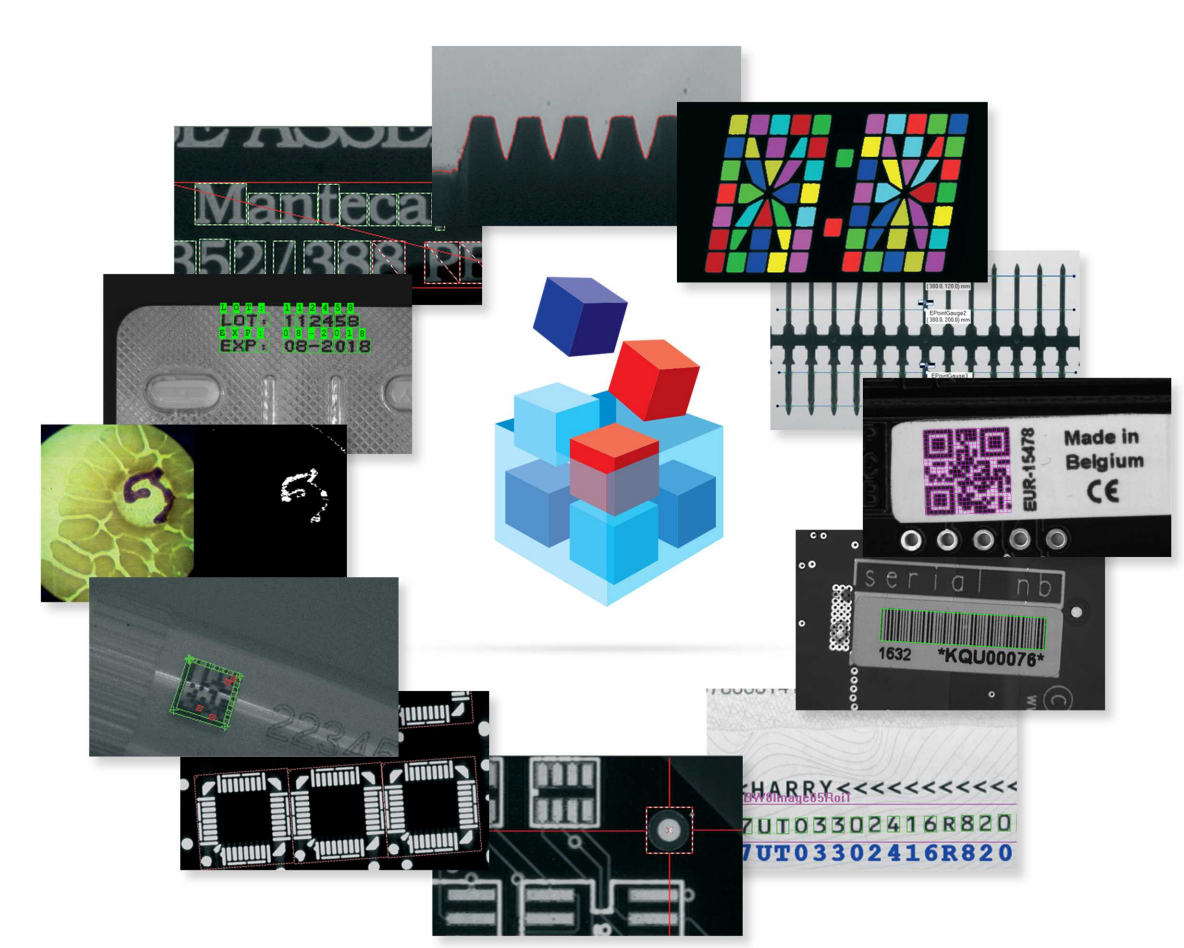
EURESYS™
Excellence in vision

USER GUIDE

Open eVision

3D Tools



Terms of Use

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

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

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

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

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

This documentation is provided with Open eVision 2.6.1 (doc build 1110).

Contents

1. Dealing with Pixel Containers and Files

1.1. Pixel Container Definition

Images

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

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

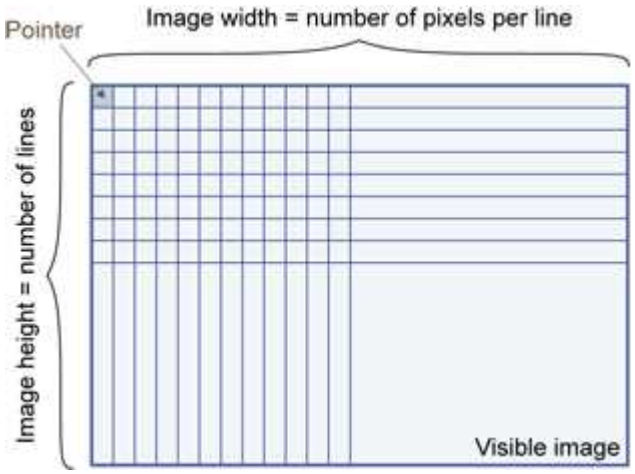


Image main parameters

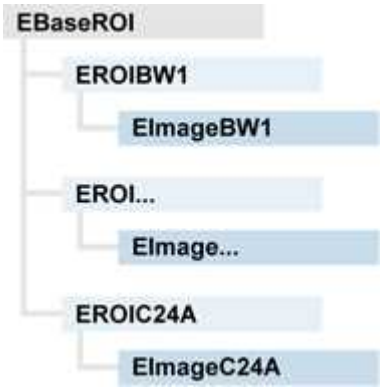
An Open eVision image object has a rectangular array of pixels characterized by `EBaseROI` parameters .

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

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

Classes

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



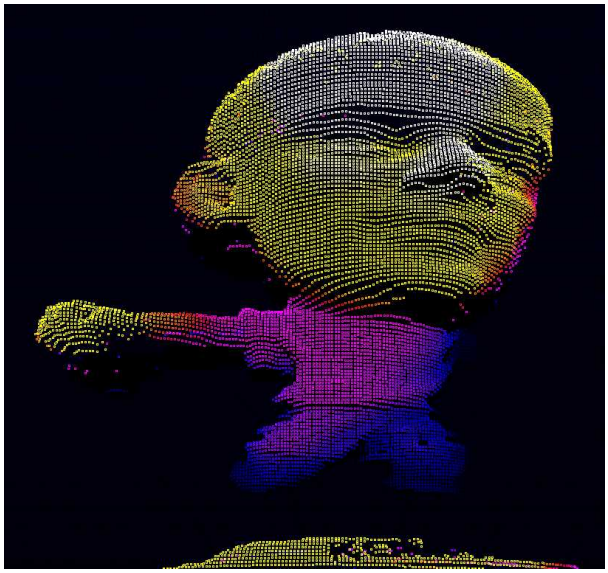
Depth maps

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

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

Point clouds

A point cloud (https://en.wikipedia.org/wiki/Point_cloud) is an unstructured set of 3D points representing discrete positions on the surface of an object.



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

1.2. Pixel Container Types

Images

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

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

BW1	1-bit black and white images (8 pixels are stored in 1 byte)	EImageBW1
BW8	8-bit grayscale images (each pixel is stored in 1 byte)	EImageBW8
BW16	16-bit grayscale images (each pixel is stored in 2 bytes)	EImageBW16
BW32	32-bit grayscale images (each pixel is stored in 4 bytes)	EImageBW32
C15	15-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images and MultiCam RGB15 format.	EImageC15

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

Depth Maps

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

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

Point Clouds

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

1.3. Supported Image File Types

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

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

1.4. Pixel and File Types Compatibility

Depth map to image conversion

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

Pixel and file types compatibility

Pixel access

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

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

Direct access

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

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

Quick Access to BW8 Pixels

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

Supported structures

- `EBW1`, `EBW8`, `EBW32`
- `EC15` (*), `EC16` (*), `EC24` (*)
- `EC24A`
- `EDepth8`, `EDepth16`, `EDepth32f`,

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

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

Pixel and File Type compatibility during Load or Save operations

Type	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
BW1	Ok	N/A	N/A	Ok	Ok	Ok
BW8	Ok	Ok	Ok	Ok	Ok	Ok
BW16	N/A	N/A	Ok	Ok	Ok (***)	Ok
BW32	N/A	N/A	N/A	N/A	Ok (***)	Ok
C15	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C16	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C24	Ok	Ok	Ok	Ok	Ok (**)	Ok
C24A	Ok	N/A	N/A	Ok	N/A	Ok
Depth8	Ok	Ok	Ok	Ok	Ok	Ok
Depth16	N/A	N/A	Ok	Ok	Ok (***)	Ok
Depth32f	N/A	N/A	N/A	N/A	N/A	Ok

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

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

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

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

1.5. Color Types

EISH: Intensity, Saturation, Hue color system.

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

ELCH: Lightness, Chroma, Hue color system.

ELSH: Lightness, Saturation, Hue color system.

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

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

EVSH: Value, Saturation, Hue color system.

EXYZ: CIE XYZ color system.

EYIQ: CCIR Luma, Inphase, Quadrature color system.

EYSH: CCIR Luma, Saturation, Hue color system.

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

2. Manipulating Pixels Containers and Files

2.1. Pixel Container File Save

Images and Depth Maps

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

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

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

Save throws an exception when:

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

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

image file type arguments

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

(*) Default value.

Assigned image file type if argument is ImageFileType_Auto or missing

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

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

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

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

JPEG compression values

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

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

Representative JPEG 2000 compression quality values

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

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

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

Point Clouds

- Use the `Save` method to save the point cloud in Open eVision proprietary file format.
- Use the `SavePCD` method to save the point cloud in a file compatible with other software such as PCL (Point Cloud Library).

2.2. Pixel Container File Load

Images and Depth Maps

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

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

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

Point Clouds

- Use the `Load` method to save the point cloud in Open eVision proprietary file format.
- Use the `LoadPCD` method to save the point cloud in a file compatible with other software such as PCL (Point Cloud Library).

2.3. Memory Allocation

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

Internal Memory Allocation

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

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

External Memory Allocation

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

1. Declare an image object, for instance `EImageBW8`.
2. Create a suitably sized and aligned buffer (see Image Buffer).
3. Set the image size with the `SetSize` function.
4. Access the buffer with `GetImagePtr`. See also Retrieving Pixel Values.

2.4. Image and Depth Map Buffer

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

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

[1]device-independent bitmap

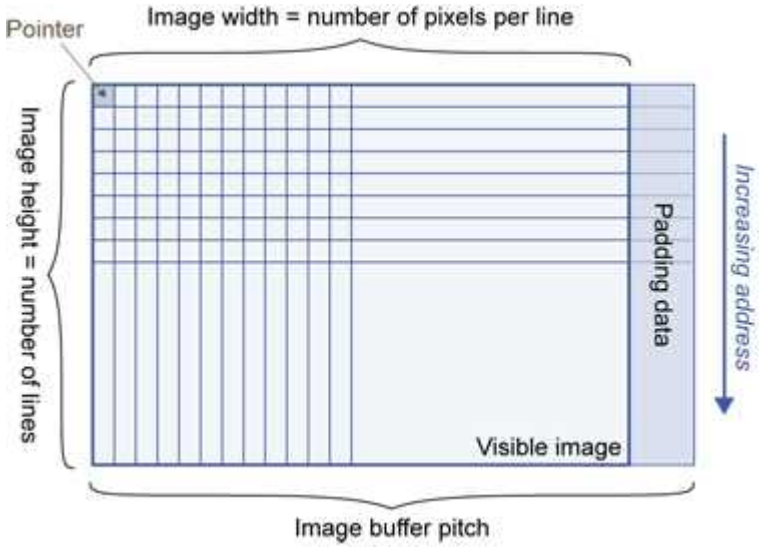


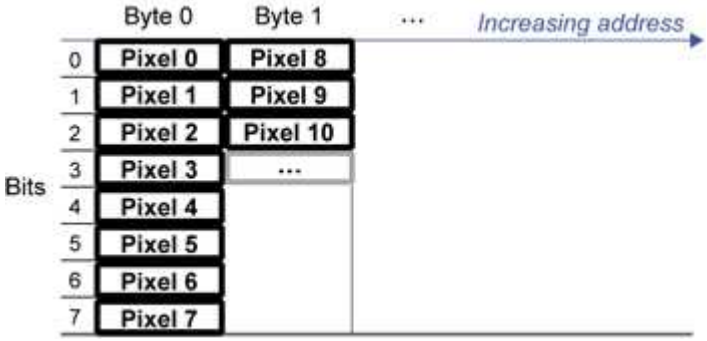
Image Buffer pitch

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

Memory Layout

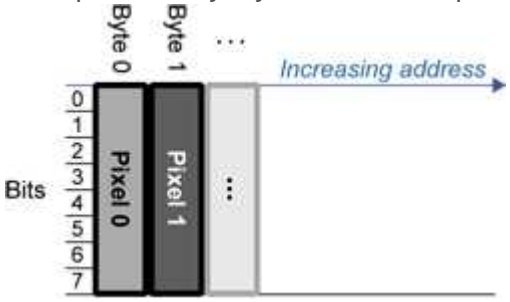
- `EImageBW1` stores 8 pixels in one byte.

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



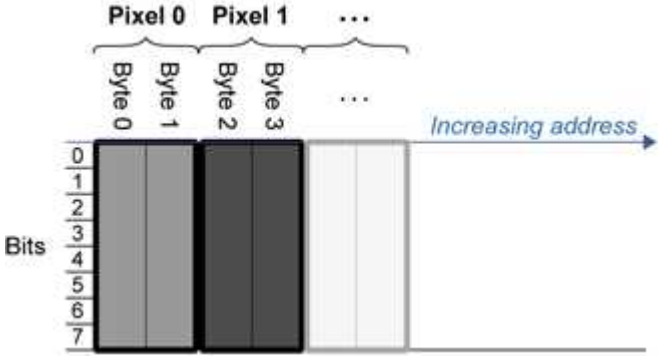
- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

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



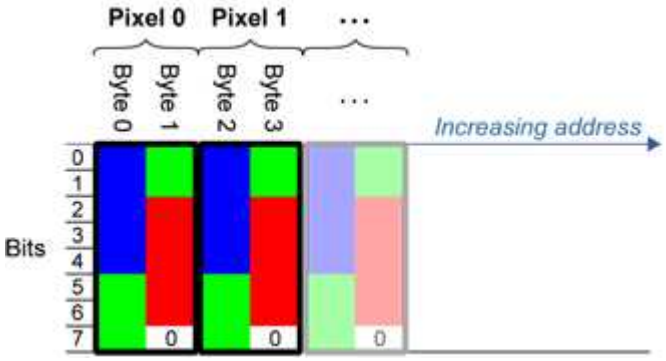
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).

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



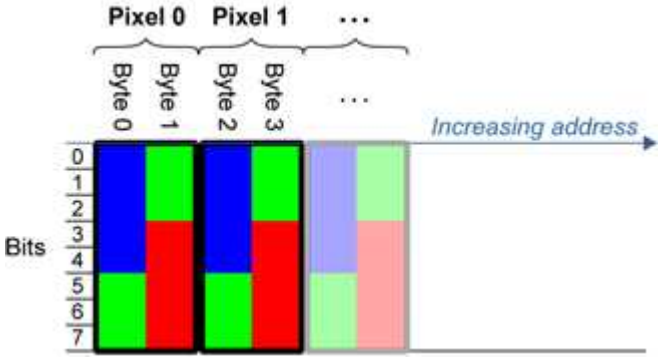
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits.
  The 16th bit is left unused.

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



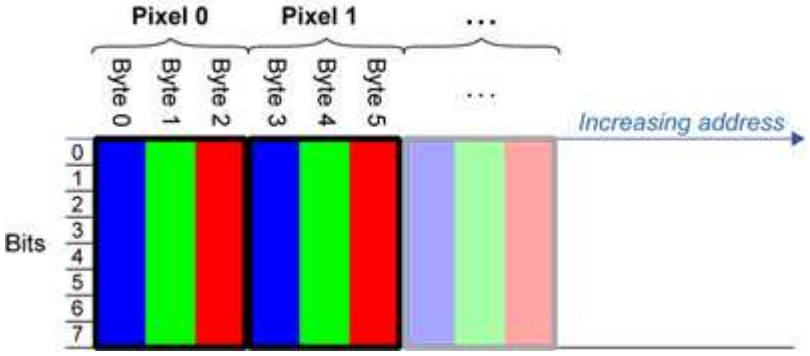
- `EImageC16` stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
  The second color component is coded with 6-bits.

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



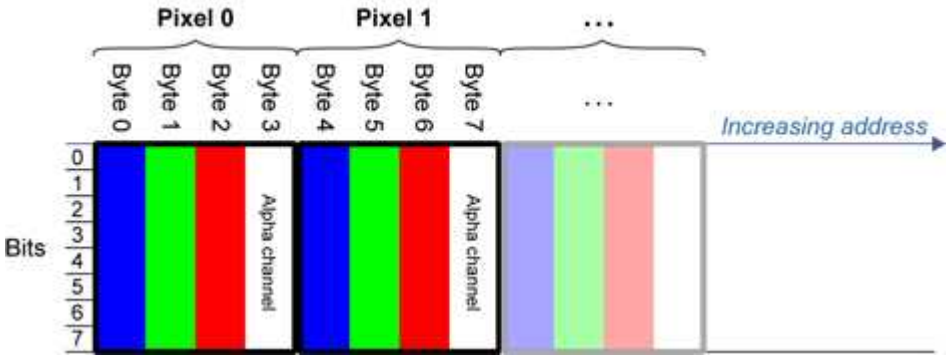
- `EDepthMap16` store each pixel in 2 bytes using a fixed point format.
- `EImageC24` stores each pixel in 3 bytes. Each color component is coded with 8-bits.

Example memory layout of the first pixels of a C24 image buffer:



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

  Example memory layout of the first pixels of a C24A image buffer:



- `EDepthMap32f` store each pixel in 4 bytes using a float format.

2.5. Image Drawing and Overlay

- Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.
- The color palette in 256-color display mode gives optimal rendering. Gray-level images can be improved using LUT[3]s (using histogram stretching techniques or pseudo-coloring).
- The zoom can be different horizontally and vertically.
- `DrawFrameWithCurrentPen` method draws a frame.
- **Non-destructive overlaying** drawing operations do not alter the image contents, such as `MoveTo`/`LineTo`.

---

[1]Graphics Device Interface

[2]Microsoft Foundation Class

[3]LookUp Tables

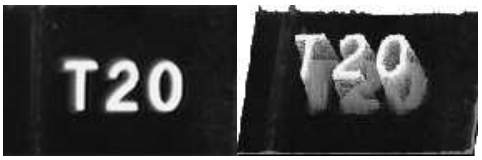
- **Destructive overlaying** drawing operations alter the image contents by drawing inside the image such as `Easy.OpenImageGraphicContext`. Gray-level [color] images can only receive a gray-level [color] overlay.

2.6. 3D Rendering of 2D Images

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

Gray 3D Rendering

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

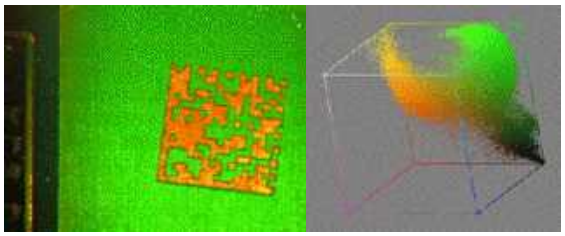


3D rendering

Color Histogram 3D Rendering

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

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

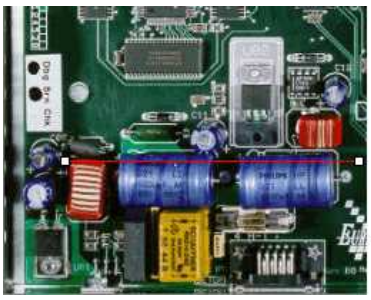


Color histogram rendering

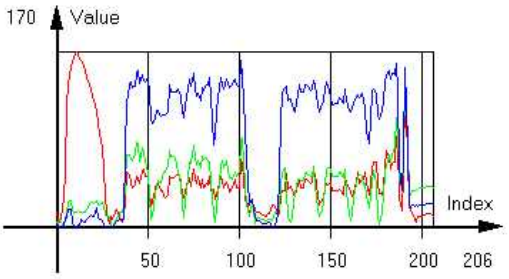
2.7. Vector Types and Main Properties

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

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



Profile in a C24 image



RGB values plot along profile

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

RGB values array (EC24Vector)

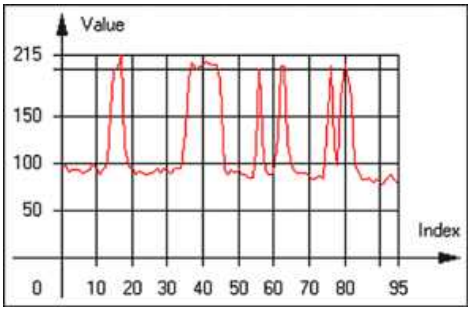
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

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

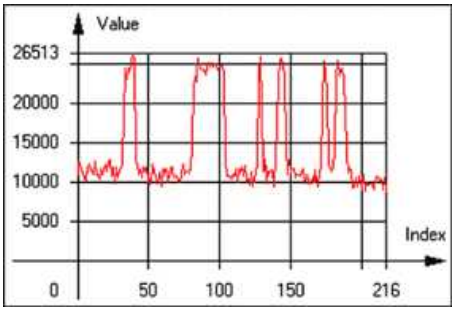
Vector types

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



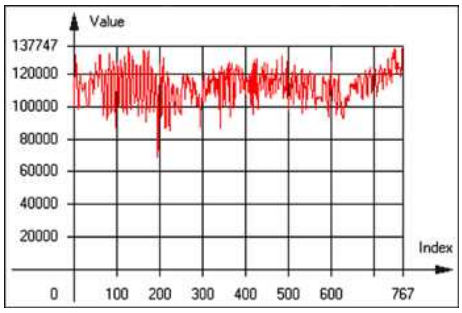
Graphical representation of an EBW8Vector (see Draw method)

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



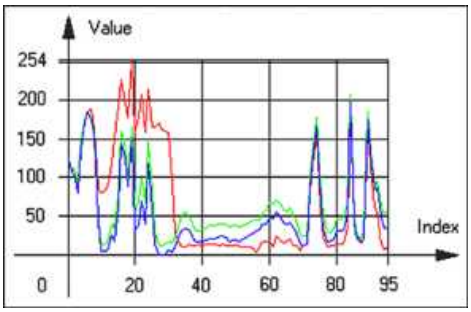
Graphical representation of an `EBW16Vector`

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



Graphical representation of an `EBW32Vector`

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

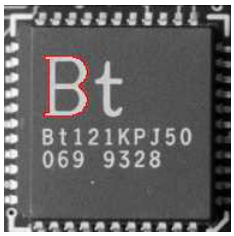


Graphical representation of an `EC24Vector`

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

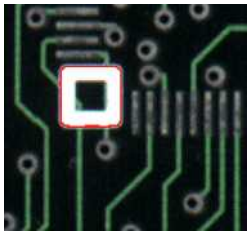
Graphical representation of an `EBW8PathVector` (see `Draw` method)

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



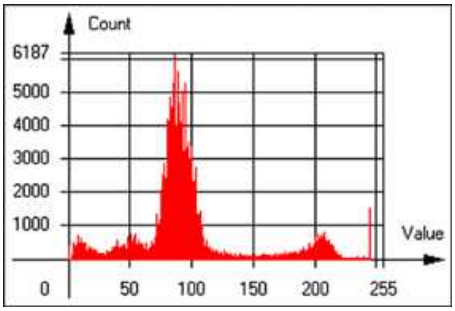
Graphical representation of an `EBW16PathVector` (see `Draw` method)

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



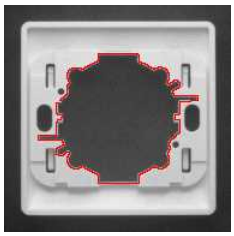
Graphical representation of an `EC24PathVector` (see `Draw` method)

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



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

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



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

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

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

2.8. ROI Main Properties

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

Save and load

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

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

ROI Classes

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

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

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

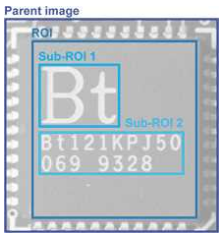
Attachment

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

Nesting

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

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



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

Cropping

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

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

Resizing and moving

- ROIs can easily be resized and positioned by two functions and dragging handles:
  - `EBaseROI.Drag` adjusts the ROI coordinates while the cursor moves.
  - `EBaseROI.HitTest` informs if the cursor is placed over a dragging handle. Once the handle is known, the cursor shape can be changed by an `OnSetCursor` MFC event handler. HitTest is unpredictable if called while dragging is in progress.
    HitTest can be used in an OnSetCursor MFC event handler to change the cursor shape, or before a dragging operation like OnLButtonDown,
    (or EvSetCursor and EvLButtonDown in Borland/CodeGear's OWL)
    (or FormMouseMove and FormMouseDown in Borland/CodeGear's VCL).
    In VB6, MouseDown, MouseMove,MouseUp events return the current cursor position in twips rather than pixels, so conversion is mandatory.

2.9. Flexible Masks

ROIs vs flexible masks

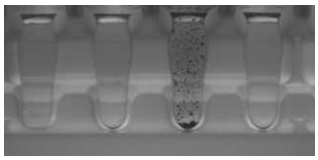
ROIs and masks restrict processing to part of an image:

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

Flexible Masks

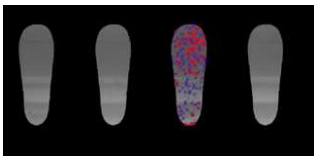
A flexible mask is a BW8 image with the same height and width as the source image. It contains shapes of areas that must be processed and ignored areas (that will not be considered during processing):

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



Source image

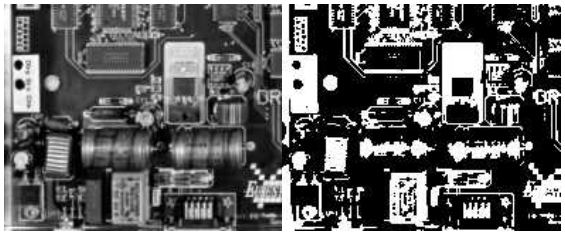
Associated mask



Processed masked image

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

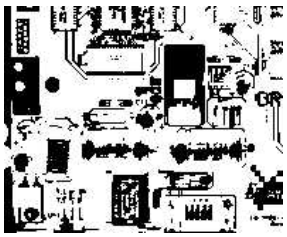
Flexible Masks in EasyImage



Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

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



Resulting image

EasyImage Functions that support flexible masks

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

- AutoThreshold.
- Histogram (function `HistogramThreshold` has no overload with mask argument).
- RmsNoise, SignalNoiseRatio.
- Overlay (no overload with mask argument for BW8 source images).
- ProjectOnAColumn, ProjectOnARow (Vector projection).
- ImageToLineSegment, ImageToPath (Vector profile).

Flexible Masks in EasyObject

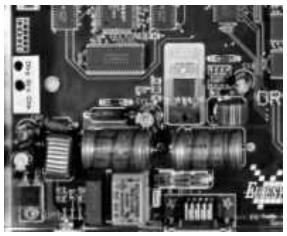
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

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

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

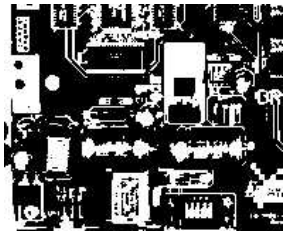
EasyObject functions that create flexible masks



Source image

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

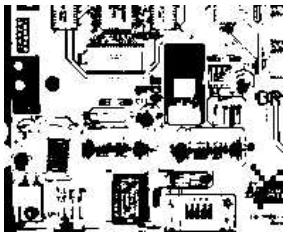
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers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



BW8 resulting image that can be used as a flexible mask

2) ECodedElement.RenderMask: from a blob or hole

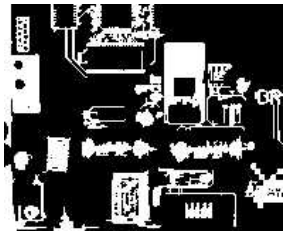
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement.RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

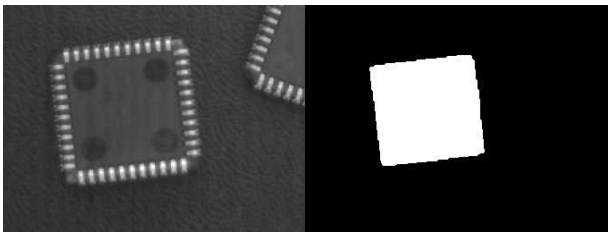
3) EObjectSelection.RenderMask: from a selection of blobs

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



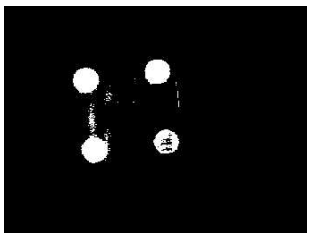
BW8 resulting image that can be used as a flexible mask

Example: Restrict the areas encoded by EasyObject



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

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



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

2.10. Profile

Profile Sampling

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

- `EasyImage.ImageToLineSegment` copies the pixel values along a given line segment (arbitrarily oriented and wholly contained within the image) to a vector. The vector length is adjusted automatically. This function supports flexible mask.
- A path is a series of pixel coordinates stored in a vector.
  `EasyImage.ImageToPath` copies the corresponding pixel values to the vector. This function supports flexible mask.
- A **contour** is a closed or not (connected) path, forming the boundary of an object.
  `EasyImage.Contour` follows the contour of an object, and stores its constituent pixels values inside a profile vector.

Profile Analysis

The profile can be processed to find peaks or transitions:

- A transition corresponds to an object edge (black to white or white to black). It can be detected by taking the first **derivative** of the signal (which transforms transitions (edges) into peaks) and looking for peaks in it.
  `EasyImage.ProfileDerivative` computes the first derivative of a profile extracted from a

gray-level image.
The `EBW8` data type only handles unsigned values, so the derivative is shifted up by 128. Values under [above] 128 correspond to negative [positive] derivative (decreasing [increasing] slope).

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

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

Profile Insertion Into an Image

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

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

3. 3D Tools

3.1. Understanding 3D Concepts

Basic Concepts

Easy3D

Easy3D is a set of tools for solving computer vision problem using 3D acquisition and processing. Easy3D supports laser line triangulation for fast and precise acquisition of depth maps.

Depth maps are gray scale images where each pixel represents a displacement in the third dimension. Because of the acquisition procedure, they are usually not dimensionally correct. So, while Open eVision 2D image operators are compatible with depth maps, you should not use them for processes requiring precise measurements.

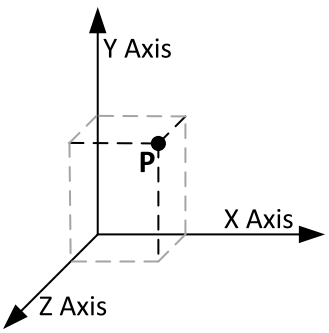
Easy3D provides a calibration tool to generate corrected, metric point clouds and meshes from depth maps. Most 3D operators work on point clouds or meshes. The included export functions to the standard PCD file format allows integration with other 3D tools.

Easy3D also allows the computation of ZMaps. A ZMap is the projection of a point cloud on a given reference plane. Like depth maps, ZMaps are gray scale images, but are also dimensionally correct. As such, they can be used with all Open eVision 2D functions.

All the Easy3D tools are placed in the Easy3D namespace.

3D representation

Open eVision uses a right-handed cartesian 3D coordinate system. In this system, each 3D point is represented by its 3 coordinates X, Y and Z.



Open eVision provides different containers to store 3D objects :

- Depth maps
- Point clouds
- Meshes
- ZMaps

Depth map

A depth map is a way to represent a 3D object using a 2D grayscale image where each pixel (u, v) in the image contains a third coordinate as its gray value.

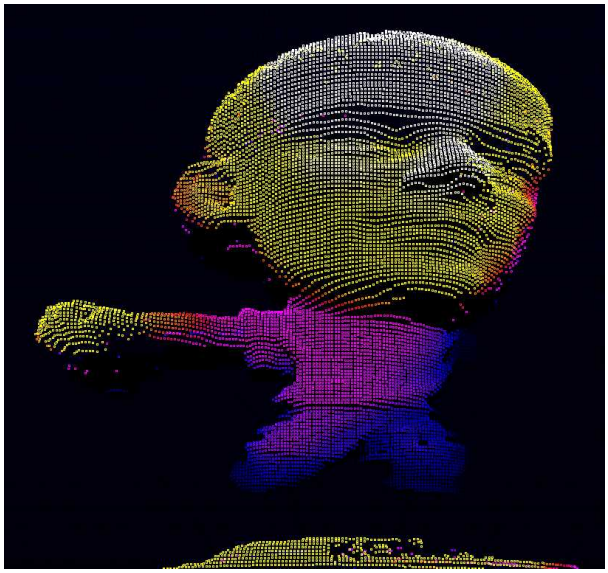
The grayscale values of a depth map do not necessarily represent a Z metric coordinate. In the context of a laser triangulation setup, these values represent the displacement of the laser line profile, which is not the physical height of the 3D surface.

A depth map contains a gray scale image coded on 8 or 16 bits. One specific gray value, called the undefined value, is reserved for the representation of invalid pixels. By default, this value is 0 for all depth map types.

The calibration process aims to convert the depth map representation to real, metric 3D representations such as point clouds or meshes.

Point cloud

A point cloud is a set of 3D points (x, y and z coordinates) representing the scanned object in the world metric space.



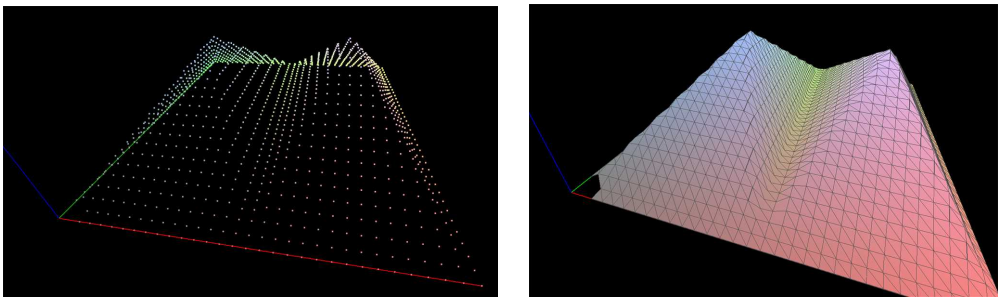
In addition to the calibration process included in Easy3D, point clouds can be produced using various 3D acquisition techniques, like stereo reconstruction or time of flight cameras.

Mesh

A Mesh is a geometric representation of a 3D surface, a set of connected 3D points.

In an `EMesh` object, 3 points are connected to define a triangle.

This kind of 3D representation is also called a "triangle mesh".



A point cloud and the corresponding mesh (displayed with Open eVision `E3DViewer`)

An `EMesh` object contains a point cloud and the indexes of the vertices of all mesh triangles.

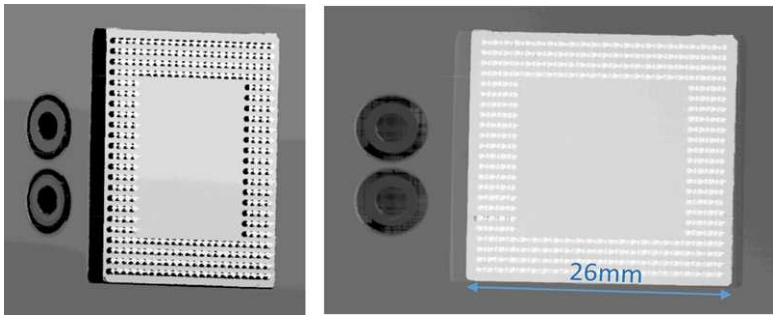
`EMesh` uses a metric space representation that can be generated from a depth map and that can be used to produce a ZMap.

ZMap

ZMaps are another representation for 3D data.

- They are grayscale images like depth maps but represent metric and corrected 3D points.
- They are convenient representations for measurement and matching.
- They are compatible with most of the 2D processing functions.

ZMaps are generated by the projection of a point cloud or a mesh onto an arbitrary 3D plane.



A depth map and the corresponding ZMap

A ZMap contains an image in which each pixel value represents a positive distance from the reference plane.

Use the method `AsEImage()` to obtain a reference to the contained image.

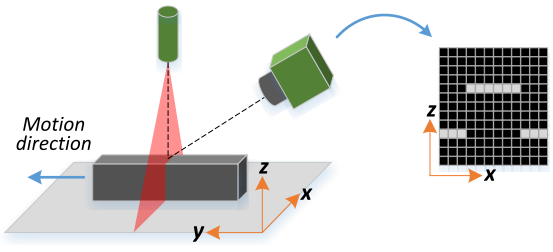
A ZMap also contains the following information:

- The transformation from the World coordinates to the ZMap coordinates.
- The size of a pixel, called the "resolution".

Like in a depth map, a specific pixel value is reserved to represent invalid pixels. To get this pixel value, use the method `GetUndefinedValue()`.

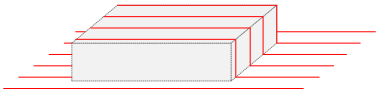
Laser Triangulation

In a laser-line triangulation system, a laser line is projected on the object to measure. A camera is looking at the laser line from a different point of view. The line deformation observed by the camera contains the shape information of the measured object.



The scanning of the object consists in moving it under the laser line and recording multiple images.

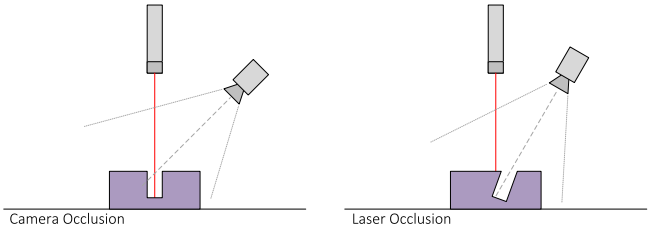
From the scanning you can reconstruct its 3D shape.



Occlusions

Using the laser triangulation method, the laser may be unable to reach some parts of the object or the camera may be unable to view them. This is called occlusion.

- On the left illustration, the camera does not see the bottom of the hole, inducing camera occlusion.
- On the right illustration, the laser does not reach the bottom of the hole, inducing laser occlusion.

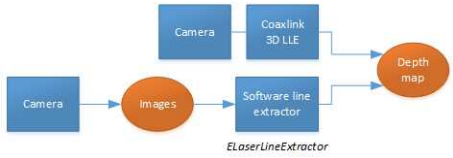


You can limit or avoid occlusions by using advanced scanning methods, for example by using two cameras or two lasers.

The Laser Line 3D Acquisition Pipeline

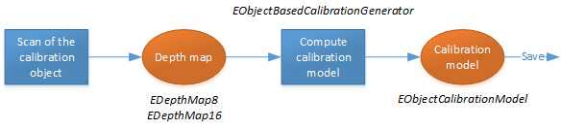
The 3D acquisition pipeline starts with the acquisition of a laser line profile and ends up with the point cloud, mesh or ZMap.

The source material for 3D processing is the depth map, coming from a Coaxlink Quad 3D-LLE or generated from a list of images. Two types of depth map are available, one for each different pixel coding scheme (8- or 16-bit).



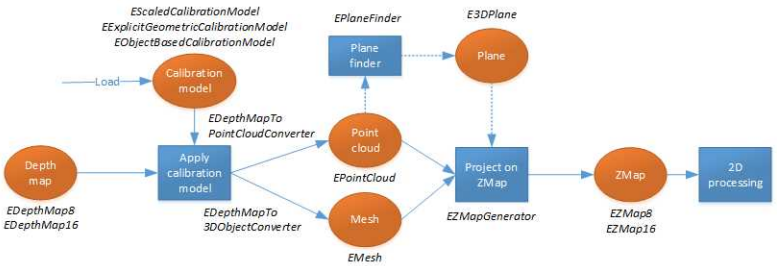
The generation of a depth map, from a hardware or a software source

Some processing methods can use the depth map directly, but most measurement and matching processes need metric, distortion-free representations. Calibration of the laser triangulation setup is therefore required. Calibration is used to turn the depth map into a point cloud or mesh expressed in a metric space that we call “world space”.



The generation of an object based calibration model, from a scan of the reference object

A point cloud is a list of 3D points, expressed in a world space coordinate system. The point cloud can be projected on a plane, producing a ZMap, which is a convenient and effective representation for 2D processing with a metric scale.



The workflow from the depth map to the ZMap

The following sections describe the classes and methods useful for a 3D workflow. The Use Case - Measuring a Remote Controller goes through this processing pipeline.

3.2. Easy3D - Using 3D Toolset

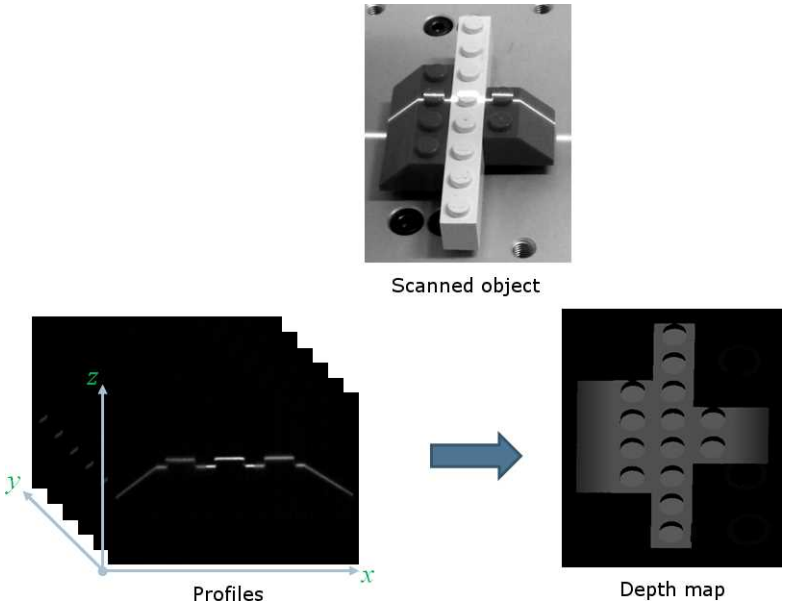
Laser Line Extraction

A Laser Line Extraction (LLE) algorithm is required to create a depth map from a sequence of profiles of the object captured by the camera sensor.

The objective of an LLE algorithm is to measure the line position along a vertical profile in every column of a sensor frame, within a user-defined region of interest (ROI).

For every step of the object position, the detection analyzes each column of a frame individually and produces a row of output positions, stored as gray values.

The figure below illustrates a depth map generation.

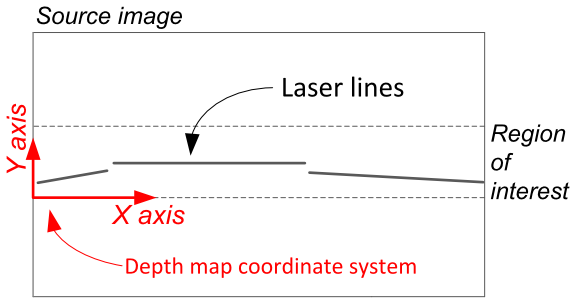


The `ELaserLineExtractor` class provides the laser line extraction functionality in Open eVision. It implements several algorithms to extract the laser line:

- **Maximum detection** returns the position of the pixel of maximum intensity. It's the fastest method but it doesn't support sub-pixel precision.
- **Peak detection** approach detects local maxima. If several maxima are detected, the one with the highest intensity is returned. The position is returned with sub-pixel precision.
- **Center of gravity** algorithm is suitable when the laser line is spread over several pixels. The position is returned with sub-pixel precision.

You can also set a threshold to exclude pixels with low intensity.

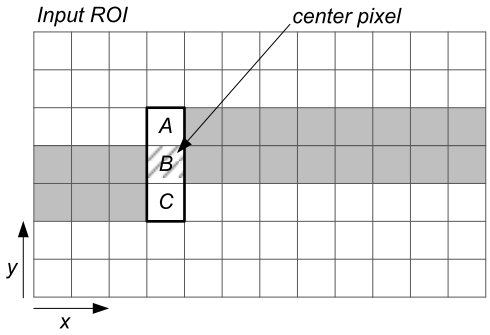
The line position returned by the laser line extraction algorithms is relative to the bottom of the region of interest. So, values in the depth map range from 0 (bottom of the ROI) to the height of the ROI.



Low-pass linear filter

Optionally, you can apply a low-pass linear filter in front of the line extraction in order to reduce noise and high frequencies in the image.

The low-pass filter applies a convolution operator on a 1 x 3 sliding window. The 3 elements of the convolution kernel (A, B and C) are configurable, accepting any positive integer. The figure below illustrates the positioning of the convolution kernel elements within a given ROI.



You can activate the low-pass filter for any of the laser line extraction methods with the method `ELaserLineExtractor::SetEnableSmoothing(true/false)`. Parameters A, B and C are set with `ELaserLineExtractor::SetSmoothingParameters(A, B, C)`.

Calibration

The calibration is used to apply the transformation between a depth map and a point cloud or a mesh.

There are 3 ways to setup this conversion:

- Apply a simple scale on the pixel coordinates of the depth map (`EScaleCalibrationModel` class)
- Use the explicit geometric model (`EExplicitGeometricCalibrationModel` class)
- Use the object-based calibration approach (`EObjectBasedCalibrationModel` class)

These models share the same base class `ECalibrationModel` and exposes the method `Apply ()`, which is used to apply the conversion between a depth map pixel and a 3D point. It takes as input the coordinates of one point in a depth map and it returns the coordinates of the corresponding point in the 3D space.

The method `Apply` is not aware of the possible mirroring of the corresponding depth map and cannot make use of `EDepthMap::EAxisSystemType` (see below). If necessary (when the corresponding depth map is vertically mirrored) the y coordinates should be flipped before calling the `Apply` method.

- The class `EDepthMapToPointCloudConverter` generates a point cloud from a depth map, using one of the calibration models.
- The class `EDepthMapToMeshConverter` generates a mesh from a depth map, using one of the calibration models.

By convention:

- The origin of the referential is the lower-left corner of the depth map.
- The center of the first pixel at the lower-left corner is at x = 0.5 and y = 0.5.
- The center of the pixel at the upper-right corner is at x = width - 0.5 and y = height - 0.5 where width is the width of the depth map and height is its height.

Mirrored depth maps

By default, Easy3D considers that the origin of the 3D axis of the depth map is the bottom left of the internal image buffer, and the Y axis is pointing up. This means that the depth map image is not seen as vertically mirrored compared to the real world image of the scanned object.

Nevertheless, depending on your acquisition setup this mirroring can happen (for example if the direction of the scan is inverted).

If this is your case, you can set the `EDepthMap::EAxisSystemType` to `EAxisSystem_UpperLeftCorned`, meaning that the origin of the 3D axis is on the upper left corner and the Y axis is pointing down.

This value changes the behavior of the methods :

- `EObjectBasedCalibrationGenerator.Compute`
- `EDepthMapToPointCloudConverter.Convert`

- `EDepthMapToMeshConverter.Convert`

Scale calibration

The scale model (`EScaleCalibrationModel`) only applies a simple factor on the X, Y and Z axis. These factors are the only parameters of `EScaleCalibrationModel`.

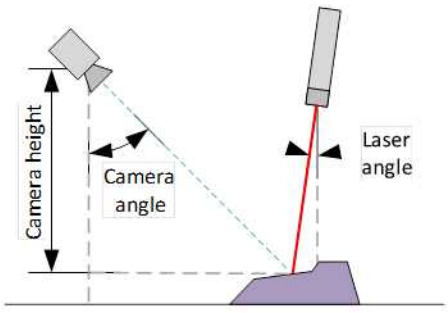
For depth maps coming from laser triangulation setup, this transformation does not produce corrected, metric points. It's main use is to display depth maps as 3D data with the `E3DViewer` class.

Explicit geometric calibration

The explicit geometric model (`EExplicitGeometricCalibrationModel`) defines a simple and ideal laser triangulation setup. The explicit calibration makes some strong assumptions on the setup geometry and can only be used when a minimum set of parameters are known:

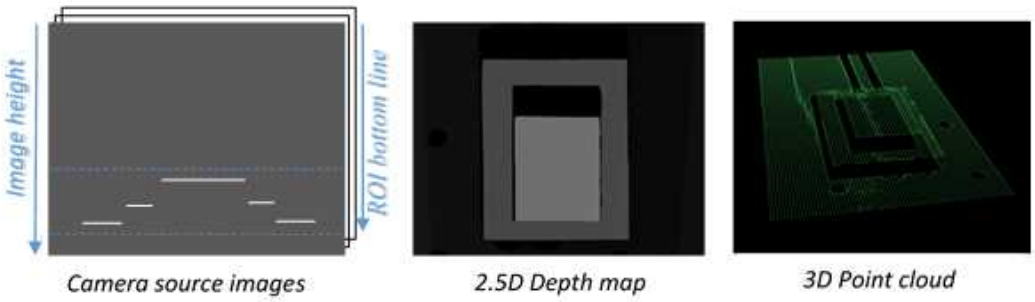
- The angles of the camera and the laser plane, in the counter clockwise direction. The camera angle must be positive.
- The height of the camera above the scanned object.
- The field of view of the camera defined by the sensor size (mm) and the optical focal length (mm).
- The physical distance between two line scans of the depth map (depends on acquisition rate and motion speed).
- The size of the image and the ROI origin used in laser line extraction (between the top (0) and the bottom (height) of the image).

Use the "Easy3D_Setup_Configuration.xlsx" spreadsheet to compute and check your setup configuration and parameters.



Explicit calibration setup with camera angle, laser angle and camera height

The setup of an explicit geometric calibration uses the constructor of the `EExplicitGeometricCalibration` class.



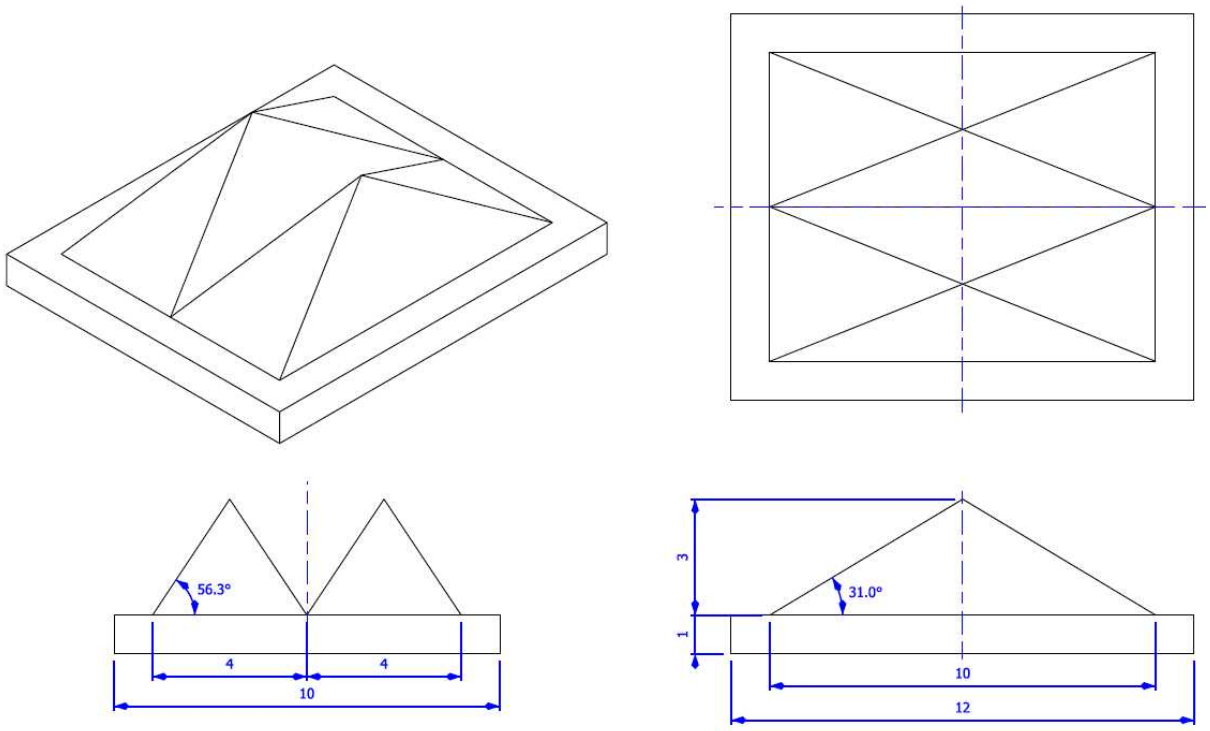
Object-based calibration

Object-based calibration gives real world, metric, coordinates from an arbitrary laser triangulation setup. From the scan of a reference object, the calibration process tries to calculate all the parameters required for the transformation to the world space (position and attributes of the camera, position of the laser plane, relative motion of the object, optical distortion…).

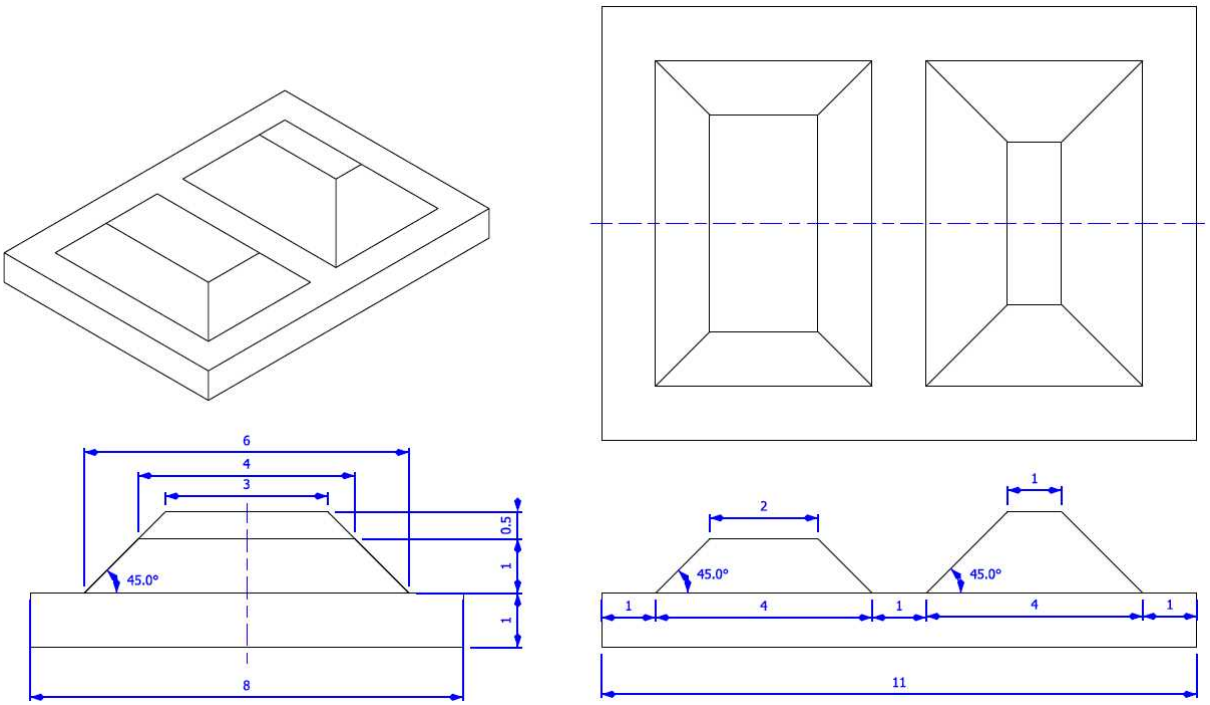
In the current version of Open eVision, 2 types of reference objects are supported, the double pyramid and the truncated pyramids. The calibration process operates only from acquisitions of such objects only.

Depending on the target object size, a calibration object must be manufactured as a scaled version of one of the reference objects described below.

CAD files in various standard formats are available on request.



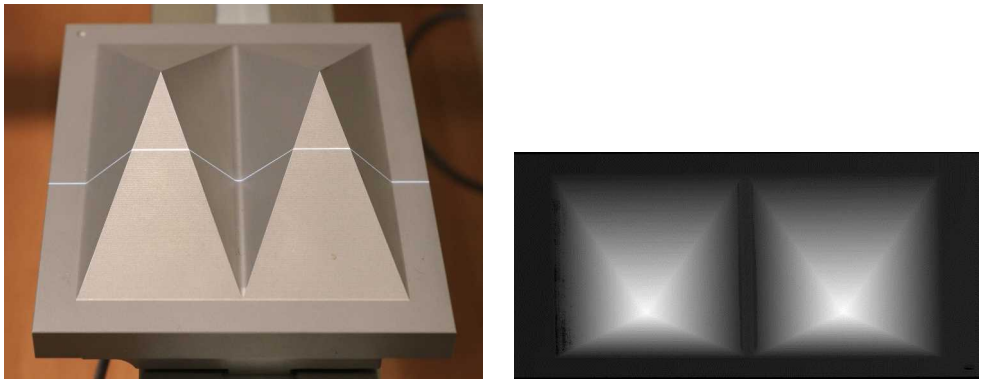
First reference object: the double pyramid



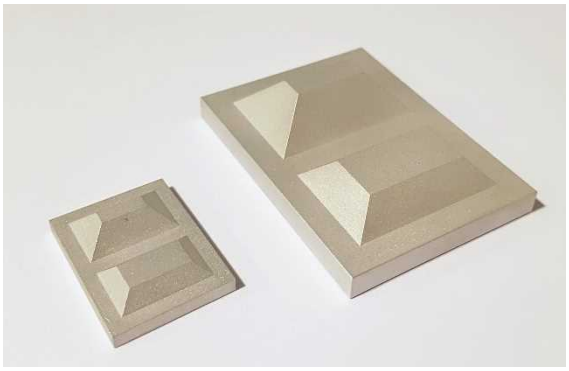
Second reference object: the truncated pyramids

- The class `EObjectBasedCalibrationModel` is the container for the object based calibration model.
- The class `EObjectBasedCalibrationGenerator` performs the computation of such a model.
  - Its `Compute()` method takes a depth map as input and returns a calibration model.
  - The computation of the calibration model may take some time.
  - Use the `Save()` method to store the calibration model for later use.
- The real size of the calibration object is set relatively to the dimensions of the figure above, by the `SetCalibrationObjectScale()` method.
  - The dimensions of the figures above are expressed in millimeters.
  - For example, if the double pyramid calibration object has a physical width of 100 mm, the method should be called with a factor of 10 (10 x 10 = 100 mm).
- The calibration object model must be set with the method `SetCalibrationObjectType()`. The possible values are `DoublePyramid` and `TruncatedDoublePyramid`.

If possible, prefer the truncated pyramids model as it does not show symmetry and thus prevents possible orientation confusion.



A manufactured double pyramid under the laser triangulation device and the corresponding captured depth map



Truncated pyramids of different sizes

Recommendations for the acquisition of the calibration object

To ensure a reliable and precise calibration, the following conditions must be satisfied during the acquisition of the calibration object:

- All the calibration object faces must be visible on the depth map (that is a constraint on the camera and laser orientation)
- The calibration object must be horizontal (relative to the motion axis)
- There must be no other object higher than the calibration object in the depth map
- The depth map must have at least 200 x 200 pixels resolution
- The calibration object must cover at least 50% of the defined pixel of the depth map

Point Cloud

Coordinates Transformations

Affine Transforms

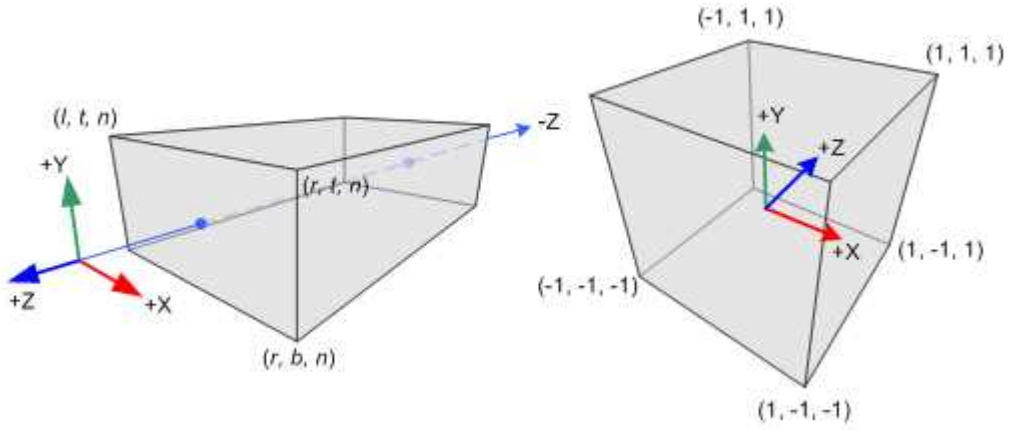
Affine transforms allow you to reposition the point cloud inside the 3D space.

Open eVision provides you with the following basic transformations:

- Rotation around the X, Y or Z axis
- Translation along the X, Y and/or Z axis
- Scaling, around the origin, and either isotropic (the same in all directions) or anisotropic (different along the different axis)

It also provides you with projection transformations, both orthographic and perspective:

- An orthographic projection transforms a volume of space in the shape of a rectangular parallelepiped (and the points it contains) into the canonical view (a cubic space of size 2 and centered on the origin).



- A perspective projection transforms a volume of space in the shape of a frustum (basically a truncated pyramid) into the canonical view. This projection allow you to simulate the perspective effect given by an eye or a camera.

Reducing a Point Cloud

Cropping

Cropping allows you to exclude points from the point cloud based on geometrical considerations.

Open eVision provides the following cropping functions:

- `ESimpleCropper`: simple cropping on the X, Y and/or Z coordinates (aligned rectangle 3D region)
- `ERectangularCropper`: cropping the points outside (or inside) an oriented rectangular parallelepiped
- `ESphericalCropper`: cropping the points outside (or inside) a sphere.
- `EPlaneCropper`: cropping the points depending on their position with respect to a plane

These classes produces a new point cloud with the selected points.

Decimation

The random decimator, `ERandomDecimator`, decimates a point cloud by copying a specified number of points, randomly selected, to a new point cloud.

Specify the number of points to keep as parameter of the constructor.

```
EPointCloud pc;
pc.LoadPCD("c:\\images\\data.pcd");
// Explicitely decimate the point cloud
ERandomDecimator decimator(5000);
EPointCloud pcDecimated;
decimator.Decimate(pc, pcDecimated);
pcDecimated.SavePCD("c:\\images\\decimatedData.pcd");
```

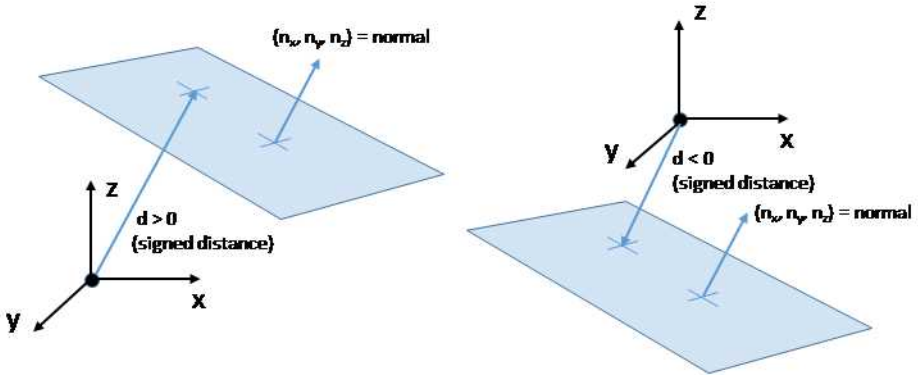
Managing Planes

E3DPlane

A plane can be represented as an `E3DPlane` object.

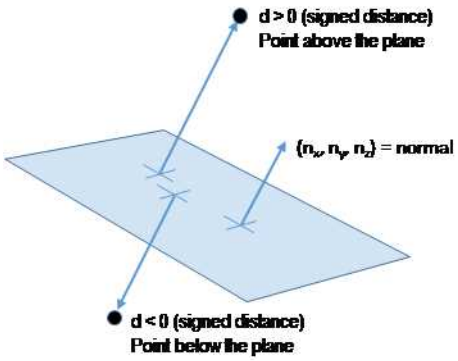
This plane is characterized by:

- Its normal which is a vector of norm 1, perpendicular to the plane.
- Its signed distance from the origin, which is the smallest distance from the origin to the plane. The signed distance is positive when the vector binding the origin to the closest point on the plane has the same direction as the normal and is negative when it has the opposite direction.



Once a plane is defined, you can measure the signed distance between this plane and any point in the space (using the method `DistanceTo()`):

- A positive distance means that the vector connecting the plane to the point has the same direction as the normal.
- A negative distance means that the vector has the opposite direction.



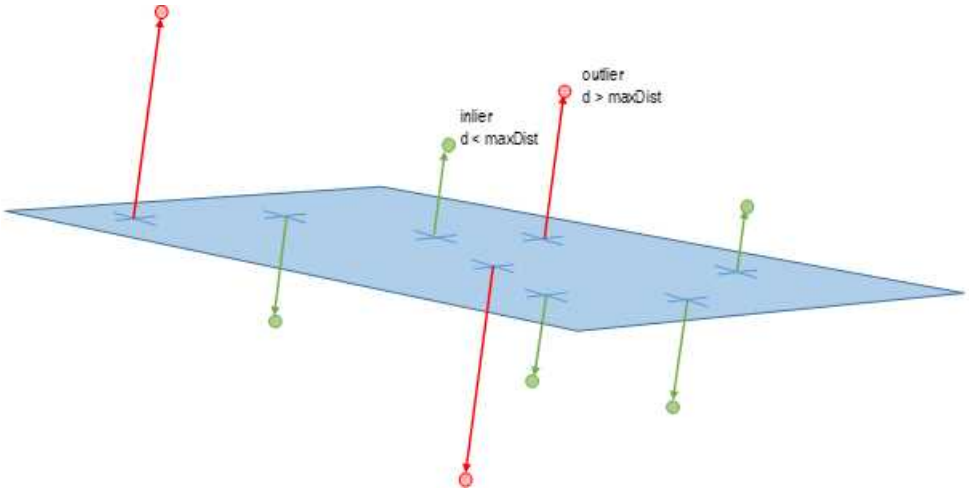
EPlaneFinder

You can search for a plane in a point cloud using the object `EPlaneFinder` object.

The main parameters of this object are:

- The maximum distance between the searched plane and a point that belongs to this plane.
- The expected ratio between the numbers of inliers and the total number of points in the point cloud.
  - An **inlier** is point that belongs to a plane (closer than this maximum distance).
  - An **outlier** is a point that is not an inlier.

The picture below illustrates how points of the space are classified as inliers (in green) and outliers (in red) according to their distance to the searched plane.

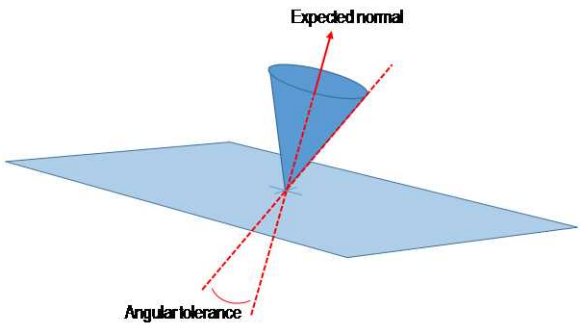


A `EPlaneFinder` object produces a `E3DPlane` object. The algorithm searches for a plane containing as many inliers as possible. This plane is the biggest plane if the samples are evenly distributed.

The maximum distance between the plane and the inliers is a mandatory parameter: it should include the deviation due to the noise but also take warpage into account.

The parameter that specifies the ratio of inliers with respect to the total number of points has a default value of 0.3, meaning that we estimate that about 30% of the points belong to the plane. This parameter is not as critical as the maximum distance but it affects the maximum time the algorithm will spend in searching a plane as well as its robustness.

Optionally, you can specify the expected normal vector to the plane to search. In that case, you should also specify an angular tolerance with respect to this expected direction.



When an expected normal is specified, the algorithm only searches for a plane that satisfies the condition. Setting this condition might speed up the plane search.

Finally, it is important to note that, by default, the `EPlaneFinder` decimates the input point cloud to accelerate the search. The default decimator reduces the input point cloud to 10000 points. Alternatively, you can disable this decimation, or you can decimate a point cloud explicitly, by using an `ERandomDecimator` object and use the decimated point cloud as input for the `EPlaneFinder`. In this case you should disable the default decimator.

Once the main plane is found, a fit is done on all the inliers points and the result is returned (see `EPlaneFitter` below).

EPlaneFitter

The `EPlaneFitter` operator computes a fit on all the points of a point cloud and returns a `E3DPlane` object.

Aligning

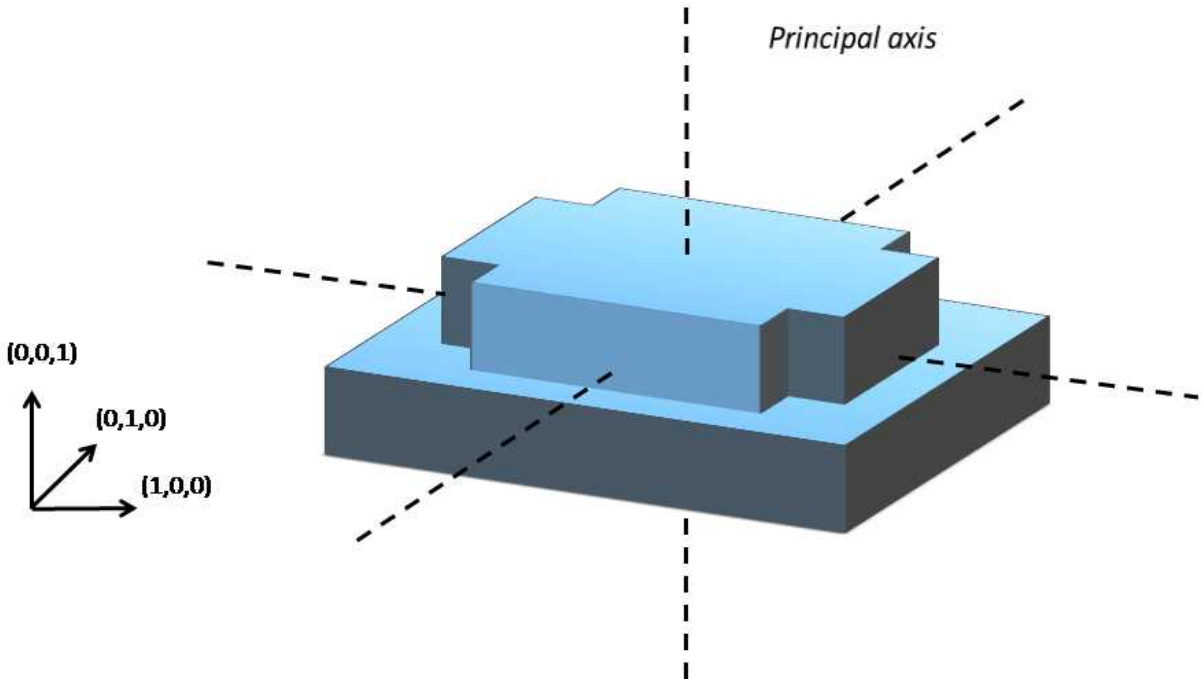
EPrincipalAxisExtractor

The `EPrincipalAxisExtractor` computes the “principal axis” of an object from a point cloud (`EPointCloud`) and returns a `E3DTranformMatrix` containing a solid transformation that defines a new orthogonal basis.

This new orthogonal basis has the following characteristics:

- The center is the center of gravity of the point cloud.
- The axis are oriented along the “principal axis” of the object. This is the result of the “PCA” calculation (principal axis analysis).
- The directions of the axis are selected so that the new basis is as close as possible of the basis defined by the reference transformation.

The next figure illustrates the orientation of the principal axis of an object.



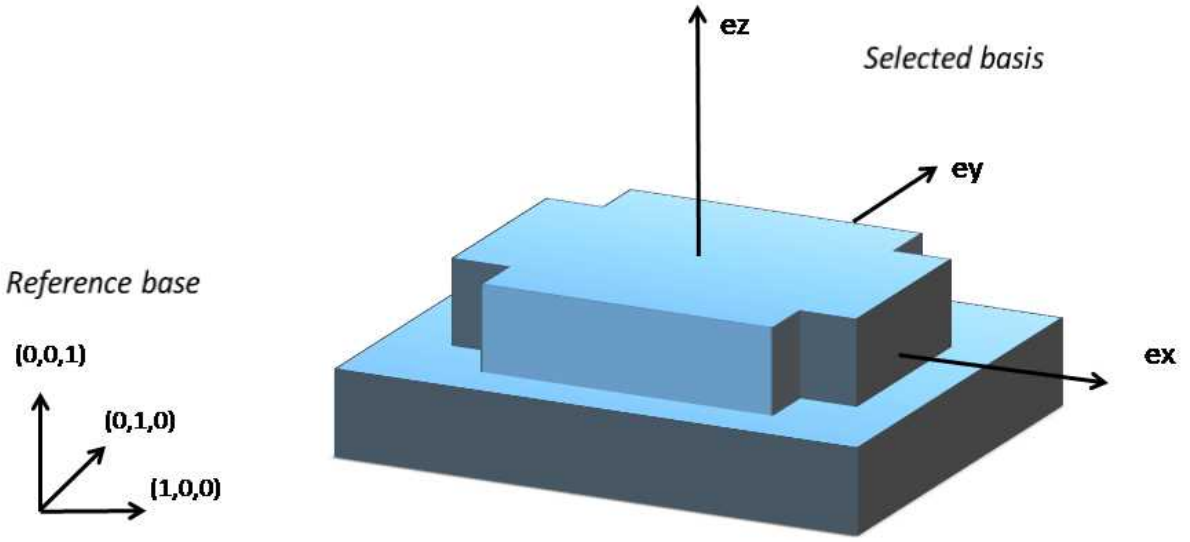
The principal axis extraction is done using the `Extract()` method that takes a `EPointCloud` as input and returns an `E3DTransformMatrix`. Optionally, you can pass 3 other output parameters by reference to retrieve the value of the standard deviation along the 3 principal axis.

You can use the returned `E3DTranformMatrix` object to transform the 3D coordinates of a point. For example, apply the transformation matrix to the origin (0, 0, 0) to return the center of gravity of the object.

Specification of a reference transformation

The reference transformation is an optional parameter of the `EPrincipalAxisExtractor` object. It defines a reference basis used to select an orthogonal basis out of the principal axis. The selected basis will be the closest to the reference basis.

If no reference transformation was supplied, the default reference basis is ((0, 0, 1), (0, 1, 0), (0, 0, 1)), that corresponds to the identity transformation. On the figure below, the default reference basis determines the direction of the axis ***ex****,* ***ey*** *and* ***ez****.*



EFeaturesAligner

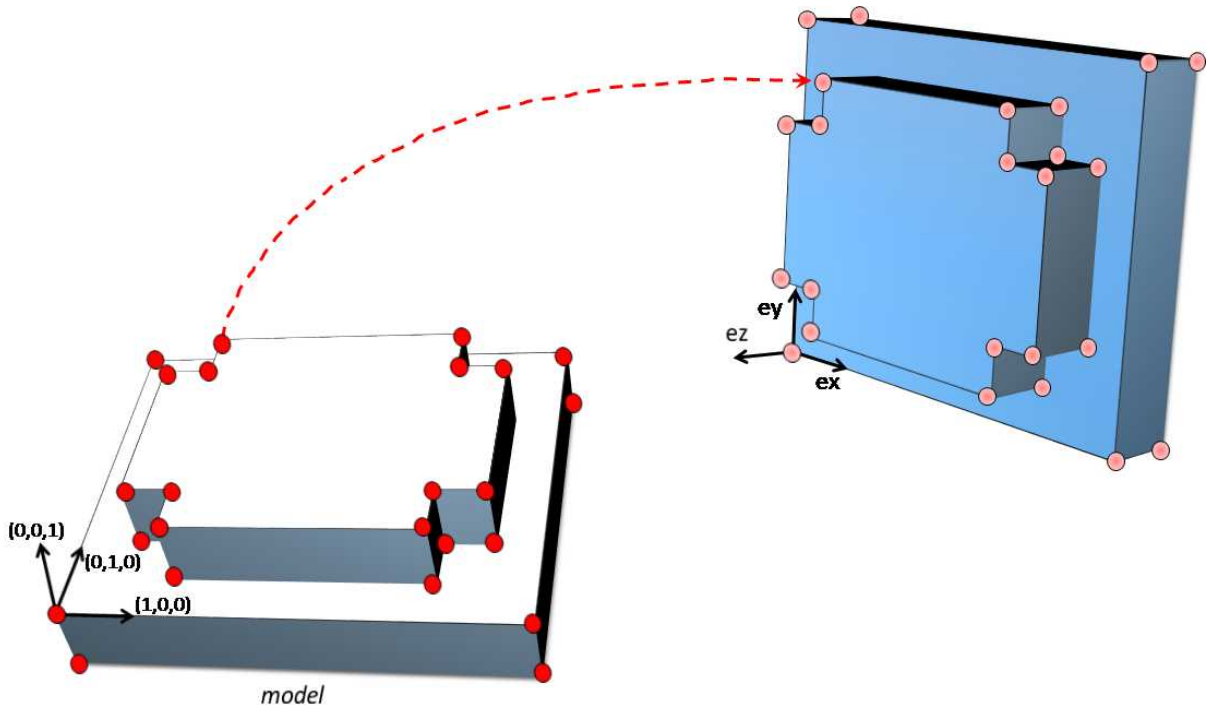
A `EFeaturesAligner` object finds the best transformation that maps a list of points to another list of points.

- The first list of points is called the "model". It is stored in the `EFeaturesAligner` object.
- The second list of points is called "measured points". It is passed as a parameter to the `Compute()` method. If successful, the result of this method is a `E3DTransformMatrix` object.
- The 2 lists should form matching pairs. In other words, the first point of the first list matches the first point of the second list, the second point of the first list matches the second point of the second list, and so on…

With the Polarity parameter, you can define which transformation is returned. It can be either:

- The one that moves one point from the first list (the model) to the second list of points (the measured points) if the polarity parameter is set to `EAlignmentPolarity_ModelToMeasured` (default).
- The one that moves a point from the second list (the measured points) to the first list (the model) if the polarity parameter is set to `EAlignmentPolarity_MeasuredToModel`.

The figure below illustrates the computation of the alignment transformation. In this example a model is aligned to an object using the coordinates of their corners.



Once the transformation is computed, use the method `GetOrthoBasis` of the `E3DTransformMatrix` object to get the basis (**ex**, **ey**, **ez**) and the center point **t** that defines the new basis.

You can also apply the computed transformation on any 3D point as illustrated in the code below.

```
EFeaturesAligner alignTool;
E3DTransformMatrix alignBase;
E3DPoint ex, ey, ez, t;
std::vector<E3DPoint> model3d;
std::vector<E3DPoint> points3d;
// add points to model3d and points3d
// ...
alignTool.SetModelPoints(model3d);
alignBase = alignTool.Compute(points3d);
// Get the orthogonal basis and store it in ex, ey, ez and t
alignBase.GetOrthoBasis(ex, ey, ez, t);
// Applying the transformation on point P1, results in point P1b
E3DPoint P1 = E3DPoint(...) ;
E3DPoint P1b = alignBase*P1;
```

As you can see, the application of the transformation on a point is simply done by multiplying the transformation matrix by the point (as done in the example above).

On the other hand, if you need to transform a point cloud or a list of points, it is more efficient to use the `ApplyTransform()` method of an `EAffineTransformer` object.

Mesh

A mesh is a geometric representation of a 3D surface. The surface is defined by a triangle mesh connecting the 3D points. Like a point cloud, a mesh is expressed in the metric space.

Like a point cloud, you can generate a mesh from a depth map and use it to produce a ZMap.

Generation

An `EMesh` object is generated from a depth map using the `EDepthMapToMeshConverter` class.

Like `EDepthMapToPointCloudConverter`, this class uses a calibration model to transform the depth map pixels to 3D world positions. In addition, the depth map pixel connectivity is used to build the triangle mesh. Adjacent pixels produce surface triangles.

Use `SetCalibrationModel()` to select a calibration model and the method `Convert()` to generate an `EMesh` from an 8 bits or 16 bits depth map.

Access and usage

In an `EMesh` object the 3D world positions are stored as an `EPointCloud` (accessible through the method `GetPointCloud()`). The triangle mesh is stored as an array of point indexes, where 3 consecutive indexes define a triangle. The method `GetTriangleIndexes()` provides a read-only access to the triangle mesh.

You can use either the Open eVision proprietary format to save and load `EMesh` objects using the `Save()` and `Load()` methods, or use the STL standard file format (https://en.wikipedia.org/wiki/STL_(file_format)) using the `SaveSTL()` and `LoadSTL()` methods which respectively write to and read from ASCII STL files.

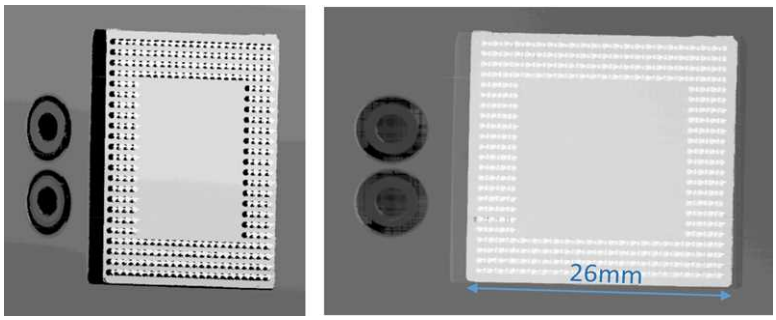
You can use an `EMesh` to produce a ZMap (see "Generating a ZMap" on the next page). Because an `EMesh` represents a surface, the so generated ZMap can show better continuity and less undefined pixels.

ZMap

Generating a ZMap

A ZMap is the projection of a point cloud or a mesh on a reference plane, with the distance coded as gray scale values:

- They are grayscale images, compatible with all Open eVision 2D libraries.
- They are distortion free, with affine transformation from/to metric coordinate system.



**A depth map (left) and the corresponding ZMap (right),
with default generation parameters and undefined pixel filling enabled**

All Open eVision 2D processing are available on ZMaps: filtering, thresholding, blob extraction, measuring with EasyGauge, model matching with EasyFind or EasyMatch…

The `EZMapGenerator` class implements the conversion from a point cloud or a mesh to a ZMap. With all parameters at default value, the `Convert()` method automatically chooses the projection plane, the orientation, the map size and the resolution.

Several methods are available to further control the conversion:

- `SetReferencePlane()` defines a world space projection plane. The values of the ZMap pixels are the distance of the point cloud to that reference plane.

  By default, the reference plane crosses the origin and is perpendicular to the world Z axis. The plane is defined as a `E3DPlane` object.

- `SetOrientationVector()` sets a world space vector representing the expected direction of the X (width) axis of the ZMap.

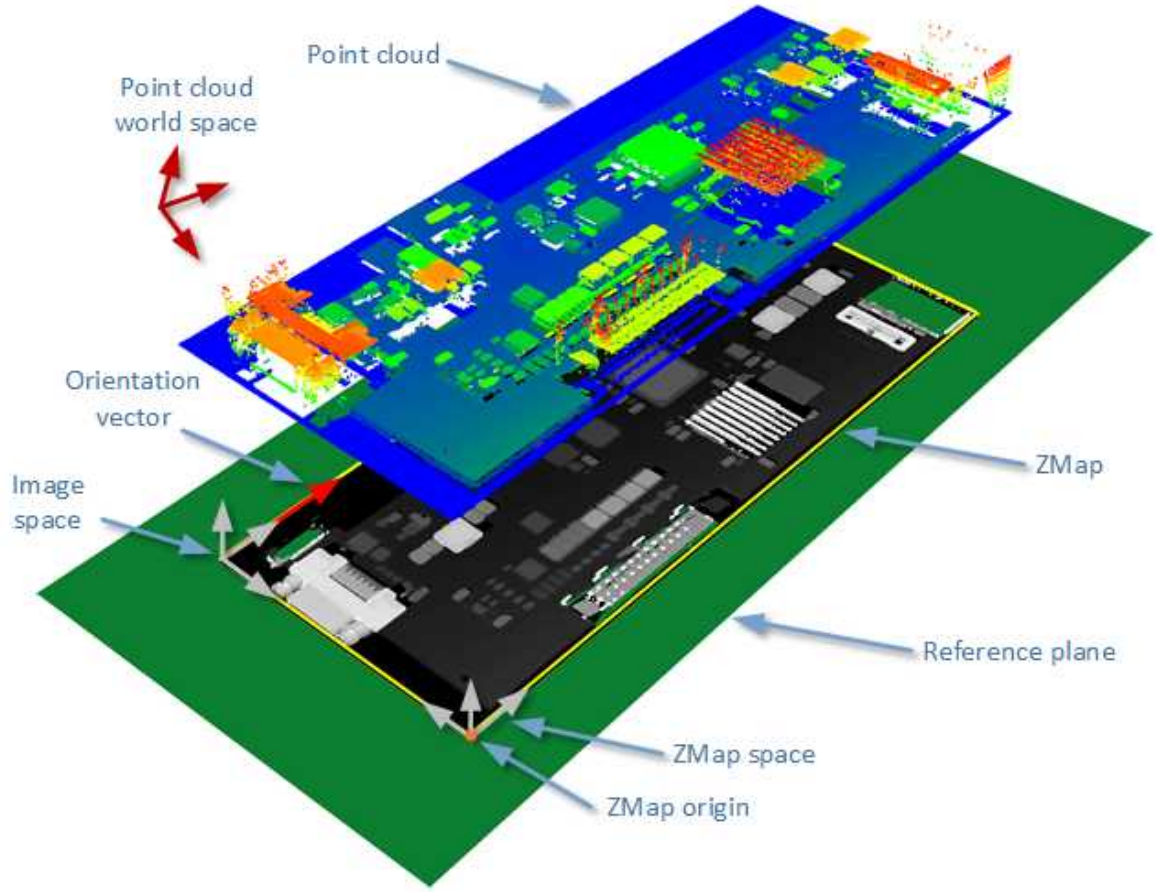
  The orientation vector allows to “rotate” the object around the normal of the reference plane.

- `SetOrigin()` specifies the world position that is on the ZMap lower left pixel (0, 0).
- `SetMapSize()` defines the resolution (number of pixels in X and Y axis) of the generated ZMap.

- `SetMapXYResolution()` adjusts the X and Y resolution of the ZMap pixels, in world space unit per pixel (for example mm/pixel). This value is used to compute the ZMap size (width and height), depending on the projected size of the point cloud on the reference plane.
- `SetMapZResolution()` sets the Z resolution, in world space unit per pixel unit (gray value). The Z resolution is used to compute the transformation of the distance to the reference plan to the integer 8, 16 or 32 bits pixel value.
- `EnableFillMode()` and `SetFillMode()` control the options used to fill the "hole" in the ZMap. A hole exists when no 3D point is projected in the ZMap at a pixel position.

The methods `SetReferencePlane()`, `SetOrientationVector()` and `SetOrigin()` are used to setup the transformation between the world space and the ZMap space. This transformation is rigid (distances are kept).

Alternatively, it is possible to directly set that transformation with the method `SetWorldToZMapTransform()` using a rigid matrix as parameter. In that case, the reference plane, the orientation vector and the origin parameters are ignored.

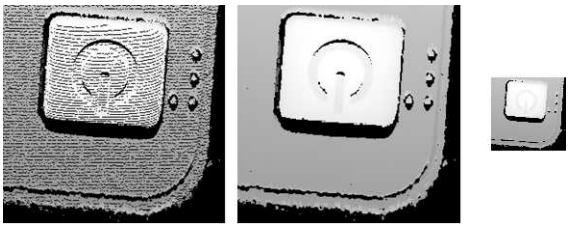


**The projection of a point cloud on a ZMap,
showing 3 coordinate systems: the world space, the ZMap space and the pixel space.**

The method `Convert()` performs the effective projection of a point cloud (`EPointCloud`) or a 3D object (`EMesh`) to the 8 bits and 16 bits ZMap.

When generating a ZMap from a point cloud, only individual points are projected on the ZMap. Depending on the point cloud density and the ZMap resolution, some regions of the ZMap may remain “undefined”. To get around this problem, adjust the resolution of the ZMap (`SetMapXYResolution` method) to remove “holes” on the ZMap.

By default, the point cloud to ZMap converter performs a filling algorithm. This process tries to replace undefined pixels with locally interpolated values.



Left: high resolution ZMap, the pixel scale exceeds the point cloud density
Center: the same generator parameters with the filling enabled
Right: a reduced ZMap scale/resolution, without filling

As a mesh defines a surface, its triangles are projected onto the ZMap plane. Thus, the generated image shows better continuity and less undefined pixels. However, the generation of a ZMap from an `EMesh` is slower than from an `EPointCloud`.

Managing the Coordinates

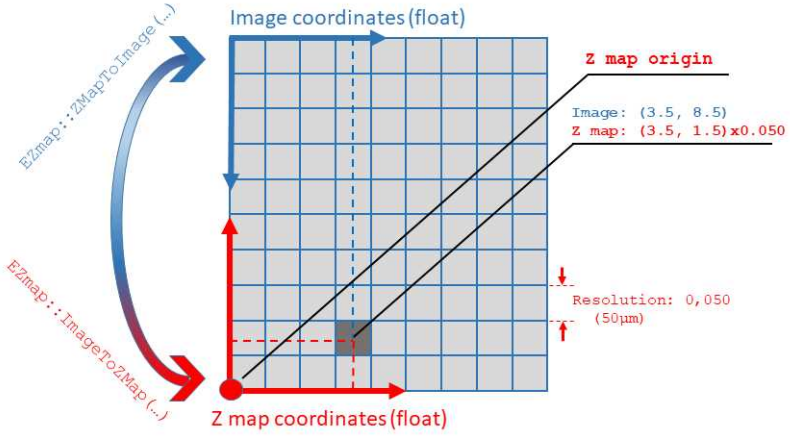
Coordinate systems on a ZMap

A ZMap has multiple coordinate systems:

- The **world space** system is the original, metric space from which the ZMap has been generated. Point clouds and meshes are expressed in the world coordinate system.
- The **ZMap space** is defined by a rigid transformation of the **world space**. The basis linked to this transformation is attached to the lower left corner of the ZMap.
- The **image space** is the system attached to the image representation of the ZMap. Its origin is the upper left corner of the ZMap and its unit length is one pixel along the X and Y axis.

The transformations between:

- The **image space** and the **ZMap space** include a scale factor.
- The **ZMap space** and the **world space** are solid transformations.



EZMap

The `EZMap` object exposes a set a methods to convert coordinates between world, ZMap and image spaces:

- `ImageToZMap` converts a 2D position in the image to ZMap coordinates.
- `ZMapToImage` is the reciprocal operation and converts a ZMap position to an image position.
- `ZMapToWorld` is a method to transform positions from the 3D ZMap space to the 3D world space. The world space is the original point cloud or mesh space.
- `WorldToZMap` is the reciprocal operation, converting from world space to ZMap.
- `ImageToWorld` and `WorldToImage` combine the functions above to transform directly from image space to world space (or the other way).

These methods only perform geometric transformations between the various coordinate systems and do not access the actual ZMap gray scale values.

The functions that accesses the pixel values are:

- `GetWorldPositionFromPixelPosition()` is a method transforming the actual pixel value at integer position (u, v) to the original world space. This method queries the ZMap internal representation to get the pixel value w and transform the pixel space (u, v, w) coordinates to a world space position.
- `GetPixelPositionFromWorldPosition()` is a method to get a pixel value from a world position. The world position is projected on the ZMap and the pixel value is returned. If the world position is outside the ZMap domain, the method returns `FALSE`.

Static Methods

EFilters class

The `EFilters` class contains static methods used to apply filters to ZMaps or depth maps.

RemoveNoise

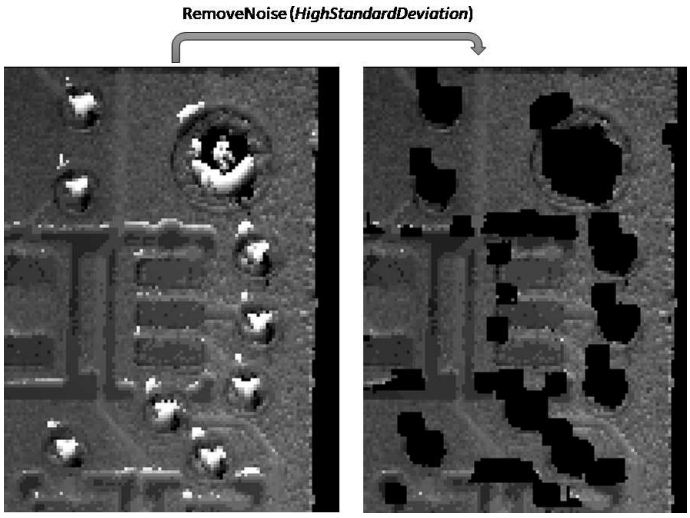
The `RemoveNoise()` method removes outliers from a depth map or a ZMap.

- It takes a depth map or a ZMap as input and generates a depth map or a ZMap respectively.

  The undefined points are not taken into account.

- It is based on a square moving kernel. The size of the kernel is (2 x `halfKernelSize` + 1) where `halfKernelSize` is a parameter of the method.
- The `threshold` parameter is scaled with regard to the Z resolution of the filtered depth map or ZMap.
- There are 3 variations of this filter, depending on the `method` parameter:
  - `ENoiseRemovalMethod_AbsoluteDifferenceFromMean` removes a point when it deviates from the average in the neighborhood, including itself. The threshold is an absolute difference.
  - `ENoiseRemovalMethod_RelativeDifferenceFromMean` removes a point when it deviates from the average in the neighborhood, including itself. The threshold is a multiple of the standard deviation.
  - `ENoiseRemovalMethod_HighStandardDeviation` removes a point when the standard deviation in the neighborhood, including itself, is higher than a defined threshold.

Example: Removing points showing a high standard deviation



The code below removes pixels with a standard deviation higher than a defined threshold.

```
// Load the ZMap data
EZMap16 zmap;
zmap.Load(...);
// Compute the filtered ZMap. The new ZMap is called filteredZmap
// The size of the kernel is 7x7, the threshold is 30.0
EZMap16 filteredZmap;
filteredZmap.SetSize(zmap);
EFilters::RemoveNoise(zmap, filteredZmap, ENoiseRemovalMethod_HighStandardDeviation, 3, 30.0, 0.0);
```

EStatistics class

The `EStatistics` class contains static methods used to compute statistics on ZMaps or depth maps.

ComputeAverageMap

The `ComputeAverageMap()` method computes the local average map.

- You can use this method as a low-pass filter.
- Undefined points are not taken into account.
- This method is based on a square moving kernel. The size of the kernel is (2 x `halfKernelSize` + 1) where `halfKernelSize` is a parameter of the method.

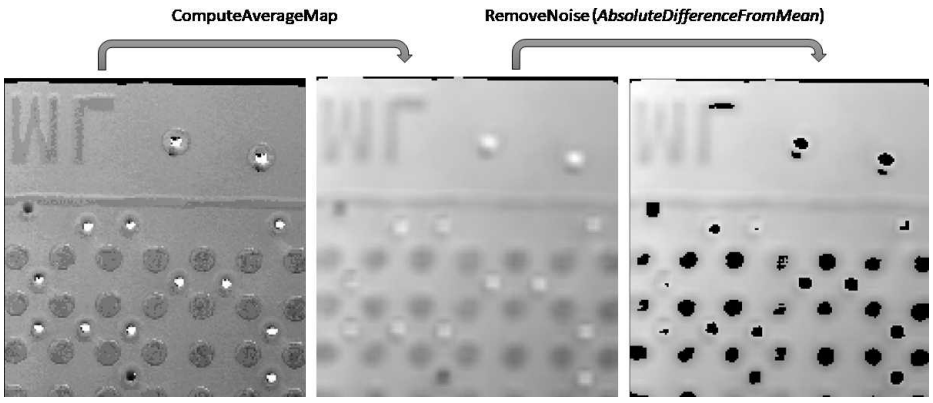
ComputeStandardDeviationMap

The `ComputeStandardDeviationMap()` method computes a map of the local standard deviation.

- You can use this method to determine visually the threshold value to use with the `RemoveNoise()` method when using the `ENoiseRemovalMethod_HighStandardDeviation` setting.

  Note: *Be aware, however, that in the generated map, a pixel with the value 0 can either be undefined or have a standard deviation equal to zero.*

Example: Using a low pass filter on a ZMap, then removing points showing a deviation larger than a defined threshold



The code below first applies an low pass filter, then removes from the result the pixels showing a deviation from the neighborhood larger than the defined threshold.

```
// Load the ZMap data
EZMap16 zmap;
zmap.Load(...);
// Compute the filtered ZMap. The new ZMap is called averagedZMap
// The size of the kernel is 7x7, the threshold is 30.0
EZMap16 averagedZMap;
averagedZMap.SetSize(zmap);
EStatistics::ComputeAverageMap(zmap, averagedZMap, 3, 0.2);
// Compute the filtered ZMap. From averagedZMap, compute filteredZMap
// The size of the kernel is 31x31, the threshold is 20.0
EZMap16 filteredZMap;
filteredZMap.SetSize(zmap);
EFilters::RemoveNoise(averagedZMap, filteredZMap, ENoiseRemovalMethod_AbsoluteDifferenceFromMean,
15, 20.0, 0.2);
```

ComputePixelStatistics

The `ComputePixelStatistics()` method returns basic statistical information about pixel values:

- Minimum
- Maximum
- Average
- Standard deviation
- Number of valid (not undefined) pixels).

Use an `ERegion` object to specify the region of the ZMap or depth map used to compute the statistics.

ComputeStatistics

The `ComputeStatistics()` method returns the same information as the `ComputePixelStatistics()` method, but scaled with respect of the Z resolution.

Use an `ERegion` object to specify the region of the ZMap or depth map used to compute the statistics.

3D Viewer

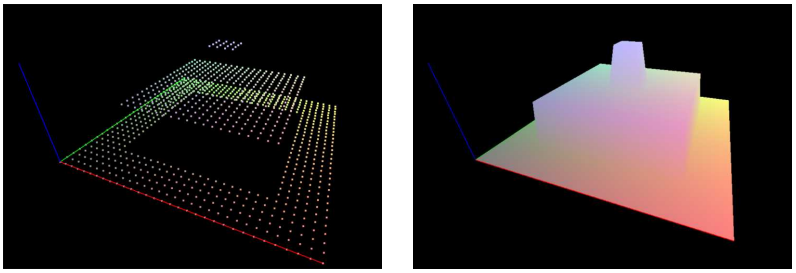
Use the `E3DViewer` class to easily create an interactive 3D display. The viewer displays point clouds, meshes and ZMaps.

You can create `E3DViewer` as a child of an existing window or without a parent. In that last case, a new window is created.

Note: *As `E3DViewer` uses OpenGL interface, it requires a compatible display device.*

Call the `ConfigureRenderSource()` method with a valid 3D geometry to display it. At each call, `ConfigureRenderSource()` replaces the current displayed object. The supported classes are `EPointCloud`, `EMesh` and `EZMap8`/`EZMap16`.

When you configure a new render source with `ConfigureRenderSource()`, the view point is automatically adapted to display the whole object.

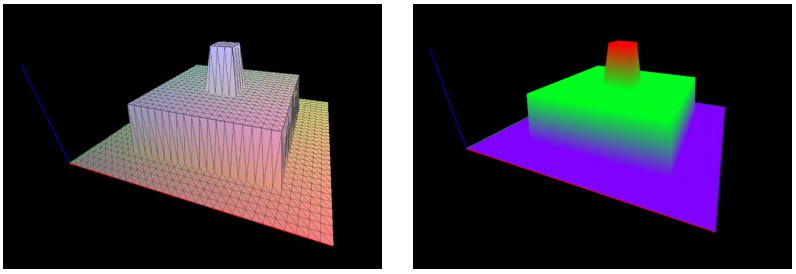


`E3DViewer` in action: point cloud display (left) and 3D object display (right)

To display the geometry in false colors:

- Use the `GenerateColors()` method that computes RGB colors from the position of the vertices.
- It supports various predefined color ramps.
- Use the `SetColors()` method to use custom colors (one `EC24` entry is requested for each render source vertex).

Use the methods `SetPointSize()`, `SetWireframeMode()` and `SetRenderDecimationLevel()` to adjust the rendering attributes.



`E3DViewer` in action: wire frame enable (left) and `HueFromZ` color ramp (right)

In the 3D navigation window, use the mouse as follows:

- Press the left button to rotate the image horizontally and vertically.
- Press the right button to translate the image horizontally and vertically.
- Use the wheel to zoom in and out.

In addition, use the following keys:

- Press **R** to reset the viewer.
- Press **W** to show or hide the triangle edges (in wire frame mode).
- Press **+** and **–** to increase or decrease the point size.

Use methods `SetViewTarget()`, `SetViewingAngle()` and `SetViewingDistance()` to change the view point programmatically.

4. Code Snippets

4.1. Basic Types

Loading and Saving Images

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

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

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

```
// ...
```

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

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

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

Interfacing Third-Party Images

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

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

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

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

```
// ...
```

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

Retrieving Pixel Values

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

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

```
using System.RunTime.InteropServices;
```

```
IntPtr pixAddr;
byte pix;
```

```
//...
```

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

ROI Placement

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

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

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

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

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

Vector Management

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

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

```
// Clear the vector
ramp.Empty();
```

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

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

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

Exception Management

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

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

```
    // ...
```

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

```
catch(EException exc)
{
```

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