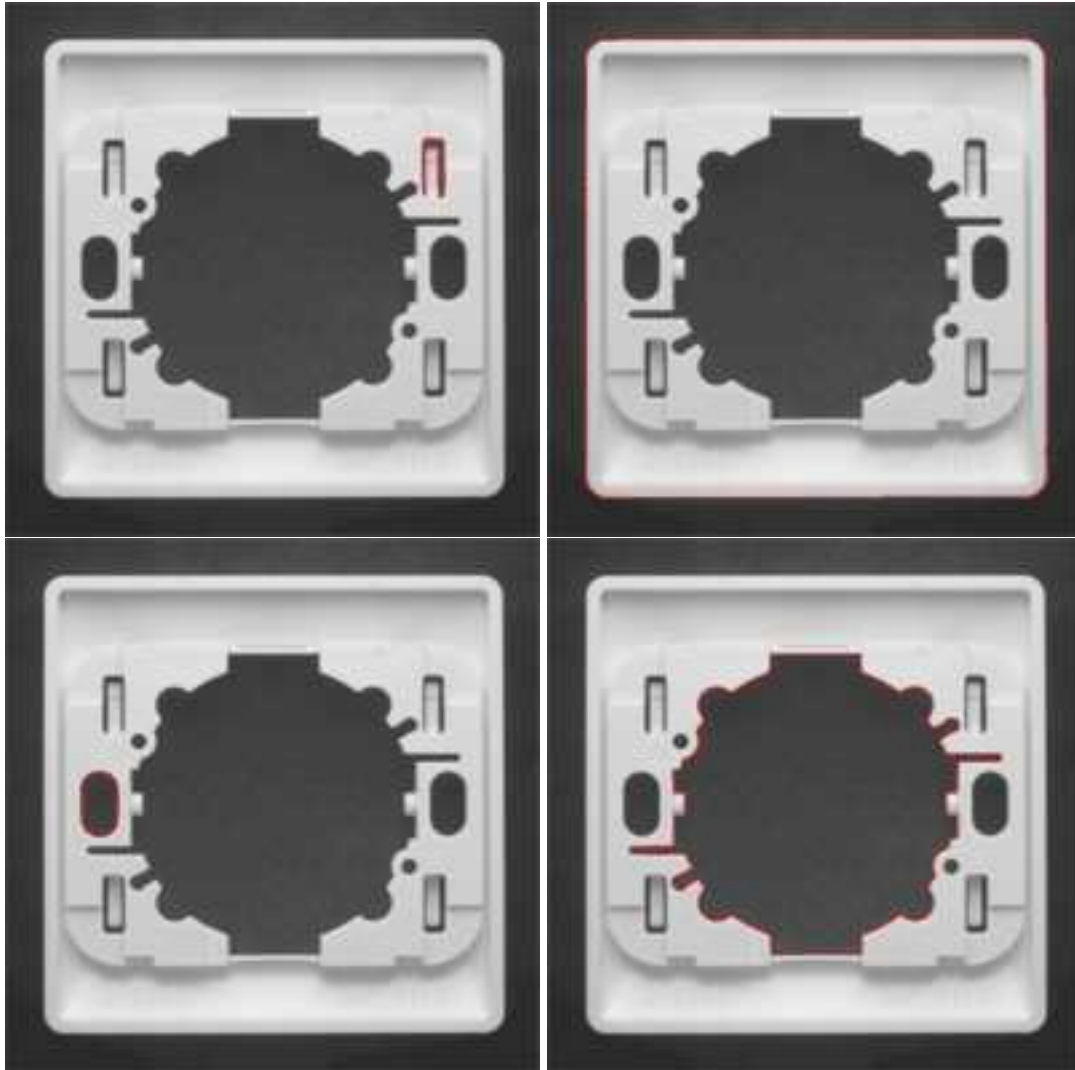
Contours are extracted from object edges

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Contour`.

2. Click the `Open` icon of the Source Image area, and load the image file `EasyMatch\Switch⌐.tif`.

3. Keep the default variable name for the new image object, and click `OK`.

Step 2: Set the destination vector

1. Click the `New` icon in the Destination Vector area.

2. Keep the default settings and variable name for the new vector object.

3. Click `OK`.

Step 3: Extract the contour

1. When moving the cursor above the image, an arrow appears.

2. Move the arrow above an object edge, and click.

From this object edge, a contour is traced, and a red line appears around the object.

The destination vector is filled with the points constituting the contour.

Transforming a Gray-Level image into its Black and White Edges

Objective

Following this tutorial, you will learn how to use EasyImage to transform a gray-level image to a binary image, keeping only the edges detected in the image. The conversion uses the Canny edge detector algorithm.

You'll need to load a source image (step 1), and simply apply the Canny edge transformation (step 2).
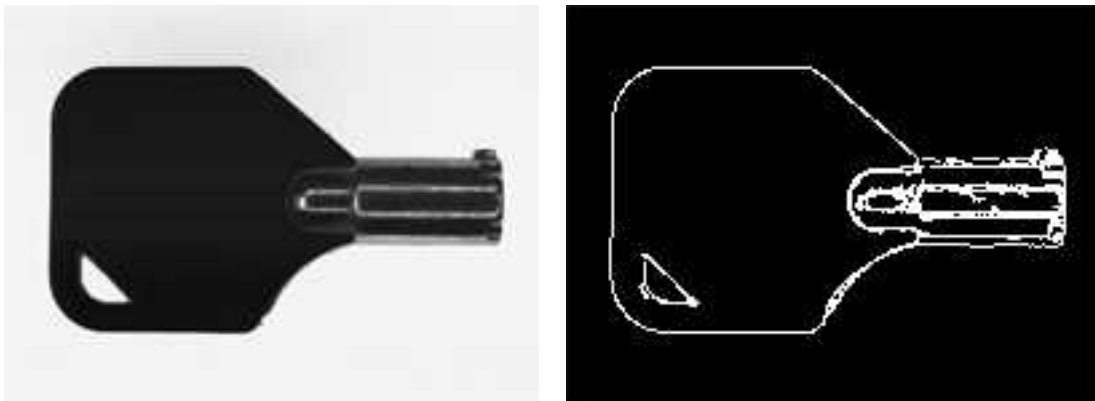
Source image (left) and destination image after Canny edge transformation (right)

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Canny Edge Detector`.

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

3. Click the `Open` icon of the Source Image area, and load the image file `EasyImage\Key⊷.tif`.

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

Step 2: Apply the Canny edge transformation

1. Click the `New` icon in the Destination Image area to create a new destination image.

2. Keep the default settings, and click `OK`.

3. In the canny edge detector dialog box, click `Apply` to perform the operation in the destination image.

Detecting the Corners of an Object Using Harris Corner Detector

Objective

Following this tutorial, you will learn how to use EasyImage to detect the corners of an object. The detection uses the Harris corner detector algorithm.

You'll need to load a source image (step 1), and simply apply the Harris corner detection (step 2).
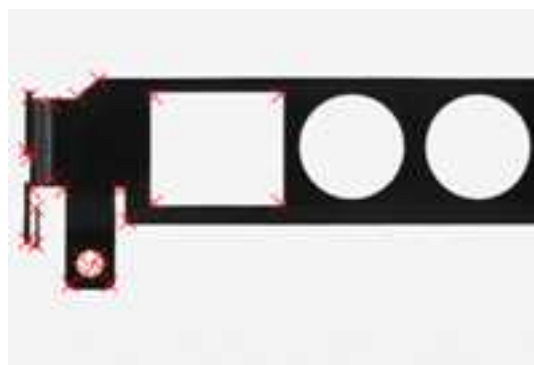
Corners are detected in the source image

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Harris Corner Detector`.

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

3. Click the `Open` icon of the Source Image area, and load the image file `EasyGauge\Bracket_.tif`.

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

Step 2: Apply the Harris corner detection

1. In the Harris corner detector dialog box, enter 2.3 for the Scale property.

2. Click `Apply` to perform corners detection.

3. Click `Results` to display the coordinates of all detected corners.

4. The `Columns` button allows you to display additional properties in the results list.

Detecting a Horizontal or Vertical Line Using Projection

Objective

Following this tutorial, you will learn how to use EasyImage to detect defects (horizontal/vertical line) in a gray-level image.

You'll need first to load a source image (step 1), set a vector (step 2), and then detect the defect (horizontal line) (step 3).
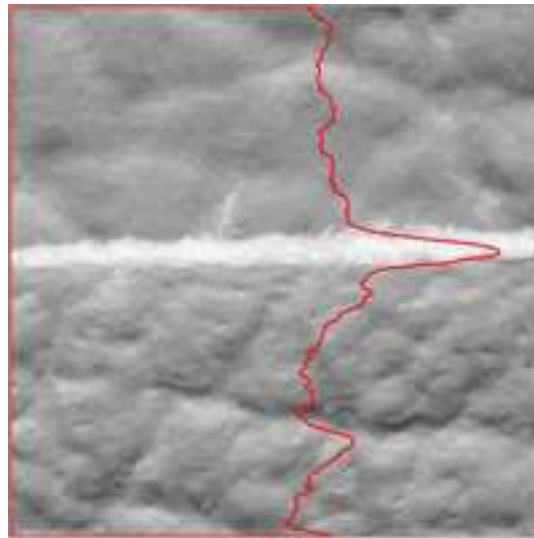
Defects can be detected using the image projection

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Projection`.
2. Click the `Open` icon of the Source Image area, and load the image file `EasyImage\Leather.bmp`.
3. Keep the default variable name for the new image object, and click `OK`.

Step 2: Set the destination vector

1. Click the `New` icon in the Destination Vector area.
2. Select the BW32 option for the vector type, and click `OK`.

Step 3: Detect the defects

1. In the Image Projection dialog box, select the `column` button, and click `Execute` to perform the operation.
2. The resulting vector and the corresponding plot are displayed in the destination vector window. The graphical result also appears on the image. Each vector value is the sum of all pixels values across the corresponding horizontal row (or vertical column). By this mean, horizontal (or vertical) defects are easily detected.

Creating a Flexible Mask

Objective

Following this tutorial, you will learn how to create a flexible mask from a source image, to restrict a future processing to an arbitrary-shaped do-care area.

Flexible masks can be created in any ways to build a bi-level image. Here, we will first load the source image (step 1), and then successively invert it, and threshold it (steps 2-3). The resulting image —the flexible mask— will be saved as a new image file (step 4). This new image file is a bi-level image. However, there are still black areas that need to be erased, before using the image as a flexible mask. You can use a third-party software, such as Paint, to clear the unwanted areas.
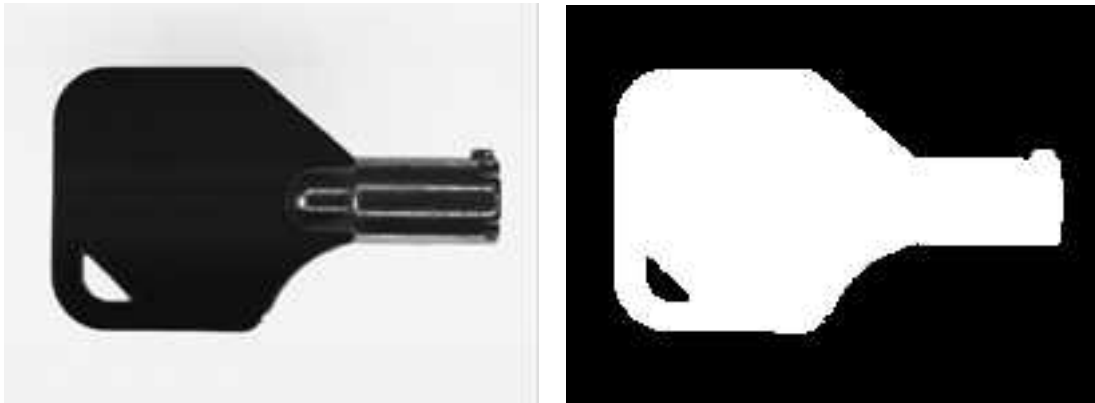
Source image (left) and flexible mask image (right)

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Arithmetic & logic`.
2. Click the `Open` icon of the Source Image 0 area, and load the image file `EasyImage\Key⊳.tif`.
3. Keep the default variable name for the new image object, and click `OK`.

Step 2: Invert the image

1. Click the `New` icon of the `Destination` area.
2. Keep the default settings for the new image object, and click `OK`.
3. In the `Operation` drop-down list, select `Invert`, and click `Execute`

Step 3: Threshold the image

1. From the main menu, click `EasyImage`, then `Threshold`.
2. In the Source Image area, select the inverted image from the drop-down list.
3. Click the `New` icon of the Destination area.
4. Keep the default settings for the new image object, and click `OK`.
5. Select the Absolute option, enter '46' as the threshold value, and click `Execute`.

Step 4: Save the flexible mask

1. Right-click in the destination image, and select `Save As...`.

2. Type a file name for the new flexible mask file. Finally, click Save.

> ✓ **TIP**
> The new image now is a bi-level image. However, there are still black areas
> that need to be erased, before using the image as a flexible mask. You can
> use a third-party software, such as Paint, to clear the unwanted areas.

Computing Gray-Level Statistics Using a Flexible Mask

"Using Flexible Masks" on page 122

Objective

Following this tutorial, you will learn how to compute gray-level statistics on an arbitrary-shaped area only.

You'll need first to load a source image (step 1), and a flexible mask image (step 2). The mask image must be applied on the source image (step 3), to separate do-care areas (that must be considered) and don't-care areas (that should not be considered). Finally, the gray-level statistics are computed on the do-care area only (step 4).
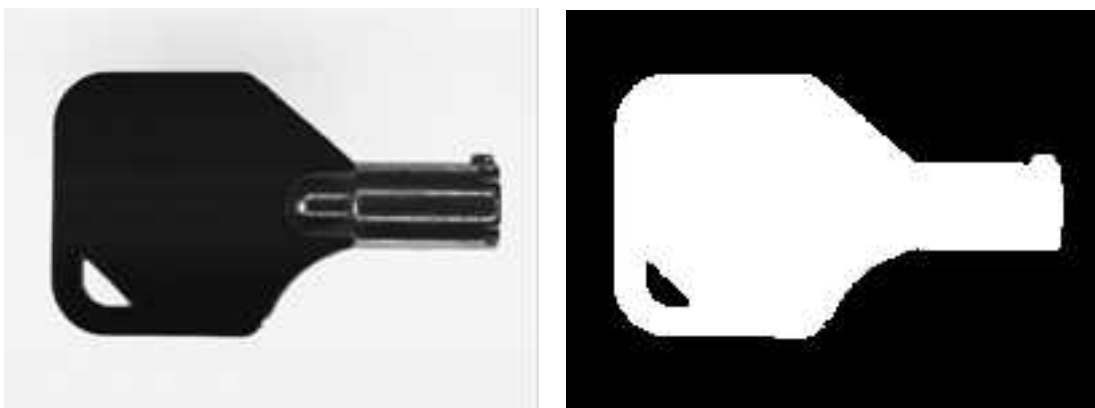
Source image (left) and flexible mask image (right)

Step 1: Load the source image

1. From the main menu, click EasyImage, then Image Statistics, Gray Scale.

2. Click the Open icon of the Source Image area, and load the image file EasyImage\Key.tif.

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

Step 2: Load the flexible mask image

1. From the main menu, click Image, then Open....

2. Load the image file EasyImage\Mask2.bmp.

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

Step 3: Apply the flexible mask on the source image

1. In the Mask area of the `Gray Scale Image Pixels Statistics` dialog box, select the mask image from the drop-down list.

2. The source image preview in the dialog box shows (in red diagonal lines) the don't-care area, that is the area that will be not be considered when computing the gray-level statistics.

Step 4: Compute the gray-level statistics

1. Select the `Pixel Count` check-box.

2. Click `Execute`.

The results are displayed in the read-only fields below.

Detecting the Corners of an Object Using Hit-and-Miss Transform

"Hit-and-Miss Transform" on page 117

Objective

Following this tutorial, you will learn how to use EasyImage to detect top corners in an image, using the hit-and-miss transform.

You'll need to load a source image (step 1), set the kernel that represents a top corner (step 2), and then set a destination image and simply execute the hit-and-miss transform (step 3).

Source image (left) and top corner detected in the source image (white dot)

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Hit And Miss`.

2. Click the `Open` icon of the Source Image area, and load the image file `EasyImage\Diamond.bmp`.

3. Keep the default variable name, and click `OK`.

Step 2: Set the hit-and-miss kernel

*   In the `Hit And Miss` dialog box, set the kernel according to the following values:

Kernel that detects top corners

Step 3: Apply the hit-and-miss transform

1. Click the `New` icon of the `Destination Image` area.

2. Keep the default parameters and variable name, and click `OK`.

3. Click `Execute` to perform the operation.

The top corner (white dot) is detected.

4. Try with other kernel configurations to detect the other corners.

Extracting a Vector Using Profile Function

"Profile Sampling" on page 118

Objective

Following this tutorial, you will learn how to use EasyImage to detect scratches.

You'll need first to load an image (step 1), set a destination vector, and detect the scratches (step 2).
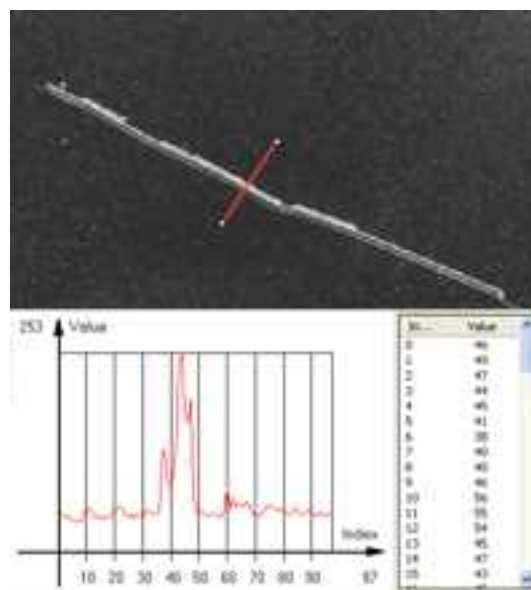
Scratches can be detected using a profile

Step 1: Load the source image

From the main menu, click `EasyImage`, then `Profile`.

1. Click the `Open` icon of the `Source Image` area, and load the image file `EasyImage\Plastic.tif`.

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

Step 2: Set the destination vector and detecting the scratches

1. Click the `New` icon in the Destination Vector area.

2. Select the BW8 option for the vector type, and click `OK`.

3. A profile appears on the image (red line segment). In the destination vector window, vector values correspond to pixels along the line segment.

The scratch is detected as a sharp deviation in the vector graph.

4. Using the mouse, drag the handles to move or resize the red line segment, and observe the plot evolution.

The sharp deviation appears whenever the line segment is placed across the scratch.

Enhancing an X-ray image

Objective

Following this tutorial, you will learn how to use EasyImage to enhance an X-ray image.

You'll need first to load an image (step 1), then define convolution parameters to enhance the image (step 2).
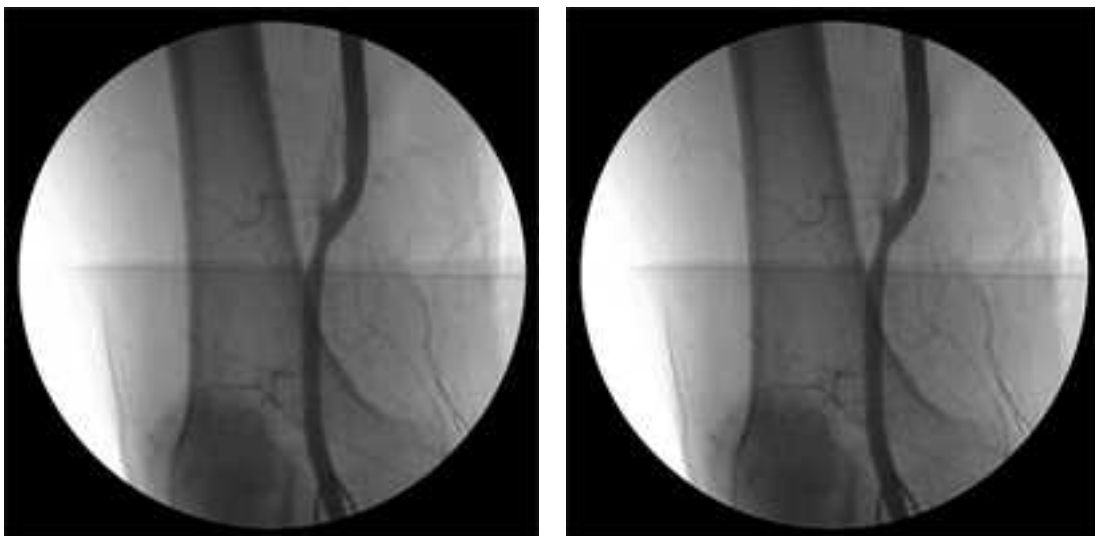
Source image (left) gnd enhanced image, after predefined and user-defined convolutions (right)

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Convolution`.

2. Click the `Open` icon of the `Source Image` area, and load the image file `EasyImage\XRay.bmp`.

3. Keep the default variable name for the new image object, and click `OK`.

4. Click the `New` icon of the `Destination Image` area to create a new destination image.

5. Keep the default variable name and click `OK`.

Step 2: Set the convolution parameters

1. From the `Predefined kernels` drop-down list, select `Highpass2`, and click `Execute` to perform the operation.

The image is no longer blurred but the result is bad because the filter has revealed the noise of the source image. We need to create a new convolution kernel that will apply a softer high-pass filtering.

2. Click the `New` icon next to the `User defined kernels` drop-down list.

3. Keep the default dimension (3x3) and variable name, and click `OK`.

4. Enter the following kernel data from left to right and top to bottom: -1, -1, -1; -1, 15, -1; -1, -1, -1, and click `Apply`.

5. Click `Execute` in the Convolution dialog box to perform the operation. The image is much clearer now.

Correcting Non-Uniform Illumination

Objective

Following this tutorial, you will learn how to use EasyImage to correct non-uniform illumination in an image.

You'll need first to load an image (step 1), load a light reference image (step 2), and perform the correction (step 3).
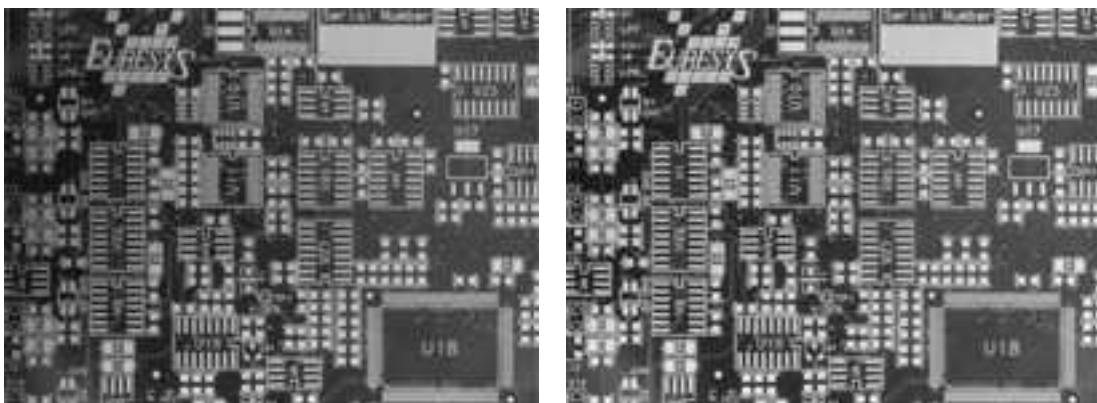
Source image, with non-uniform illumination (left) and corrected image (right)

Step 1: Load the source image

1. From the main menu, click `EasyImage`, then `Uniformize`.

2. Click the `Open` icon of the Source Image area, and load the image file `EasyImage\Board (original).tif`.

3. Keep the default variable name for the new image object, and click `OK`.

Step 2: Load the reference image

1. Click the `Open` icon of the `Light Reference` area, and load the image file `EasyImage\Board (light reference).tif`.

To obtain the light reference image, we used a white screen illuminated in the same condition as the board (original image).

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

Step 3: Perform the correction

1. Click the `New` icon in the `Destination Image` area to create a new destination image.

2. Keep the default values and click `OK`.

3. Click `Execute` to perform the operation.

4. In both source and destination images, right-click and select `3D Rendering`.

5. In the new 3D windows, click and drag the mouse to rotate the view. Compare the profiles.

Correcting Shear Effect

Objective

Following this tutorial, you will learn how to use EasyImage to correct a shear effect in an image. The following image is taken by a line-scan camera. The camera sensor was misaligned, resulting in a so-called shear effect.

You'll need first to load an image (step 1), create a destination image (step 2), and then set pivots parameters to perform the correction (step 3).
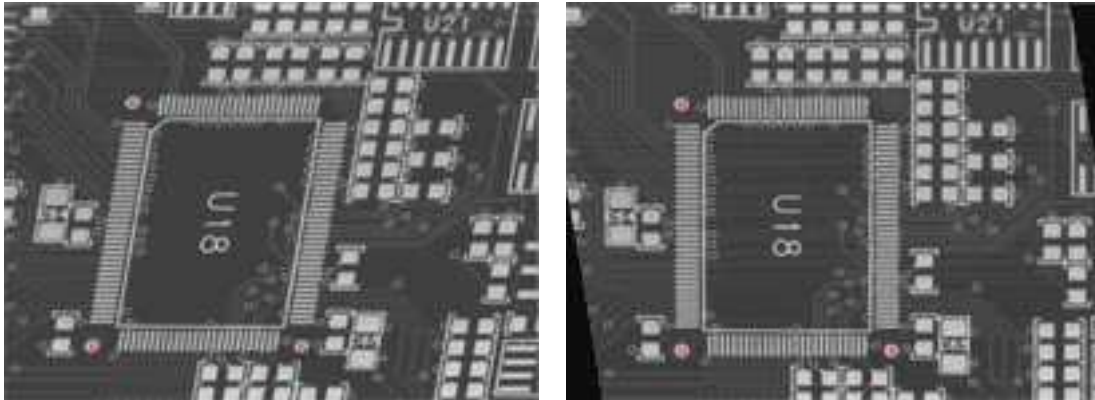
Source image, with a shear effect (left) and corrected image (right)

Step 1: Load the source image

1. From the main menu, click EasyImage, then Register.

2. Click the Open icon of the Source Image area, and load the image file EasyImage\Shear.tif.

3. Keep the default variable name for the new image object, and click OK. Three pivots points appear in the image.

Step 2: Create a destination image

1. Click the New icon of the Destination Image area.

2. Enter '768' and '576' as image width and height, and click OK to accept the default name. Three pivots points appear in the image.

Step 3: Set the pivots parameters

1. In the source image, using the mouse, drag each pivot to the center of the fiducial marks (the dots around the U18 area).

Notice that the source pivots coordinates, in the Register dialog box, have changed accordingly.

2. To correct the image, enter the following destination pivots coordinates:
   - X0: 170
   - Y0: 495
   - X1: 470
   - Y1: 495
   - X2: 170
   - Y2: 144

3. Click Execute to perform the operation.

Correcting Skew Effect

Objective

Following this tutorial, you will learn how to use EasyImage to correct skew effect in an image.

You'll need first to load an image (step 1), create a destination image (step 2), and then set a correction angle to perform the correction (step 3).
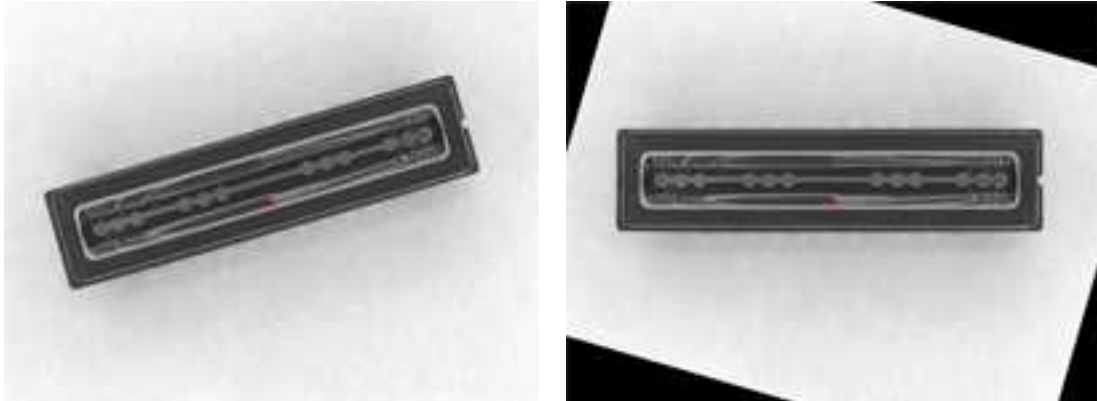
Source image, with a skew effect (left) and corrected image (right)

Step 1: Loading the source image

1. From the main menu, click `EasyImage`, then `Scale and Rotate`.
2. Click the `Open` icon of the `Source Image` area, and load the image file `EasyImage\CCD.tif`.
3. Keep the default variable name for the new image object, and click `OK`.

Step 2: Creating a destination image

1. Click the `New` icon of the `Destination Image` area.
2. Enter '768' and '576' as image width and height, and click `OK` to accept the default name.

Step 3: Setting the correction angle

1. Select the `Rotate` option, and enter -16.17 in the Angle (Deg) field. (To measure this rotation angle, refer to Measuring the rotation angle of an object.)
2. From the Interpolation bits drop-down list, select 8 bits to get a better result.
3. Click `Execute` to perform the operation.

5.2. EasyColor

Performing Thresholding on Color Images

"Color Components" on page 124

Objective

Following this tutorial, you will learn how to use EasyColor to segment a color source image, by setting a threshold value for each color component of the current color system. For example, to retrieve the solder pads on a PCB, you'll perform a color segmentation based on the golden pixels (H), with a loose discrimination on the brightness (L) and saturation (S), to eliminate surface and lighting effects.

You'll need first to load a color source image, create a destination image, and a color lookup table (steps 1-3). Then, you'll set the color system and tune each component tolerance to get a satisfying segmentation of the solder pads (step 4).
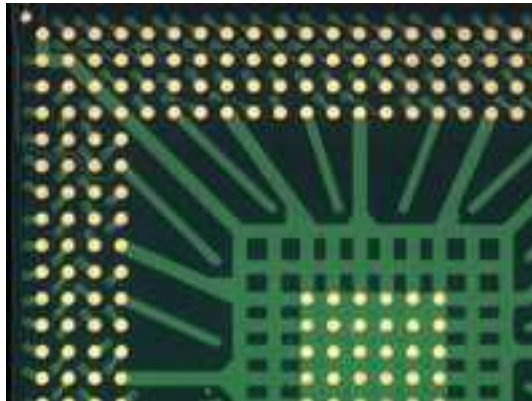
Source image

Thresholded image

Step 1: Load the source image

1. From the main menu, click `EasyColor`, then `Threshold`.

2. Click the `Open` icon of the `Source Image` area, and load the image file `EasyColor\BGA Substrate Color.jpg`.

3. Keep the default variable name for the new Image object, and click `OK`.

4. Disable the `Preview Mode` check-box to see the raw source image.

Step 2: Create a destination image

1. Click the New icon of the Destination Image area.

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

Step 3: Create a color lookup table

1. Click the New icon of the Color Lookup area.

2. Keep the default variable name for the new color lookup object, and click OK.

Step 4: Perform the color segmentation

1. Select LSH from the Color System drop-down list.

2. In the source image, click in a golden pad. The pixel lightness, saturation and hue values are updated in the Color Threshold dialog box.

3. Adjust the tolerance of lightness and saturation to enlarge the range of thresholded pixels, until you get a satisfying segmentation in the destination image.

Performing Color Segmentation

"Color Components" on page 124

Objective

Following this tutorial, you will learn how to use EasyImage to perform color segmentation.

You'll need first to load an image (step 1), create a color look-up table (step 2), and perform the segmentation (step 3).

Source image

Segmented image

Step 1: Load the source image

1. From the main menu, click EasyColor, then Threshold.

2. Click the Open icon of the Source Image area, and load the image file EasyColor\Pills.tif.

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

4. Disable the `Preview Mode` check-box to see the raw source image.

Step 2: Create a color lookup table

1. Click the `New` icon of the Color Lookup area.

2. Keep the default variable name for the new color lookup object, and click `OK`.

Step 3: Perform the color segmentation

1. Select `LSH` from the `Color System` drop-down list.

2. In the source image, click in the center of a green pill. The pixel lightness, saturation and hue values are updated in the `Color Threshold` dialog box.

3. Increase the lightness tolerance up to 120. Increase the saturation tolerance up to 50.

4. Enable the `Preview Mode` check-box to see the result of the segmentation. If needed, click in the green pills to improve the result.

5. Click the `New` icon of the Destination Image area.

6. Keep the default settings for the new Image object, and click `OK`.

7. The new image is automatically thresholded. Clicking `Execute` will insert the corresponding code into the script windows.

6. Code Snippets

6.1. Basic Types

Loading and Saving Images

Functional Guide | Reference: Load, Save, SaveJpeg

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

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

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

// ...

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

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

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

Interfacing Third-Party Images

Functional Guide | Reference: SetImagePtr

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

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

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

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

// ...

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

Retrieving Pixel Values

Functional Guide | Reference: GetImagePtr

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

using System.RunTime.InteropServices;

IntPtr pixAddr;
byte pix;

//...

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

ROI Placement

Functional Guide | Reference: Attach, SetPlacement

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

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

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

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

//Set the ROI position
myROI.SetPlacement(50, 50, 200, ↲00);
```

Vector Management

Functional Guide | Reference: Empty, AddElement

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

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

// Clear the vector
```

```
ramp.Empty();

// Fill the vector with increasing values
for(int i= 0; i < 28; i++)
{
    bw8.Value = (byte)i;
    ramp.AddElement(bw8);
}

// Retrieve the 0th element value
EBW8 value = ramp.GetElement(9);
```

Exception Management

Functional Guide | Reference: GetPixel, What

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

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

    // ...

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

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

6.2. EasyImage

Thresholding

Single Thresholding

Functional Guide | Reference: SetSize, Threshold

```
/////////////////////////////////////////////////////////
// This code snippet shows how to perform minimum residue   //
// thresholding, absolute thresholding and relative         //
// thresholding operations.                                 //
/////////////////////////////////////////////////////////

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

// ...

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Minimum residue thresholding (default method)
EasyImage.Threshold(srcImage, dstImage);

// Absolute thresholding (threshold = ⤶0)
EasyImage.Threshold(srcImage, dstImage, ⤶0);

// Relative thresholding (70% black, 30% white)
EasyImage.Threshold(srcImage, dstImage, unchecked((uint)EThresholdMode.Relative), 0, 255, 0.7f);
```

Double Thresholding

Functional Guide | Reference: DoubleThreshold

```
/////////////////////////////////////////////////////////
// This code snippet shows how to perform a thresholding    //
// operation based on low and high threshold values.        //
/////////////////////////////////////////////////////////

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

// ...

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Double thresholding, low threshold = 50, high threshold = ⤶50,
// pixels below 50 become black, pixels above ⤶50 become white,
// pixels between thresholds become gray
EasyImage.DoubleThreshold(srcImage, dstImage, 50, ⤶50, 0, ⤶28, 255);
```

Histogram-Based Single Thresholding

Functional Guide | Reference: Histogram, HistogramThreshold

```
//////////////////////////////////////////////////////
// This code snippet shows how to perform a minimum residue    //
// thresholding operation based on an histogram.               //
//////////////////////////////////////////////////////

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

// Histogram constructor
EBWHistogramVector histo= new EBWHistogramVector();

// Variables
uint thresholdValue= , unchecked((uint)EThresholdMode.MinResidue);
float avgBelowThr, avgAboveThr;

// ...

// Compute the histogram
EasyImage.Histogram(srcImage, histo);

// Compute the single threshold (and the average pixel values below and above the threshold)
EasyImage.HistogramThreshold(histo, ref thresholdValue, out avgBelowThr, out avgAboveThr);

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform the single thresholding
EasyImage.Threshold(srcImage, dstImage, thresholdValue);
```

Histogram-Based Double Thresholding

Functional Guide | Reference: Histogram, ThreeLevelsMinResidueThreshold, DoubleThreshold

```
//////////////////////////////////////////////////////////////
// This code snippet shows how to perform a double thresholding    //
// operation. The low and high threshold values are computed       //
// according to the minimum residue method based on an histogram.  //
//////////////////////////////////////////////////////////////

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

// Histogram constructor
EBWHistogramVector histo= new EBWHistogramVector();

// Variables
EBW8 lowThr= new EBW8();
EBW8 highThr= new EBW8();
float avgBelowThr, avgBetweenThr, avgAboveThr;

// ...

// Compute the histogram
EasyImage.Histogram(srcImage, histo);
```

```
// Compute the low and high threshold values automatically
// (and the average pixel values below, between and above the threshold)
EasyImage.ThreeLevelsMinResidueThreshold(histo, out lowThr, out highThr, out avgBelowThr, out avgBetweenThr, out
avgAboveThr);

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform the double thresholding
EasyImage.DoubleThreshold(srcImage, dstImage, lowThr.Value , highThr.Value);
```

Arithmetic and Logic Operations

Functional Guide | Reference: Oper

```
////////////////////////////////////////////////////
// This code snippet shows how to apply miscellaneous //
// arithmetic and logic operations to images.         //
////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcGray0= new EImageBW8();
EImageBW8 srcGray�= new EImageBW8();
EImageBW8 dstGray= new EImageBW8();
EImageC24 srcColor= new EImageC24();
EImageC24 dstColor= new EImageC24();

EBW8 bw8Constant = new EBW8(2);
EC24 c24Constant = new EC24(▸28, 64, ▸96);

// ...

// All images must have the same size
dstGray.SetSize(srcGray0);
dstColor.SetSize(srcColor);

// Subtract srcGray▸ from srcGray0
EasyImage.Oper(EArithmeticLogicOperation.Subtract, srcGray0, srcGray▸, dstGray);

// Multiply srcGray0 by a constant value
EasyImage.Oper(EArithmeticLogicOperation.Multiply, srcGray0, bw8Constant, dstGray);

// Add a constant value to srcColor
EasyImage.Oper(EArithmeticLogicOperation.Add, srcColor, c24Constant, dstColor);

// Erase (blacken) the destination image where the source image is black
bw8Constant.Value = (byte)0;
EasyImage.Oper(EArithmeticLogicOperation.SetZero, srcGray0, bw8Constant, dstGray);
```

Convolution

Pre-Defined Kernel Filtering

```
////////////////////////////////////////////////////////
// This code snippet shows how to apply miscellaneous     //
// convolution operations based on pre-defined kernels.  //
////////////////////////////////////////////////////////
```

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

// ...

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform a Uniform filtering (5x5 kernel)
EasyImage.ConvolUniform(srcImage, dstImage, 2);

// Perform a Highpass filtering
EasyImage.ConvolHighpass(srcImage, dstImage);

// Perform a Gradient filtering
EasyImage.ConvolGradient(srcImage, dstImage);

// Perform a Sobel filtering
EasyImage.ConvolSobel(srcImage, dstImage);
```

User-Defined Kernel Filtering

```
/////////////////////////////////////////////////////
// This code snippet shows how to apply a convolution  //
// operation based on a user-defined kernel.           //
/////////////////////////////////////////////////////

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

// ...

// Create and define a user-defined kernel
// (Frei-Chen row gradient, positive only)
EKernel kernel= new EKernel();
kernel.SetKernelData(0.2929f, 0, -0.2929f,
                     0.4142f, 0, -0.4142f,
                     0.2929f, 0, -0.2929f);

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Apply the convolution kernel
EasyImage.ConvolKernel(srcImage, dstImage, kernel);
```

Non-Linear Filtering

Functional Guide | Reference

Morphological Filtering

Functional Guide | Reference: ErodeBox, DilateBox, OpenDisk

```
/////////////////////////////////////////////////////
// This code snippet shows how to apply miscellaneous  //
```

116

```
// morphological filtering operations.                    //
/////////////////////////////////////////////////////////

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

// ...

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform an erosion (3x3 square kernel)
EasyImage.ErodeBox(srcImage, dstImage, ↵);

// Perform a dilation (5x3 rectangular kernel)
EasyImage.DilateBox(srcImage, dstImage, 2, ↵);

// Perform an Open operation (5x5 circular kernel)
EasyImage.OpenDisk(srcImage, dstImage, 2);
```

Hit-and-Miss Transform

Functional Guide | Reference: SetValue, HitAndMiss

```
/////////////////////////////////////////////////////////
// This code snippet shows how to highlight the left corner //
// of a rhombus by means of a Hit-and-Miss operation.        //
/////////////////////////////////////////////////////////

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

// ...

// Create and define a Hit-and-Miss kernel
// corresponding to the left corner of a rhombus
EHitAndMissKernel leftCorner= new EHitAndMissKernel(-↵, -↵, ↵, ↵);

// Left column of the kernel
leftCorner.SetValue(-↵, 0, EHitAndMissValue.Background);

// Middle column of the kernel
leftCorner.SetValue(0, -↵, EHitAndMissValue.Background);
leftCorner.SetValue(0, 0, EHitAndMissValue.Foreground);
leftCorner.SetValue(0, ↵, EHitAndMissValue.Background);

// Right column of the kernel
leftCorner.SetValue(↵, -↵, EHitAndMissValue.Foreground);
leftCorner.SetValue(↵, 0, EHitAndMissValue.Foreground);
leftCorner.SetValue(↵, ↵, EHitAndMissValue.Foreground);

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Apply the Hit-and-Miss kernel
EasyImage.HitAndMiss(srcImage, dstImage, leftCorner);
```

Vector Operations

Functional Guide | Reference

Path Sampling

Functional Guide | Reference: Empty, AddElement, ImageToPath

```
//////////////////////////////////////////////////////
// This code snippet shows how to retrieve and store the   //
// pixel values along a given path together with the       //
// corresponding pixel coordinates.                        //
//////////////////////////////////////////////////////

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

// ...

// Vector constructor
EBW8PathVector path= new EBW8PathVector();
EBW8 bw8= new EBW8(128);

// Path definition
path.Empty();
for (int i = 0; i < 100; i++)
{
  EBW8Path p;
  p.X = (short)i;
  p.Y = (short)i;
  p.Pixel = bw8;
  path.AddElement(p);
}

// Get the image data along the path
EasyImage.ImageToPath(srcImage, path);
int pixel = path.GetElement(20).Pixel.UINT32Value;
```

Profile Sampling

Functional Guide | Reference: ImageToLineSegment, LineSegmentToImage

```
//////////////////////////////////////////////////////
// This code snippet shows how to set, retrieve and store   //
// the pixel values along a given line segment.             //
//////////////////////////////////////////////////////

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

// ...

// Vector constructor
EBW8Vector profile= new EBW8Vector();

// Get the image data along segment (10,52)-(500,40)
EasyImage.ImageToLineSegment(srcImage, profile, 10, 52, 500, 40);
```

```
// Set all these points to white (255) in the image
EBW8 white = new EBW8(255);
EasyImage.LineSegmentToImage(srcImage, white, ↵0, 5↵2, 500, 40);
```

Statistics

Image Statistics

```
//////////////////////////////////////////////////////////////
// This code snippet shows how to compute basic image statistics. //
//////////////////////////////////////////////////////////////

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

// ...

// Count the number of pixels above the threshold (↵28)
int count;
EBW8 threshold = new EBW8(↵28);
EasyImage.Area(srcImage, threshold, out count);

// Compute the pixels' average and standard deviation values
float stdDev, average;
EasyImage.PixelStdDev(srcImage, out stdDev, out average);

// Compute the image gravity center (pixels above threshold)
float x, y;
EasyImage.GravityCenter(srcImage, ↵28, out x, out y);
```

Sliding Windows Statistics

Functional Guide | Reference: LocalAverage, LocalDeviation

```
////////////////////////////////////////////////////////////////
// This code snippet shows how to perform sliding windows statistics. //
////////////////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcImage= new EImageBW8();
EImageBW8 dstImage0= new EImageBW8();
EImageBW8 dstImage↵= new EImageBW8();

// ...

// All images must have the same size
dstImage0.SetSize(srcImage);
dstImage↵.SetSize(srcImage);

// Local average in a ↦x↦ window
EasyImage.LocalAverage(srcImage, dstImage0, 5, 5);

// Local deviation in a ↦x↦ window
EasyImage.LocalDeviation(srcImage, dstImage↵, 5, 5);
```

Histogram-Based Statistics

Functional Guide | Reference: Histogram, AnalyseHistogram

```
/////////////////////////////////////////////////
// This code snippet shows how to compute statistics   //
// based on an histogram.                              //
/////////////////////////////////////////////////

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

// ...

// Histogram constructor
EBWHistogramVector histo= new EBWHistogramVector();

// Compute the histogram
EasyImage.Histogram(srcImage, histo);

// Compute the average gray-level value
float average = EasyImage.AnalyseHistogram(histo, EHistogramFeature.AveragePixelValue, 0, 255);

// Compute the gray-level standard deviation
float deviation = EasyImage.AnalyseHistogram(histo, EHistogramFeature.PixelValueStdDev, 0, 255);
```

Noise Reduction by Integration

Functional Guide | Reference

Temporal Noise Reduction

Functional Guide | Reference: Oper

```
///////////////////////////////////////////////
// This code snippet shows how to perform noise  //
// reduction by temporal averaging.              //
///////////////////////////////////////////////

// Images constructor
EImageBW16 noisyImage= new EImageBW16();
EImageBW16 cleanImage= new EImageBW16();

// 16 bits work image used as an accumulator
EImageBW16 store= new EImageBW16();

// ...

// All images must have the same size
cleanImage.SetSize(noisyImage);
store.SetSize(noisyImage);

// Clear the accumulator image
EBW16 bw16= new EBW16(0);
EasyImage.Oper(EArithmeticLogicOperation.Copy, bw16, store);

// Accumulation loop
int n;
```

```
for (n = 0; n < ◣0; n++)
{
    // Acquire a new image into noisyImage
    // ...

    // Add this new noisy image into the accumulator
    EasyImage.Oper(EArithmeticLogicOperation.Add, noisyImage, store, store);
}

// Perform noise reduction
bw◣6.Value= (byte)n;
EasyImage.Oper(EArithmeticLogicOperation.Divide, store, bw◣6, cleanImage);
```

Recursive Average

Functional Guide | Reference: Oper, SetRecursiveAverageLUT, RecursiveAverage

```
///////////////////////////////////////////////
// This code snippet shows how to perform noise  //
// reduction by recursive averaging.             //
///////////////////////////////////////////////

// Images constructor
EImageBW8 noisyImage= new EImageBW8();
EImageBW8 cleanImage= new EImageBW8();

// ◣6 bits work image used as an accumulator
EImageBW◣6 store= new EImageBW◣6();

// ...

// All images must have the same size
cleanImage.SetSize(noisyImage);
store.SetSize(noisyImage);

// Clear the accumulator image
EBW◣6 bw◣6= new EBW◣6(0);
EasyImage.Oper(EArithmeticLogicOperation.Copy, bw◣6, store);

// Prepare the transfer lookup table (reduction factor = 3)
EBW◣6Vector lut= new EBW◣6Vector();
EasyImage.SetRecursiveAverageLUT(lut, 3.0f);

// Perform the noise reduction
EasyImage.RecursiveAverage(noisyImage, store, cleanImage, lut);
```

Feature Point Detectors

Harris Corner Detector

Functional Guide | Reference: GetPointCount, GetPoint

```
/////////////////////////////////////////////////////////
// This code snippet shows how to retrieve corners' coordinates //
// by means of the Harris corner detector algorithm.          //
/////////////////////////////////////////////////////////

// Image constructor
```

```
EImageBW8 srcImage= new EImageBW8();

// ...

// Harris corner detector
EHarrisCornerDetector harris= new EHarrisCornerDetector();
EHarrisInterestPoints interestPoints= new EHarrisInterestPoints();
harris.IntegrationScale= 2.0f;

// Perform the corner detection
harris.Apply(srcImage, interestPoints);

// Retrieve the number of corners
uint index = interestPoints.PointCount;

// Retrieve the first corner coordinates
EPoint point = interestPoints.GetPoint(0);
float x = point.X;
float y = point.Y;
```

Canny Edge Detector

Functional Guide | Reference: Apply

```
/////////////////////////////////////////////////
// This code snippet shows how to highlight edges  //
// by means of the Canny edge detector algorithm.  //
/////////////////////////////////////////////////

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

// ...

// Canny edge detector
ECannyEdgeDetector canny= new ECannyEdgeDetector();

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform the edges detection
canny.Apply(srcImage, dstImage);
```

Using Flexible Masks

Functional Guide | Reference

Computing Pixels Average

Functional Guide | Reference: PixelAverage

```
///////////////////////////////////////////////////
// This code snippet shows how to compute statistics    //
// inside a region defined by a flexible mask.         //
///////////////////////////////////////////////////

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

```
EImageBW8 mask= new EImageBW8();

// ...

// Compute the average value of the source image pixels
// corresponding to the mask do-care areas only
float average;
EasyImage.PixelAverage(srcImage, mask, out average);
```

```
EImageBW8 mask= new EImageBW8();

// ...
```

6.3. EasyColor

Colorimetric Systems Conversion

Functional Guide | Reference: ConvertFromRgb, Transform

```
////////////////////////////////////////////////////
// This code snippet shows how to convert a color image //
// from the RGB to the Lab colorimetric system.        //
////////////////////////////////////////////////////

// Images constructor
EImageC24 srcImage= new EImageC24();
EImageC24 dstImage= new EImageC24();

// ...

// Prepare a lookup table for
// the RGB to La*b* conversion
EColorLookup lookup= new EColorLookup();
lookup.ConvertFromRgb(EColorSystem.Lab);

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform the color conversion
EasyColor.Transform(srcImage, dstImage, lookup);
```

Color Components

Functional Guide | Reference: Compose, ConvertFromRgb, GetComponent

```
////////////////////////////////////////////////////
// This code snippet shows how to create a color image  //
// from 3 grayscale images and extract the luminance    //
// component from a color image.                        //
////////////////////////////////////////////////////

// Images constructor
EImageBW8 red= new EImageBW8();
EImageBW8 green= new EImageBW8();
EImageBW8 blue= new EImageBW8();
EImageC24 colorImage= new EImageC24();
EImageBW8 luminance= new EImageBW8();

// ...

// Source and destination images must have the same size
colorImage.SetSize(red);

// Combine the color planes into a color image
EasyColor.Compose(red, green, blue, colorImage);

// Prepare a lookup table for
// the RGB to LSH conversion
EColorLookup lookup= new EColorLookup();
lookup.ConvertFromRgb(EColorSystem.Lsh);

// Source and destination images must have the same size
```

```
luminance.SetSize(colorImage);

// Get the Luminance component
EasyColor.GetComponent(colorImage, luminance, 0, lookup);
```

White Balance

Functional Guide | Reference: PixelAverage, WhiteBalance, Transform

```
/////////////////////////////////////////////////////////
// This code snippet shows how to perform white balancing. //
/////////////////////////////////////////////////////////

// Images constructor
EImageC24 srcImage= new EImageC24();
EImageC24 dstImage= new EImageC24();
EImageC24 whiteRef= new EImageC24();

// ...

// Create a lookup table
EColorLookup lut= new EColorLookup();

// Measure the calibration values from a white reference image
float r, g, b;
EasyImage.PixelAverage(whiteRef, out r, out g, out b);

// Prepare the lookup table for
// a white balance operation
lut.WhiteBalance(1.00f, EasyColor.CompensateNtscGamma, r, g, b);

// Source and destination images must have the same size
dstImage.SetSize(srcImage);

// Perform the white balance operation
lut.Transform(srcImage, dstImage);
```

Pseudo-Coloring

Functional Guide | Reference: SetShading, PseudoColor

```
/////////////////////////////////////////////////////////
// This code snippet shows how to perform pseudo-coloring. //
/////////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcImage= new EImageBW8();
EImageC24 dstImage= new EImageC24();

// ...

// Create a pseudo-color lookup table
EPseudoColorLookup pcLut= new EPseudoColorLookup();

// Define a shade of pure tints, from red to blue
EC24 red= new EC24(255, 0, 0);
EC24 blue= new EC24(0, 0, 255);
pcLut.SetShading(red, blue, EColorSystem.Ish);

// Source and destination images must have the same size
```

```
dstImage.SetSize(srcImage);

// Generate the pseudo-colored image
EasyColor.PseudoColor(srcImage, dstImage, pcLut);
```

Bayer Pattern Decoding

Functional Guide | Reference: BayerToC24

```
/////////////////////////////////////////////////////////////////
// This code snippet shows how to perform Bayer pattern decoding. //
/////////////////////////////////////////////////////////////////

// Images constructor
EImageBW8 bayerImage= new EImageBW8();
EImageC24 dstImage= new EImageC24();

// ...

// Source and destination images must have the same size
dstImage.SetSize(bayerImage);

// Convert to true color with simple interpolation, default parity assumed
EasyColor.BayerToC24(bayerImage, dstImage);
```