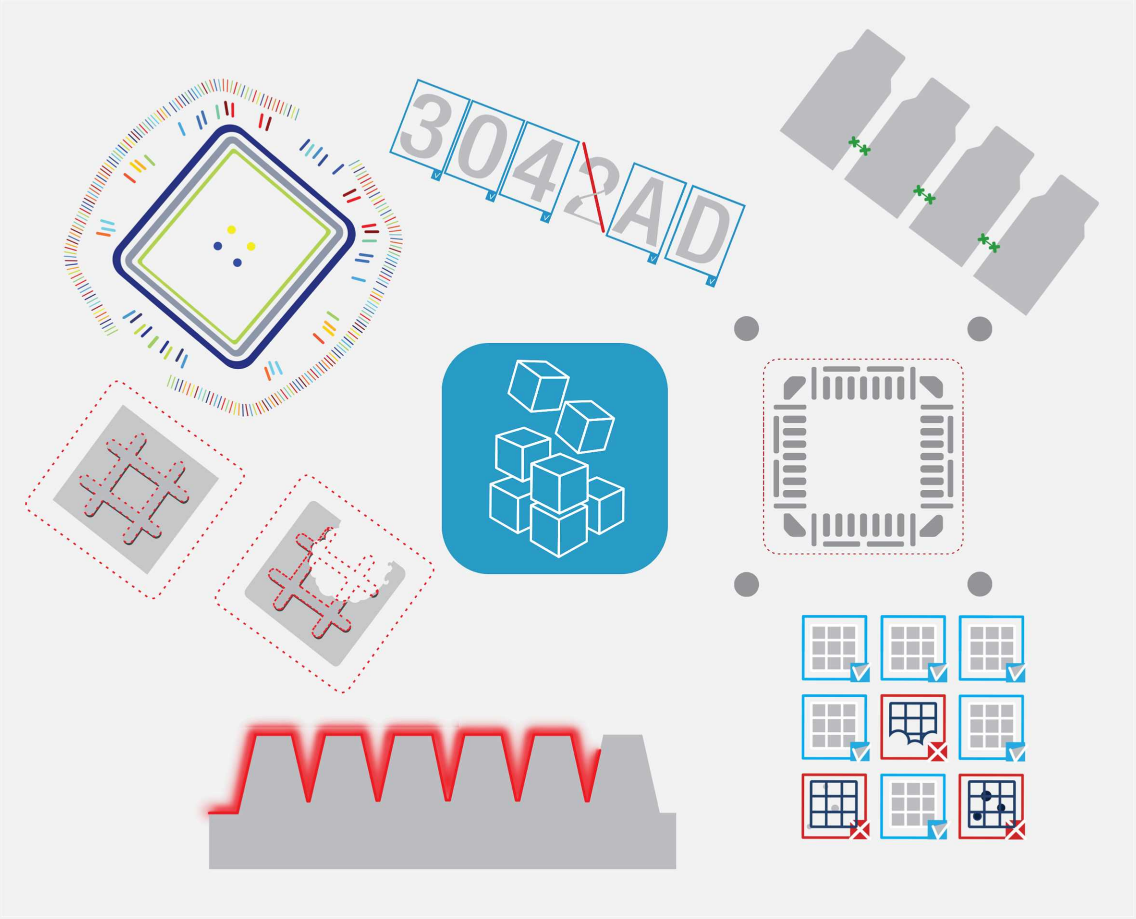
USER GUIDE

Open eVision

3D Processing Tools



Terms of Use

This documentation is provided with Open eVision 2.11.1 (doc build 1125).
© 2019 EURESYS s.a.

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.

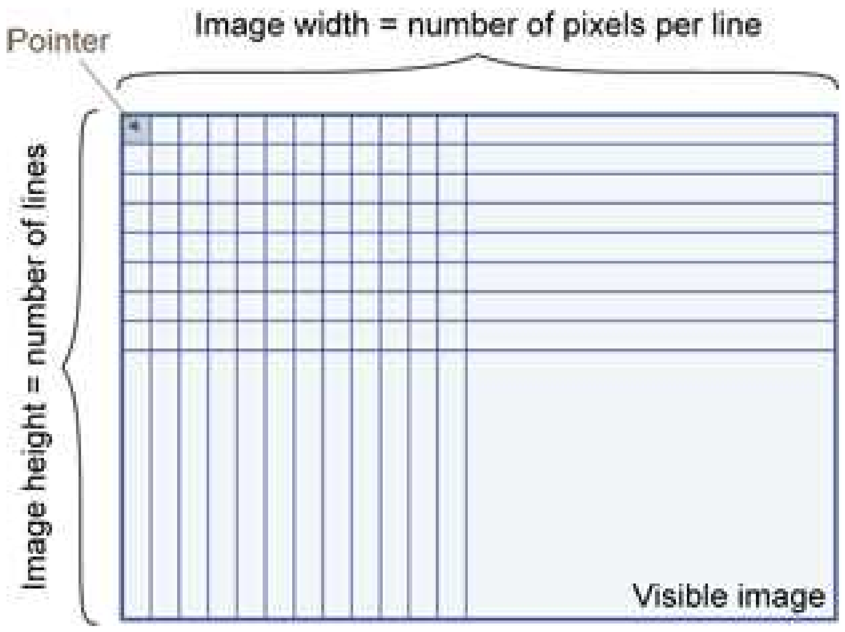

Pointer
Image width = number of pixels per line
Image height = number of lines
Visible image

Image main parameters

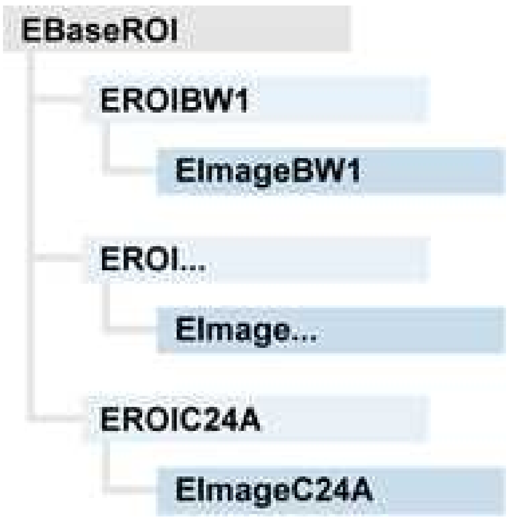
An Open eVision image object has a rectangular array of pixels characterized by `EBaseROI` parameters .

- `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.



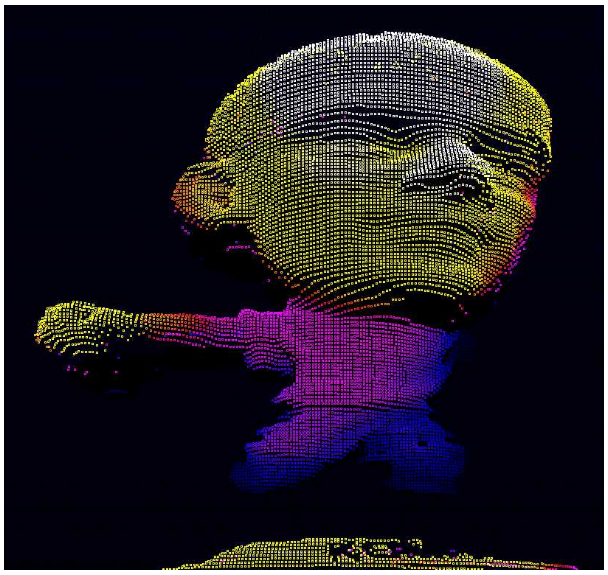
Depth maps

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.

TIP
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

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

image file type arguments

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

(*) Default value.

Assigned image file type if argument is ImageFileType_Auto or missing

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

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

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

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

JPEG compression values

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

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

Representative JPEG 2000 compression quality values

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

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

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

Point Clouds

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

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

2.2. Pixel Container File Load

Images and Depth Maps

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

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

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

Point Clouds

- Use the `Load` method to save the point cloud in Open eVision proprietary file format.

- Use the `LoadPCD` method to save the point cloud in a ASCII or a binary file compatible with other software such as PCL (Point Cloud Library).

2.3. Memory Allocation

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

Internal Memory Allocation

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

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

External Memory Allocation

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

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

2.4. Image and Depth Map Buffer

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

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

[1]device-independent bitmap

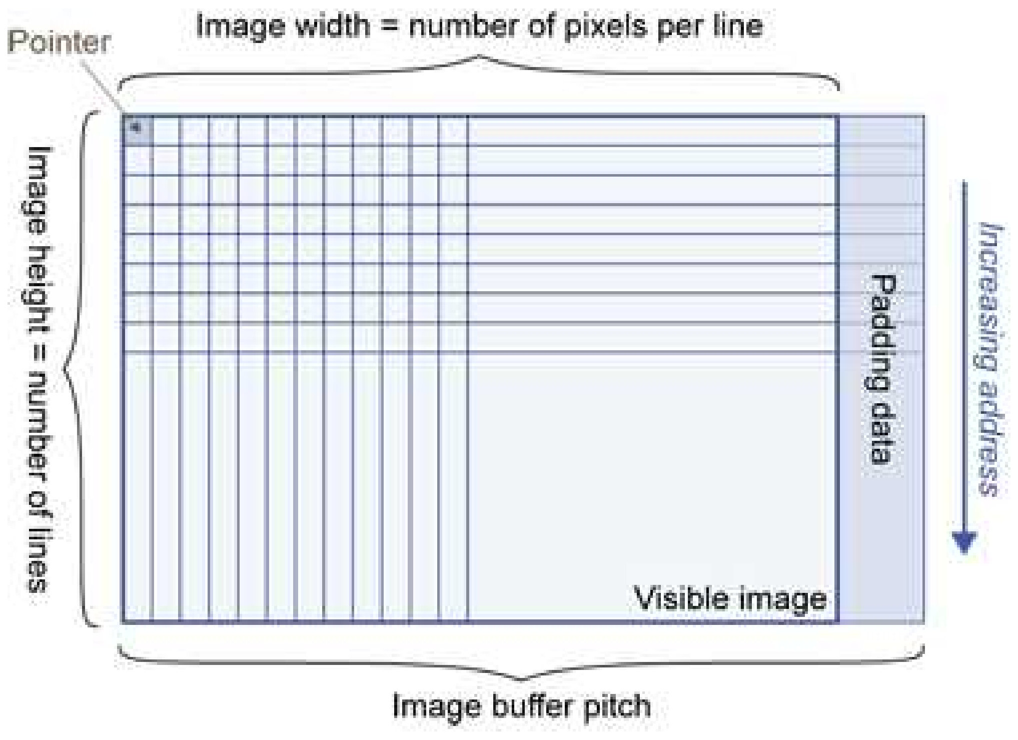

Pointer
Image width = number of pixels per line
Image height = number of lines
Padding data
Increasing address
Visible image
Image buffer pitch

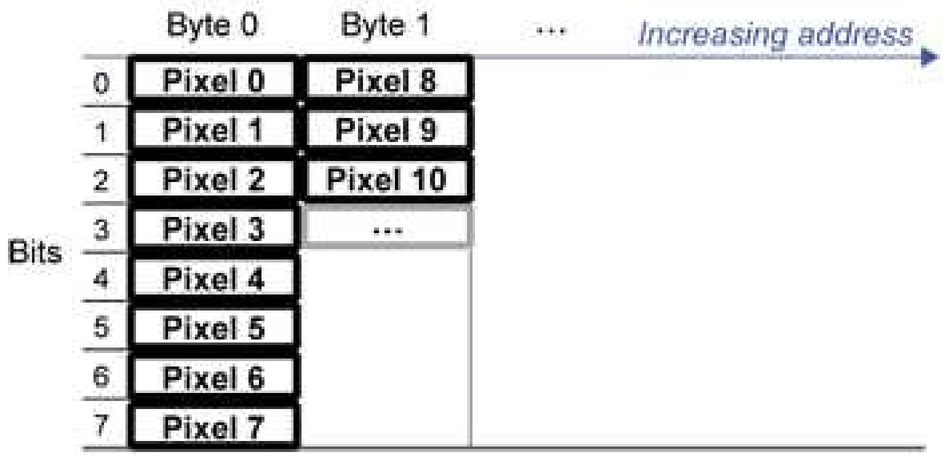
Image Buffer pitch

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

Memory Layout

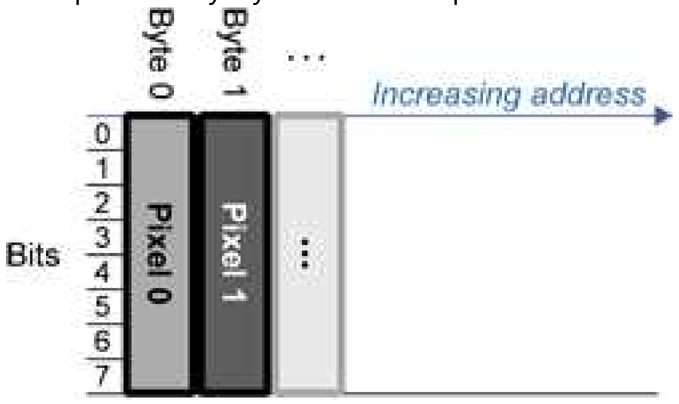
- `EImageBW1` stores 8 pixels in one byte.

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



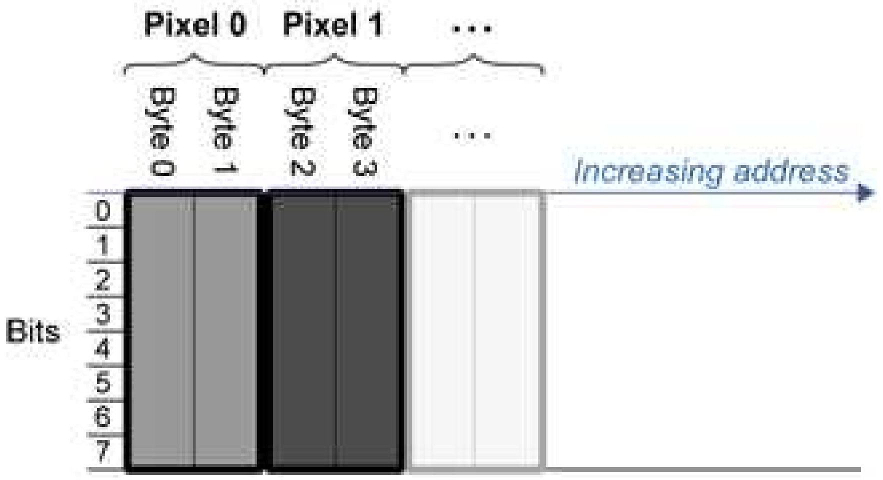
- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

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



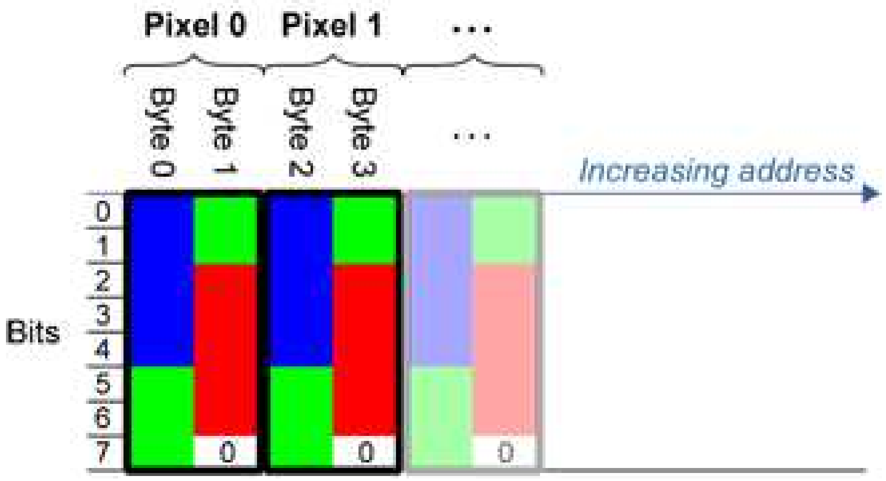
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).

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



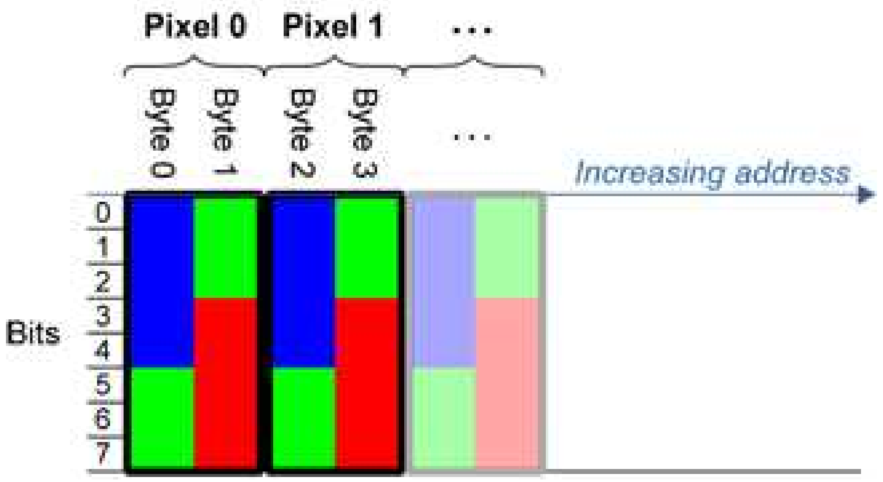
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits.
  The 16th bit is left unused.

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



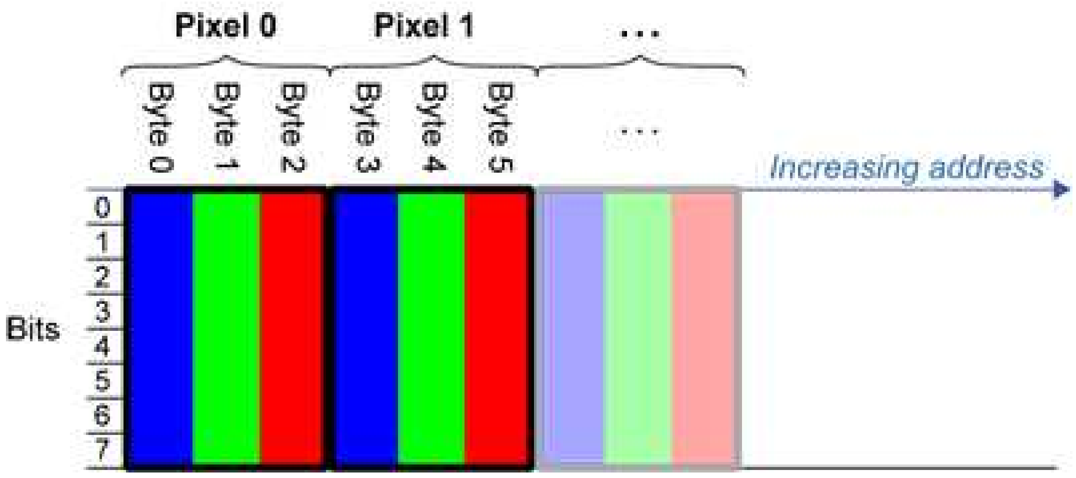
- `EImageC16` stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
  The second color component is coded with 6-bits.

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



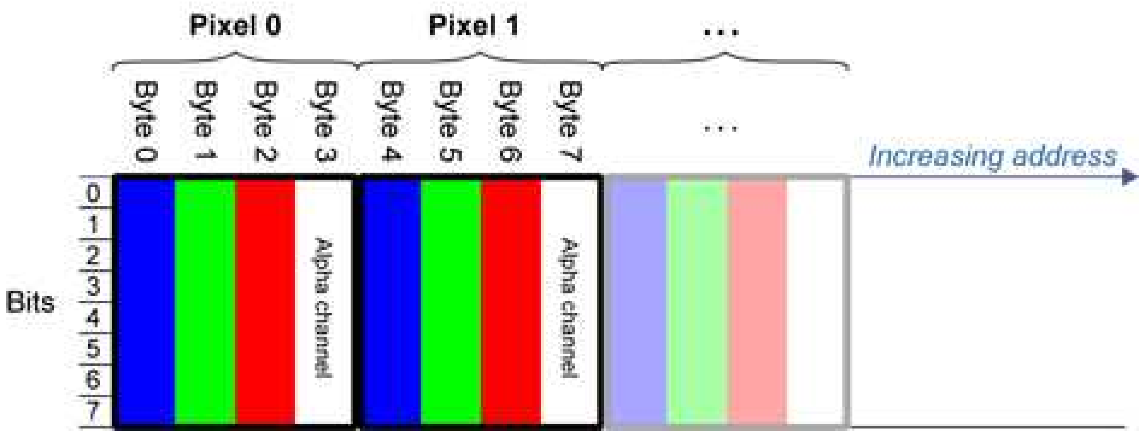
- `EDepthMap16` store each pixel in 2 bytes using a fixed point format.
- `EImageC24` stores each pixel in 3 bytes. Each color component is coded with 8-bits.

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



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

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



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

2.5. Image Drawing and Overlay

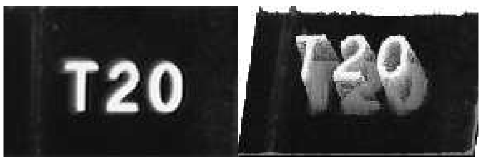
- Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.
- The color palette in 256-color display mode gives optimal rendering. Gray-level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

2.6. 3D Rendering of 2D Images

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

Gray 3D Rendering

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



3D rendering

Color Histogram 3D Rendering

`Easy::RenderColorHistogram` prepares a 3-dimensional rendering of a color image histogram. The pixels are drawn in the RGB space (not XY-plane) to show clustering and dispersion of RGB

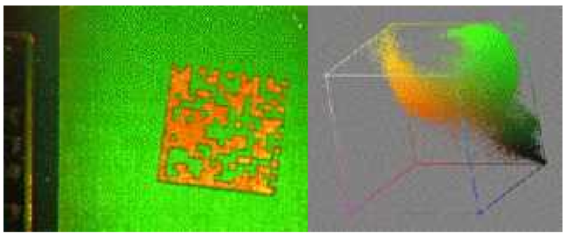
[1]Graphics Device Interface

[2]Microsoft Foundation Class

[3]LookUp Tables

values.
This function can process pixels in other color systems (using EasyColor to convert), but the raw RGB image is required to display the pixels in their usual colors.

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

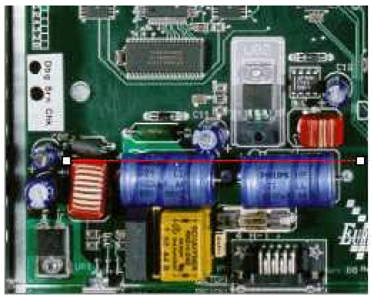


Color histogram rendering

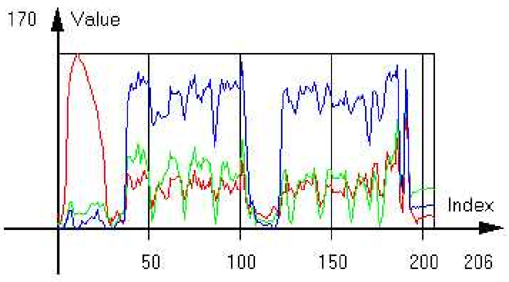
2.7. Vector Types and Main Properties

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

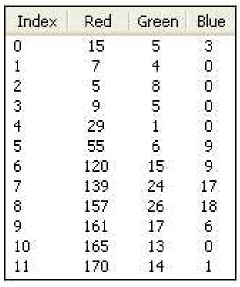
EVector is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.



Profile in a C24 image



RGB values plot along profile



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

RGB values array (EC24Vector)

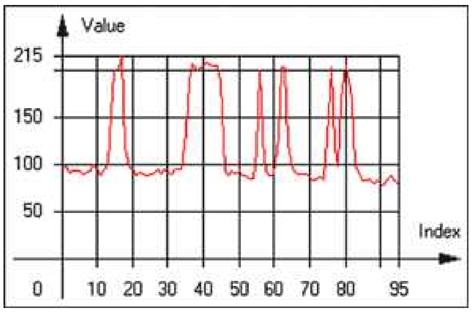
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

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

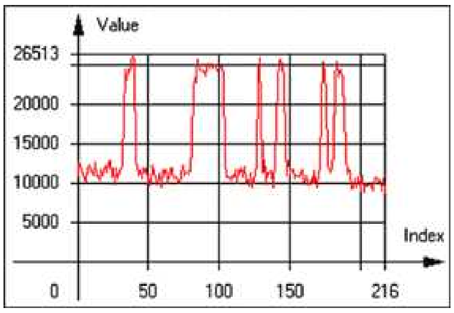
Vector types

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



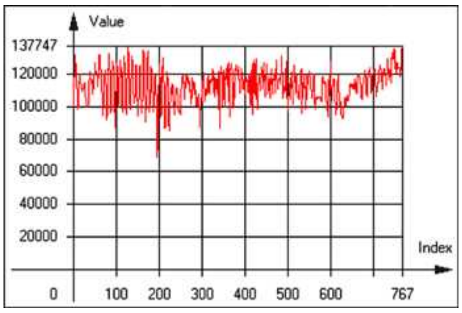
Graphical representation of an `EBW8Vector` (see `Draw` method)

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



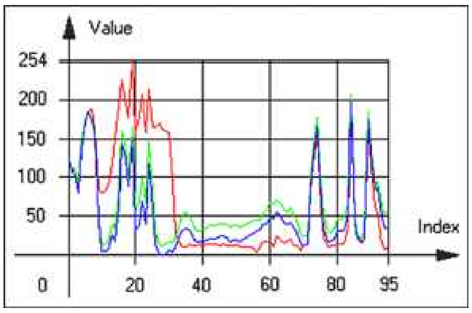
Graphical representation of an `EBW16Vector`

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



Graphical representation of an `EBW32Vector`

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



Graphical representation of an `EC24Vector`

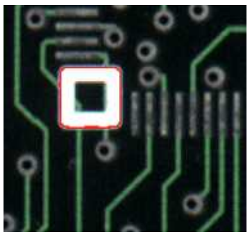
- `EBW8PathVector`: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
(used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).

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

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

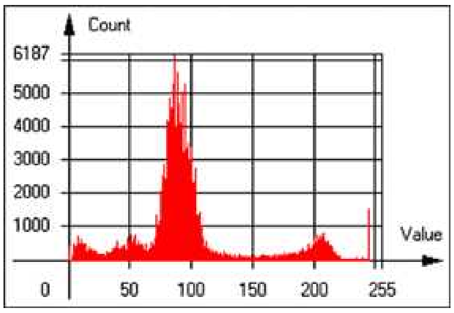
Graphical representation of an `EBW16PathVector` **(see** `Draw` **method)**

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



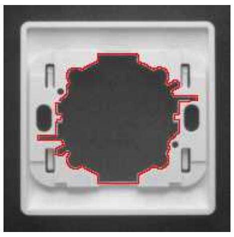
Graphical representation of an `EC24PathVector` **(see** `Draw` **method)**

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



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

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



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

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

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

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

2.8. ROI Main Properties

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

Save and load

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

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

ROI Classes

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

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

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

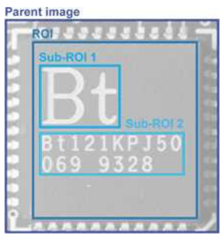
Attachment

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

Nesting

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

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



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

Cropping

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

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

Resizing and moving

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

2.9. Arbitrarily Shaped ROI (ERegion)

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

Regions or arbitrarily shaped ROI

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

In Open eVision:

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

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

Library	Method
EasyImage	`EasyImage::Threshold`
	`EasyImage::DoubleThreshold`
	`EasyImage::Histogram`
	`EasyImage::Area`
	`EasyImage::AreaDoubleThreshold`
	`EasyImage::BinaryMoments`
	`EasyImage::WeightedMoments`
	`EasyImage::GravityCenter`
	`EasyImage::PixelCount`
	`EasyImage::PixelMax`
	`EasyImage::PixelMin`
	`EasyImage::PixelAverage`
	`EasyImage::PixelStat`
	`EasyImage::PixelVariance`
	`EasyImage::PixelStdDev`
	`EasyImage::PixelCompare`
Easy3D	`EDepthMapToMeshConverter::Convert`
	`EDepthMapToPointCloudConverter::Convert`
	`EStatistics::ComputePixelStatistics`
	`EStatistics::ComputeStatistics`
	`E3DObjectExtractor::Extract`
	`EZMapToPointCloudConverter::Convert`
EasyObject	`EImageEncoder::Encode`
EasyFind	`EPatternFinder::Find`
	`EPatternFinder::Learn`
EasyOCR2	`EOCR2::Read`
	`EOCR2::Detect`

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

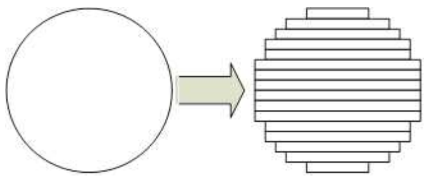
TIP
In the future Open eVision releases, the support of `ERegions` will be gradually extended to all operators.

Creating regions

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

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

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



Run-Length Encoding of a circle-shaped region

To create a region, either:

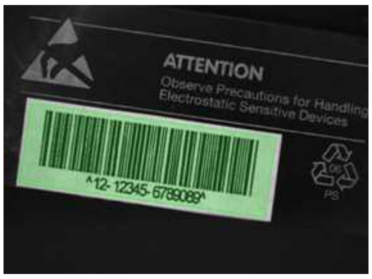
- Use one of the geometry-based region classes.
- Use the result of another tool, such as EasyFind, EasyMatch or EasyObject.
- Combine or modify other regions.
- Use a mask image.
- Directly provide the list of runs.

Geometry-based regions

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

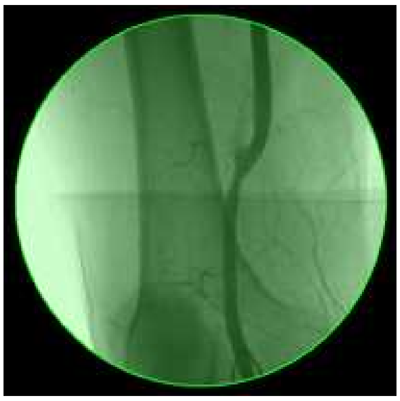
Use these classes to setup geometric regions and modify them with translation, rotation and scaling. The transformation operators return new regions, leaving the source object unchanged.

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


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

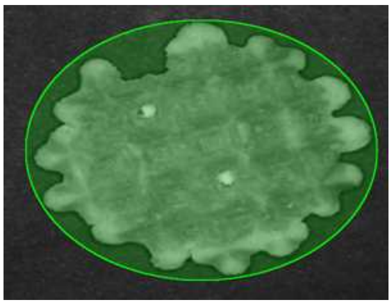
Rectangle region separating a bar code from the background

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



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

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



Ellipse region encompassing a waffle

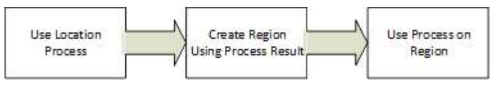
- EPolygonRegion
  - The contour of an EPolygonRegion class is a polygon.
  - It is constructed using the list of its vertices.

Polygon region encompassing a key

Using the result of other tools

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

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



Open eVision provides constructors for the following tools:

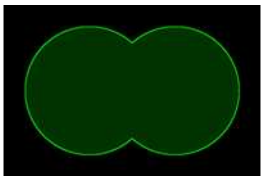
- EasyFind: `EFoundPattern`
- EasyMatch: `EMatchPosition`
- EasyGauge: `ECircle` and `ERectangle`
- EasyObject: `ECodedElement`

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

Combining regions

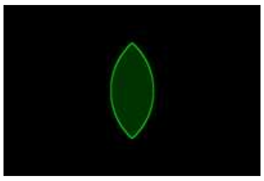
Use the following operations to create a new region by combining existing regions:

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



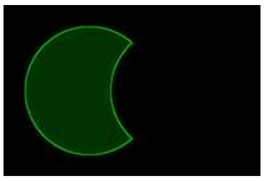
Union of 2 circles

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



Intersection of 2 circles

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



Subtraction of 2 circles

Using regions

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

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

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

Preparing the region

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

Drawing regions

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

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

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

ERegions and EROIs

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

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

ERegion and 3D

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

2.10. Flexible Masks

ROIs vs flexible masks

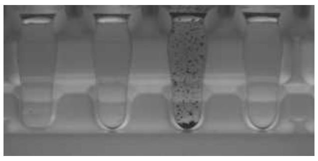
ROIs and masks restrict processing to part of an image:

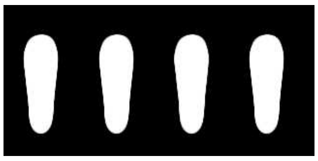
- "ROI Main Properties" on page 24 apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
- Flexible Masks are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some EasyObject and EasyImage library functions.

Flexible Masks

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

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

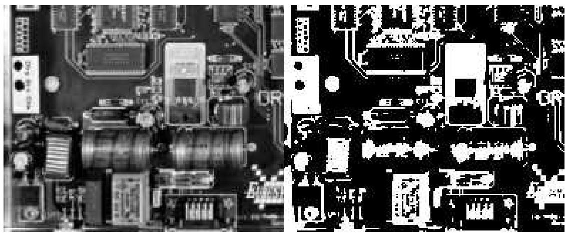

Source image


Associated mask

Processed masked image

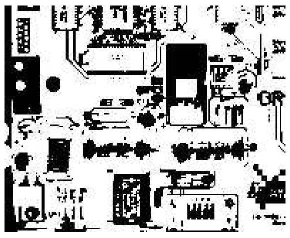
A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.

Flexible Masks in EasyImage


Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

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


Resulting image

EasyImage Functions that support flexible masks

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

- ImageToLineSegment, ImageToPath (Vector profile).

Flexible Masks in EasyObject

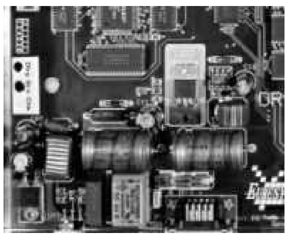
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

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

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

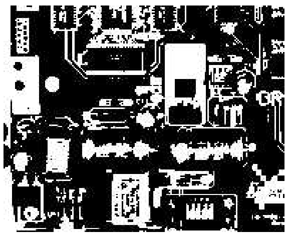
EasyObject functions that create flexible masks



Source image

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

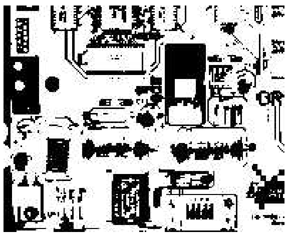
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers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



BW8 resulting image that can be used as a flexible mask

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

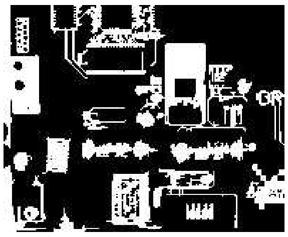
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

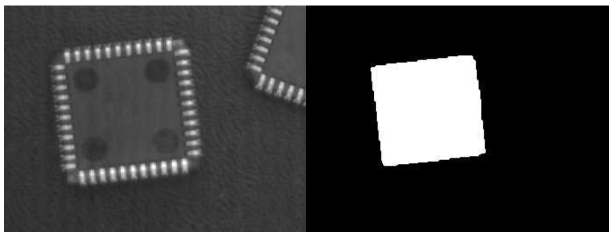
3) EObjectSelection::RenderMask: from a selection of blobs

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



BW8 resulting image that can be used as a flexible mask

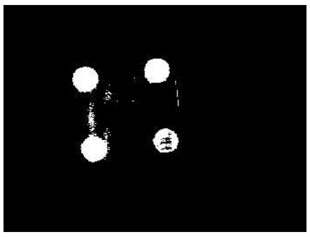
Example: Restrict the areas encoded by EasyObject



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

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

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



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

2.11. Profile

Profile Sampling

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

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

Profile Analysis

The profile can be processed to find peaks or transitions:

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

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

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

Profile Insertion Into an Image

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

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

3. 3D Tools

3.1. Easy3D - Using 3D Toolset

Basic Concepts

Easy3D

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

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

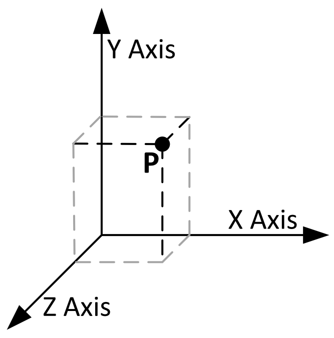
Easy3D provides a calibration tool to generate corrected, metric point clouds and meshes from depth maps. Most 3D operators work on point clouds or meshes. The included export functions to the standard PCD file format allows integration with other 3D tools.

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

All the Easy3D tools are placed in the Easy3D namespace.

3D representation

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



Open eVision provides different containers to store 3D objects :

- Depth maps
- Point clouds
- Meshes
- ZMaps

Depth map

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

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

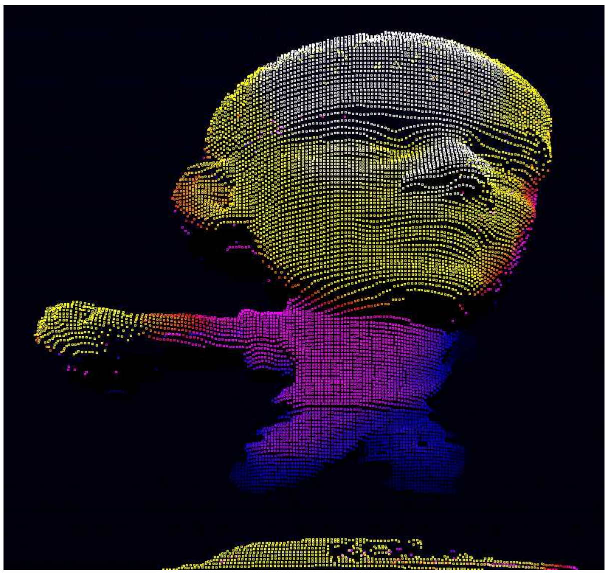
A depth map contains a gray scale image coded on 8, 16 or 32 bits per pixel.

- One specific gray value, called the undefined value, is reserved for the representation of invalid pixels.
- By default, this value is 0 for integer depth map types (`EDepthMap8` and `EDepthMap16`).
- By default, this value is the lowest float value (-3.402823 e+38) for the 32 bits floating point depth map types (`EDepthMap32f`).

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

Point cloud

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



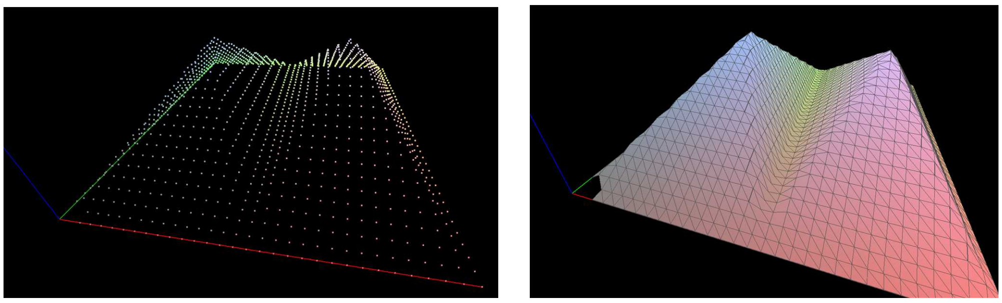
In addition to the calibration process included in Easy3D, point clouds can be produced using various 3D acquisition techniques, like stereo reconstruction or time of flight cameras.

Mesh

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

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

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



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

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

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

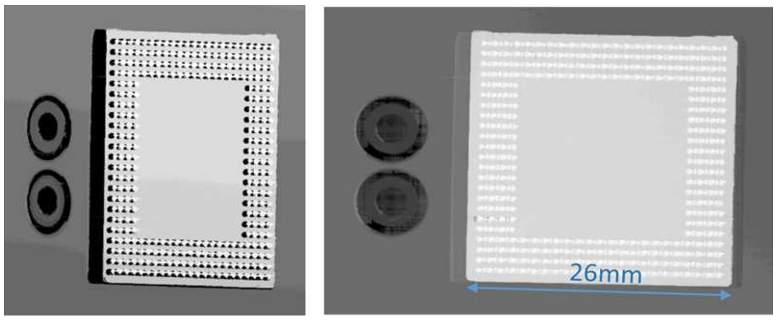
ZMap

ZMaps are another representation for 3D data.

- They are grayscale images like depth maps but represent metric and corrected 3D points.
- They are convenient representations for measurement and matching.

- They are compatible with most of the 2D processing functions.

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



A depth map and the corresponding ZMap

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

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

A ZMap also contains the following information:

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

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

Static Methods

EFilters class

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

RemoveNoise

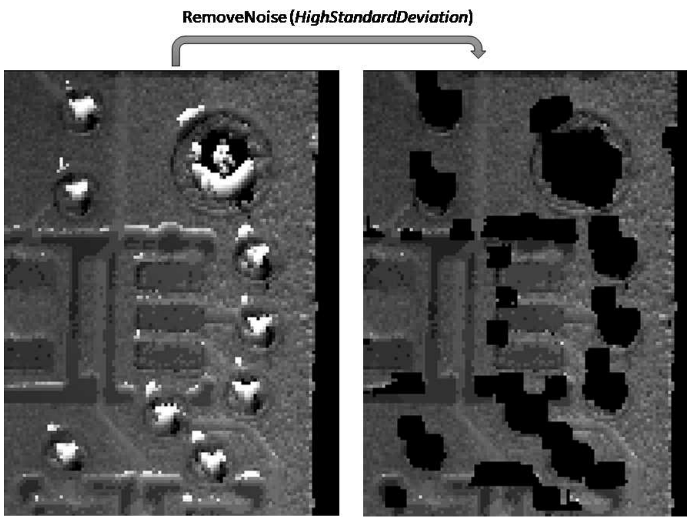
The `RemoveNoise()` method removes outliers from a depth map or a ZMap.

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

  The undefined points are not taken into account.
- It is based on a square moving kernel. The size of the kernel is (2 x `halfKernelSize` + 1) where `halfKernelSize` is a parameter of the method.

- The threshold parameter is scaled with regard to the Z resolution of the filtered depth map or ZMap.
- There are 3 variations of this filter, depending on the method parameter:
  - ENoiseRemovalMethod_AbsoluteDifferenceFromMean removes a point when it deviates from the average in the neighborhood, including itself. The threshold is an absolute difference.
  - ENoiseRemovalMethod_RelativeDifferenceFromMean removes a point when it deviates from the average in the neighborhood, including itself. The threshold is a multiple of the standard deviation.
  - ENoiseRemovalMethod_HighStandardDeviation removes a point when the standard deviation in the neighborhood, including itself, is higher than a defined threshold.

Example: Removing points showing a high standard deviation



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

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

EStatistics class

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

ComputeAverageMap

The ComputeAverageMap() method computes the local average map.

- You can use this method as a low-pass filter.
- Undefined points are not taken into account.

- This method is based on a square moving kernel. The size of the kernel is (2 x `halfKernelSize` + 1) where `halfKernelSize` is a parameter of the method.

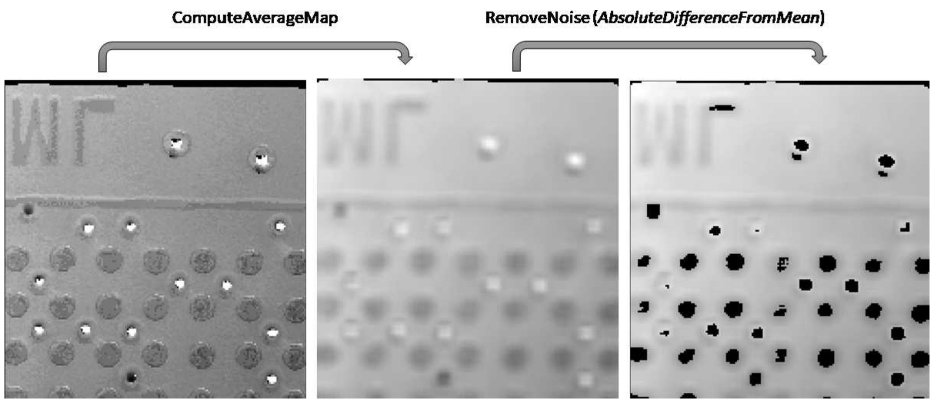
ComputeStandardDeviationMap

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

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

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

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



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

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

ComputePixelStatistics

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

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

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

ComputeStatistics

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

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

Point Cloud

Coordinates Transformations

Affine Transforms

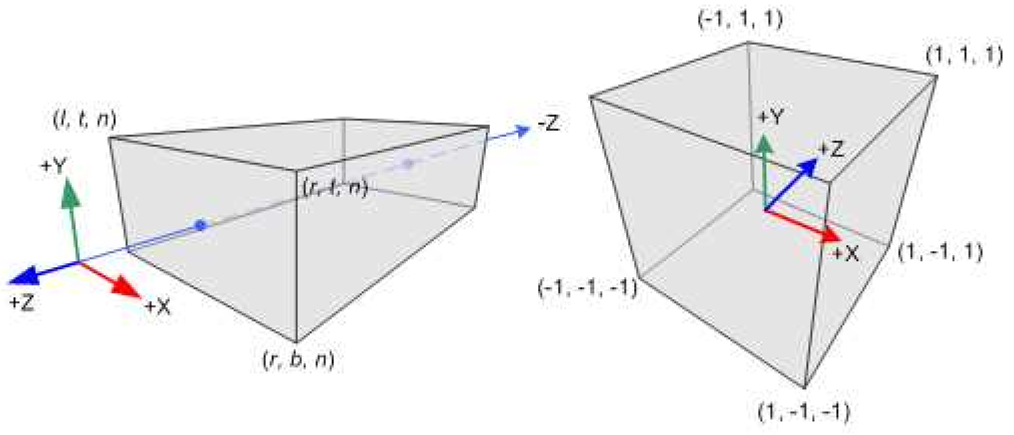
Affine transforms allow you to reposition the point cloud inside the 3D space.

Open eVision provides you with the following basic transformations:

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

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

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



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

Reducing a Point Cloud

Cropping

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

Open eVision provides the following cropping functions:

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

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

Decimation

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

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

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

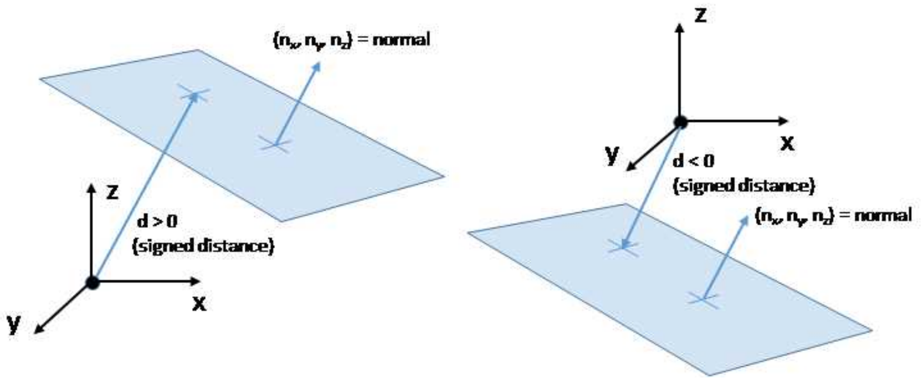
Managing Planes

E3DPlane

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

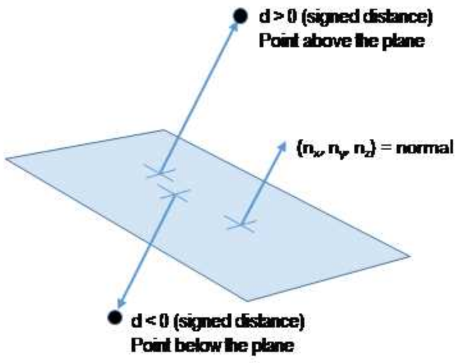
This plane is characterized by:

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



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

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



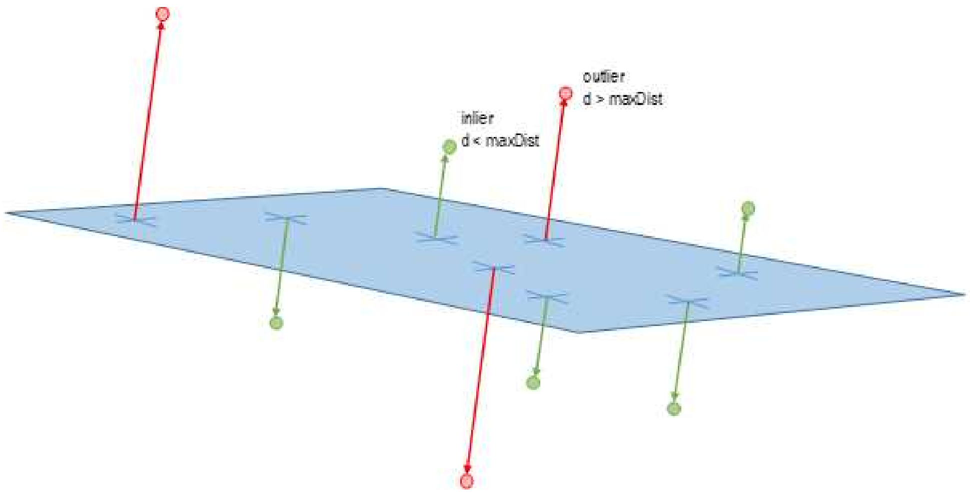
EPlaneFinder

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

The main parameters of this object are:

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

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

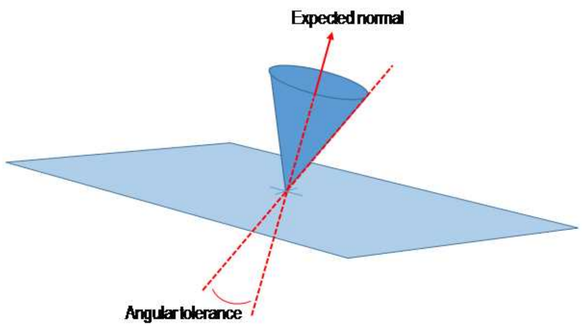


A `EPlaneFinder` object produces a `E3DPlane` object. The algorithm searches for a plane containing as many inliers as possible. This plane is the biggest plane if the samples are evenly distributed.

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

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

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



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

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

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

EPlaneFitter

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

Aligning

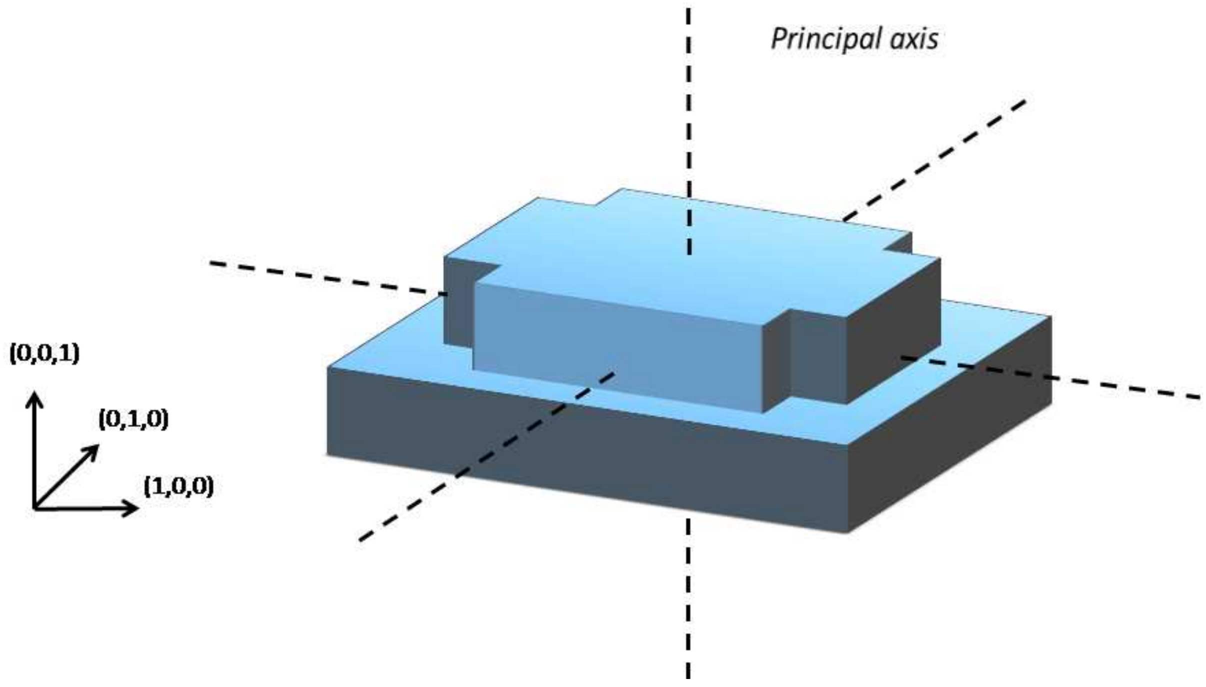
EPrincipalAxisExtractor

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

This new orthogonal basis has the following characteristics:

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

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



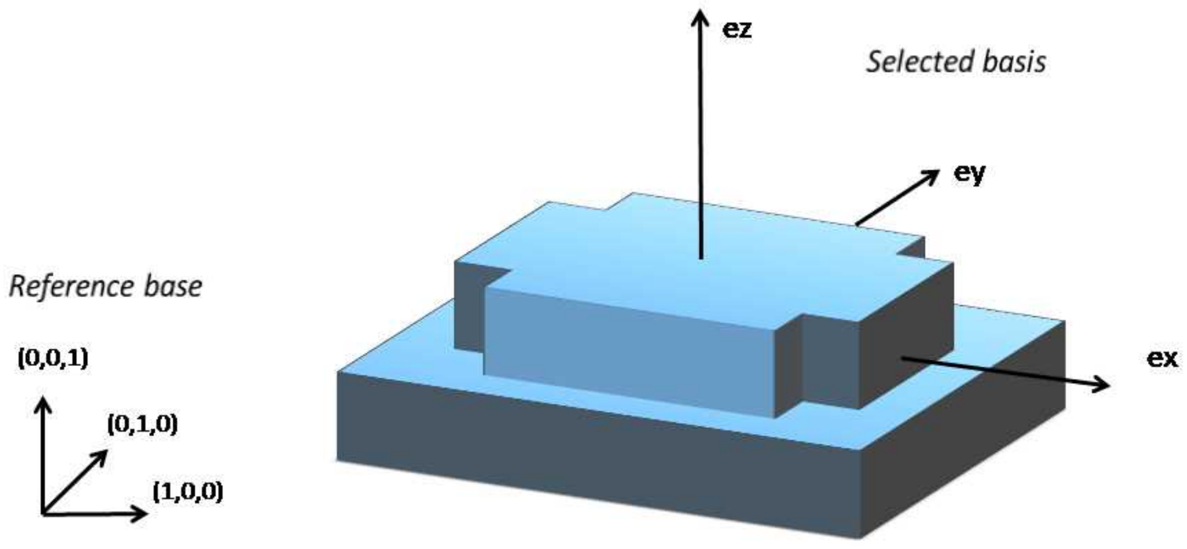
The principal axis extraction is done using the `Extract()` method that takes a `EPointCloud` as input and returns an `E3DTransformMatrix`. Optionally, you can pass 3 other output parameters by reference to retrieve the value of the standard deviation along the 3 principal axis.

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

Specification of a reference transformation

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

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



EFeaturesAligner

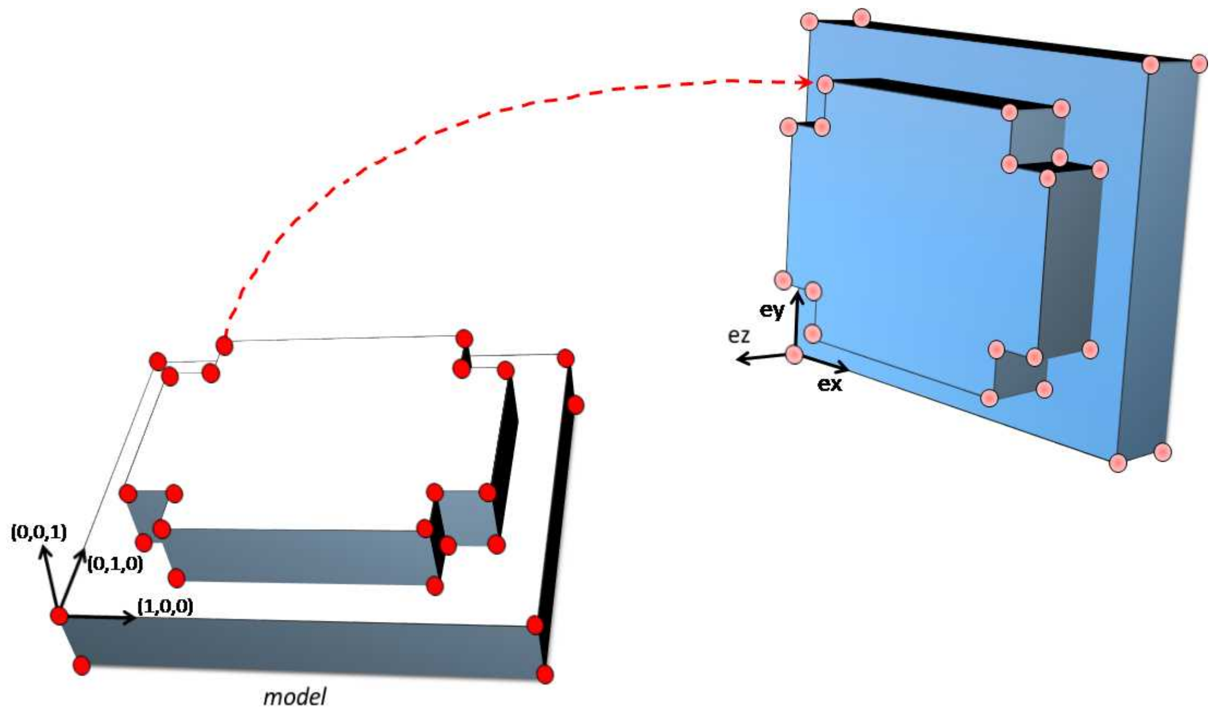
A `EFeaturesAligner` object finds the best transformation that maps a list of points to another list of points.

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

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

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

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



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

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

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

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

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

Mesh

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

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

Generation

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

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

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

Access and usage

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

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

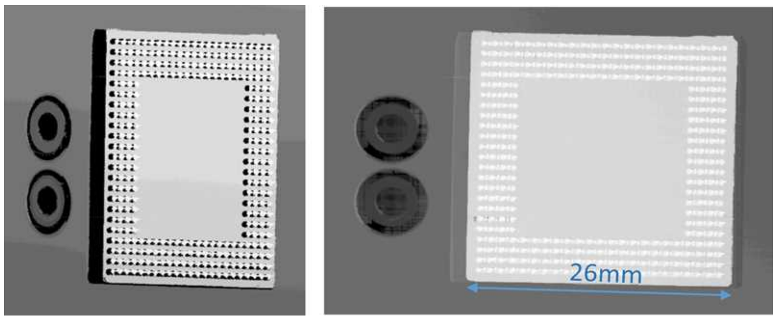
You can use an `EMesh` to produce a ZMap (see "Generating a ZMap" on the next page). Because an `EMesh` represents a surface, the so generated ZMap can show better continuity and less undefined pixels.

ZMap

Generating a ZMap

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

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



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

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

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

Several methods are available to further control the conversion:

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

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

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

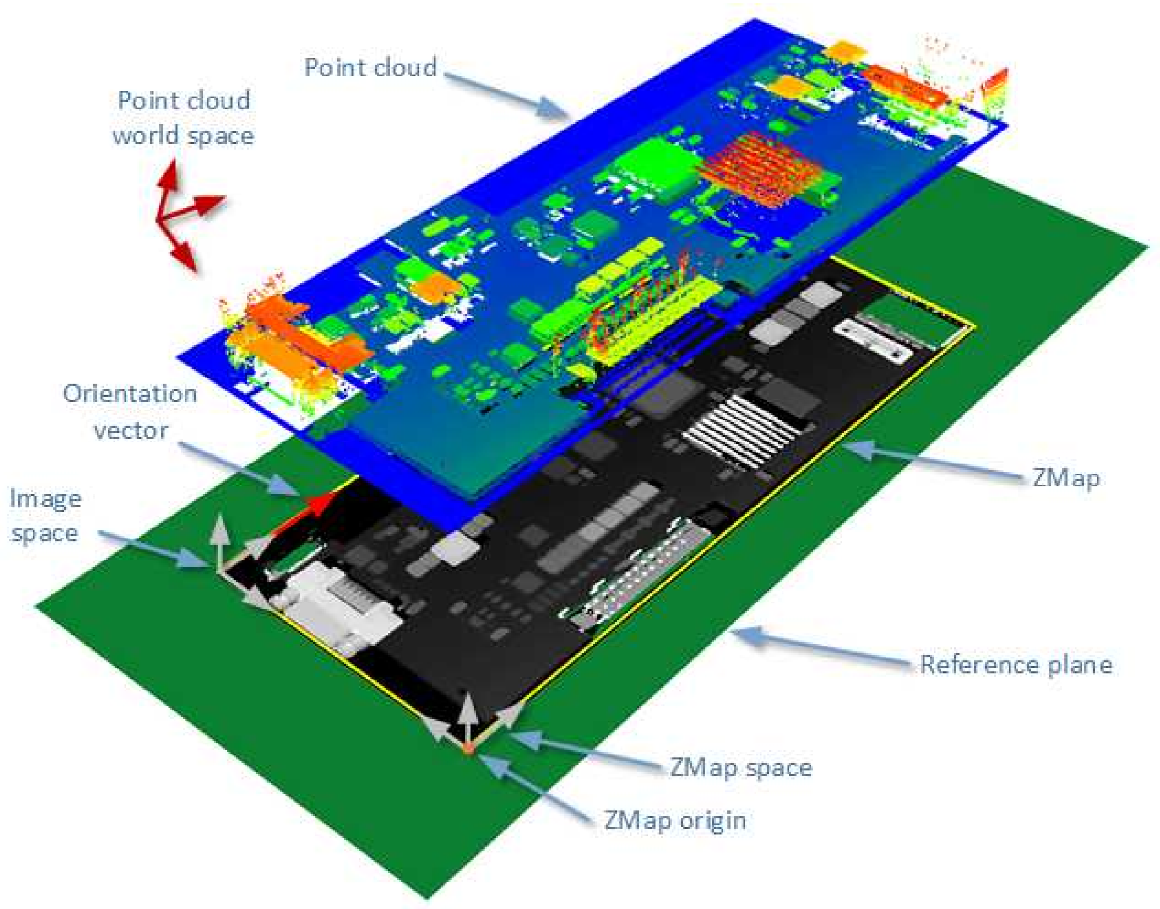
  The orientation vector allows to “rotate” the object around the normal of the reference plane.

- `SetOrigin()` specifies the world position that is on the ZMap lower left pixel (0, 0).
- `SetMapSize()` defines the resolution (number of pixels in X and Y axis) of the generated ZMap.
- `SetMapXYResolution()` adjusts the X and Y resolution of the ZMap pixels, in world space unit per pixel (for example mm/pixel). This value is used to compute the ZMap size (width and height), depending on the projected size of the point cloud on the reference plane.

- SetMapZResolution() sets the Z resolution, in world space unit per pixel unit (gray value). The Z resolution is used to compute the transformation of the distance to the reference plan to the integer 8, 16 or 32 bits pixel value.
- EnableFillMode() and SetFillMode() control the options used to fill the "hole" in the ZMap. A hole exists when no 3D point is projected in the ZMap at a pixel position.

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

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

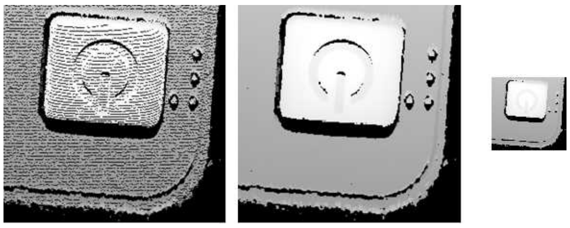


The projection of a point cloud on a ZMap,
showing 3 coordinate systems: the world space, the ZMap space and the pixel space.

the `Convert()` method performs the effective projection of a point cloud (`EPointCloud`) or a 3D object (`EMesh`) to the 8, 16 or 32 bits ZMap.

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

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



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

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

Creating a Point Cloud from a ZMap

To generate a point cloud from a ZMap, use the `EZMapToPointCloudConverter` class.

The `Convert()` method takes:

- A ZMap source
- A `EPointCloud` destination.
- 2 optional parameters:
  - An `ERegion` that defines the domain of the ZMap to convert.
    By default, Open eVision uses all the defined pixels of the ZMap generate the point cloud.
  - A parameter to select the world space (by default) or the ZMap space to store the resulting positions in the point cloud.

Managing the Coordinates

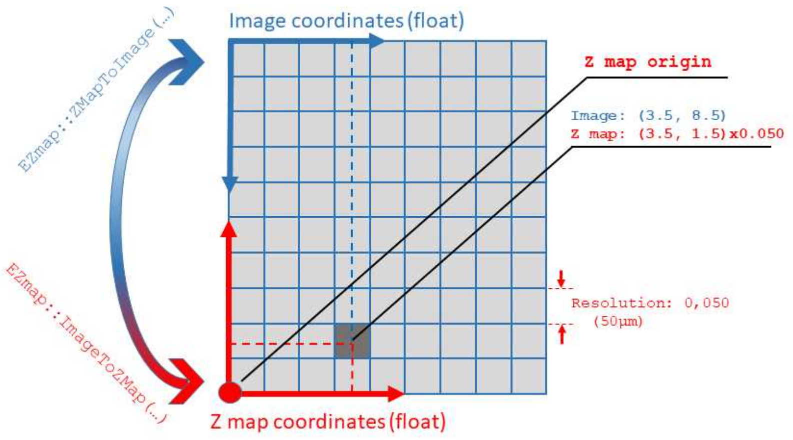
Coordinate systems on a ZMap

A ZMap has multiple coordinate systems:

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

The transformations between:

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



EZMap

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

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

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

The functions that accesses the pixel values are:

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

3D Viewer

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

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

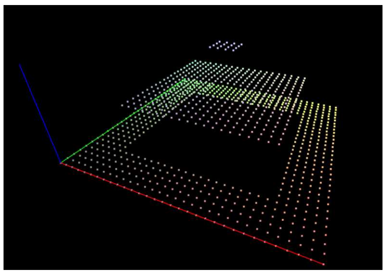
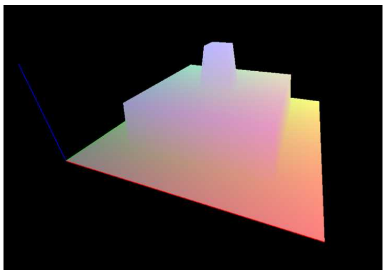
NOTE
As `E3DViewer` uses OpenGL interface, it requires a compatible display device.

Call the `ConfigureRenderSource()` method with a valid 3D geometry to display it. At each call, `ConfigureRenderSource()` replaces the current displayed object.

The supported classes are:

- EPointCloud
- EMesh
- EZMap8, EZMap16 and EZMap32f.

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

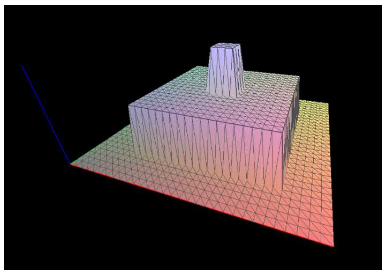
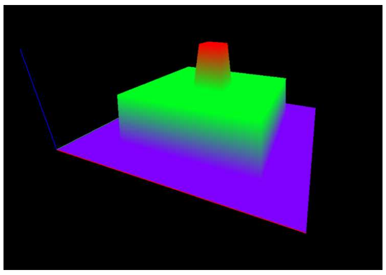



E3DViewer in action: point cloud display (left) and 3D object display (right)

To display the geometry in false colors:

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

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

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

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

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

In addition, use the following keys:

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

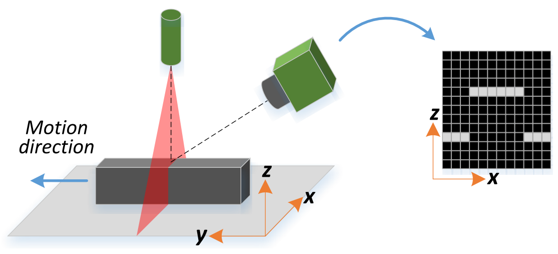
Use methods `SetViewTarget()`, `SetViewingAngle()` and `SetViewDistance()` to change the view point programmatically.

Use the methods `SetAutoRotate()` and `StopAutoRotate()` to manage the automatic rotation of the 3D view.

3.2. Easy3DLaserLine - Laser Line Extraction and Calibration

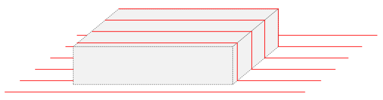
Laser Triangulation

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



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

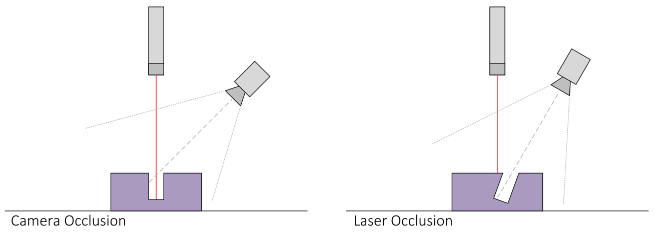
From the scanning you can reconstruct its 3D shape.



Occlusions

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

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



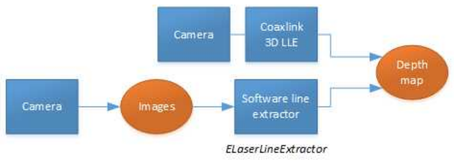
TIP
You can limit or avoid occlusions by using advanced scanning methods, for example by using two cameras or two lasers.

The Laser Line 3D Acquisition Pipeline

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

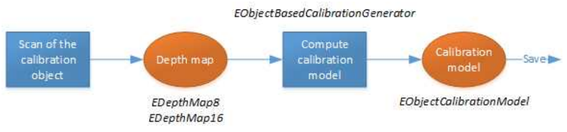
The source material for 3D processing is the depth map, coming from a Coaxlink Quad 3D-LLE or generated from a list of images.

3 types of depth map are available, one for each different pixel coding scheme (8, 16 or 32 bits).



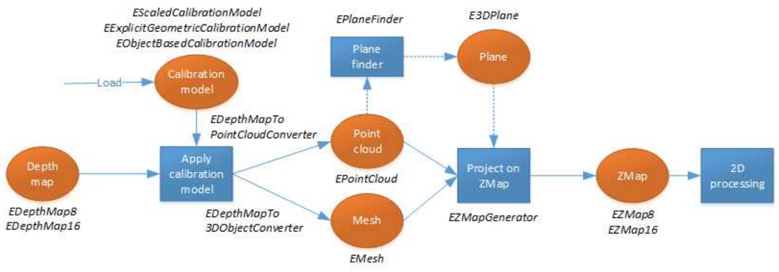
The generation of a depth map, from a hardware or a software source

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


EObjectBasedCalibrationGenerator
Scan of the calibration object
Depth map
Compute calibration model
Calibration model
Save
EDepthMap8
EDepthMap16
EObjectCalibrationModel

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

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



The workflow from the depth map to the ZMap

The following sections describe the classes and methods useful for a 3D workflow. The "Measuring a Remote Controller" on page 98 goes through this processing pipeline.

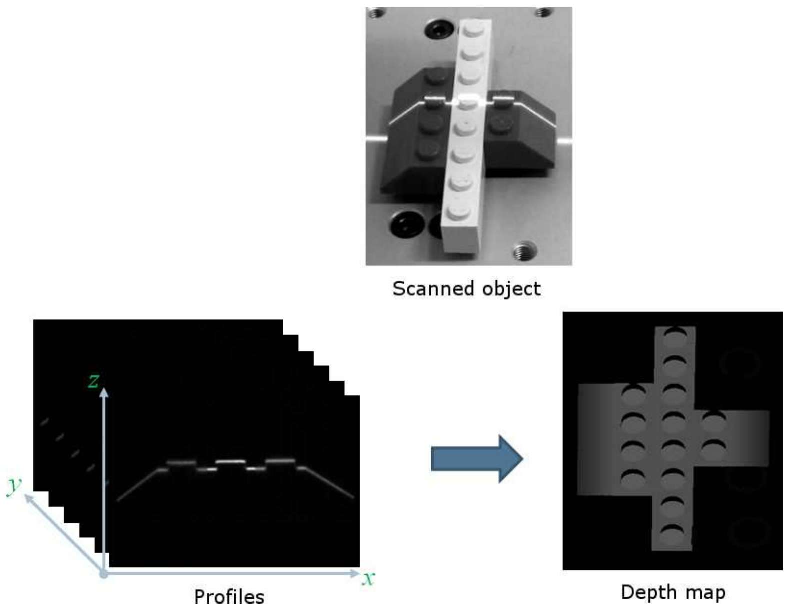
Laser Line Extraction

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

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

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

The figure below illustrates a depth map generation.

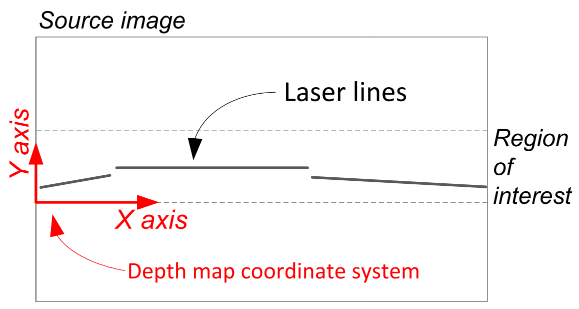


The ELaserLineExtractor class provides the laser line extraction functionality in Open eVision. It implements several algorithms to extract the laser line (see below for more details):

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

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

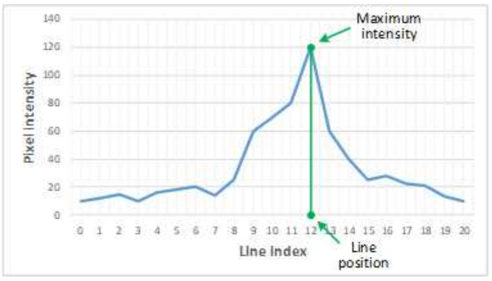
The line position returned by the laser line extraction algorithms is relative to the bottom of the region of interest. So, values in the depth map range from 0 (bottom of the ROI) to the height of the ROI.



Laser line extraction methods

Maximum detection

The maximum detection algorithm analyzes all the pixels in a ROI column to determine the one with the maximum intensity. The figure below shows the laser line position on a given ROI column.



Maximum detection on a ROI profile

We also recommend to include in the processing chain:

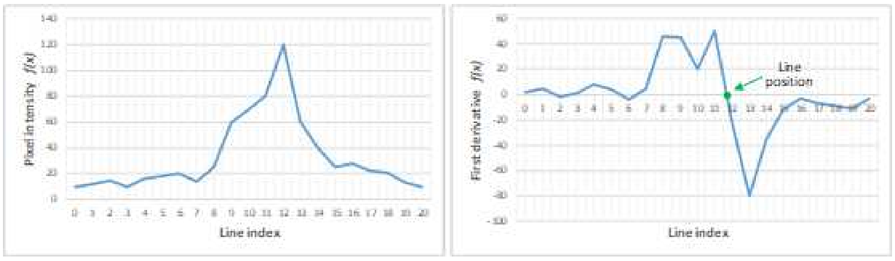
- A low-pass filter to reduce the high frequency variations in the image.
- A threshold to eliminate the background noise from the sensor.

Peak detection

The peak detection algorithm relies on a discrete simplification of the first derivative function.

$$\frac{df}{dx} = f(x+1) - f(x-1) = f'(x)$$

The f '(x) outputs the slope of a given f (x) along the x.



f (x) and f '(x) plots

We compute the line position by detecting where f ’(x) changes its signal based on the two-point form line equation:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

where (x_1, y_1) and (x_2, y_2) are two points on the line with $x_2 \neq x_1$, we obtain the following equation for y = 0:

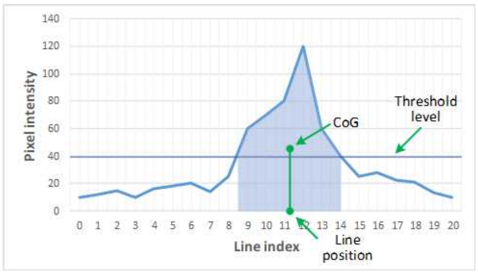
$$x = \frac{x_1 y_2 - x_2 y_1}{y_2 - y_1}$$

Center of gravity

The center of gravity (CoG) method uses an algorithm that calculates the center of mass of an image object. Also know as "centroid of plane figures", the CoG is obtained by the following equations:

$$\overline{X} = \frac{\sum ax}{\sum a} \qquad \overline{Y} = \frac{\sum ay}{\sum a}$$

where $\overline{X}$ and $\overline{Y}$ are the coordinates of the CoG and a is the pixel intensity along the x and y axes.

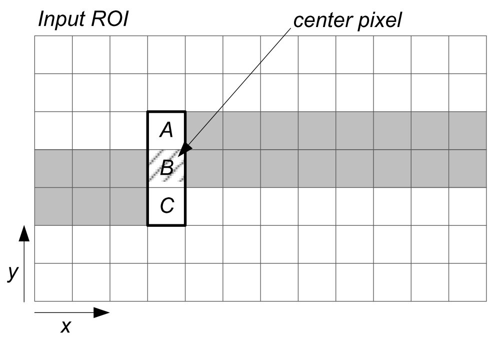


Center of gravity on a ROI profile

Low-pass linear filter

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

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



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

Calibration

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

There are 3 ways to setup this conversion:

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

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

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

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

By convention:

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

Mirrored depth maps

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

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

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

This value changes the behavior of the methods :

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

- `EDepthMapToMeshConverter.Convert`

Scale calibration

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

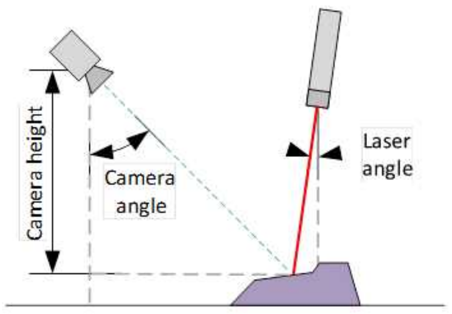
For depth maps coming from laser triangulation setup, this transformation does not produce corrected, metric points. It's main use is to display depth maps as 3D data with the `E3DViewer` class.

Explicit geometric calibration

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

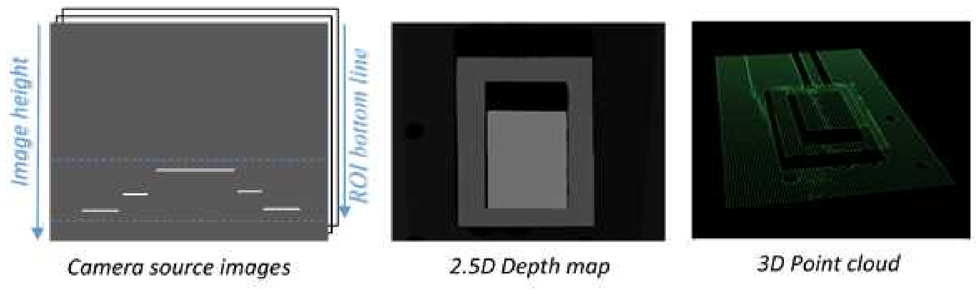
- The angles of the camera and the laser plane, in the counter clockwise direction. The camera angle must be positive.
- The height of the camera above the scanned object.
- The field of view of the camera defined by the sensor size (mm) and the optical focal length (mm).
- The physical distance between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

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



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

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



Object-based calibration

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

For more details, please refer to the "Object-Based Calibration Guidelines" below section.

Object-Based Calibration Guidelines

Easy3D calibration is a powerful process that uses a single scan of a calibration object to calibrate a laser triangulation setup.

1. The calibration process generates a calibration model.
2. **Easy3D** uses this calibration model to transform the laser profile scans (or depth maps) into metric, distortion free point clouds.

- The calibration model includes all the geometric parameters required for this transformation:
  - The relative position of the laser and the camera.
  - The projection and the distortion model of the camera.
  - The relative motion of the object.

This document explains all the steps involved in the calibration process, from the design of the calibration object to the **Open eVision** API.

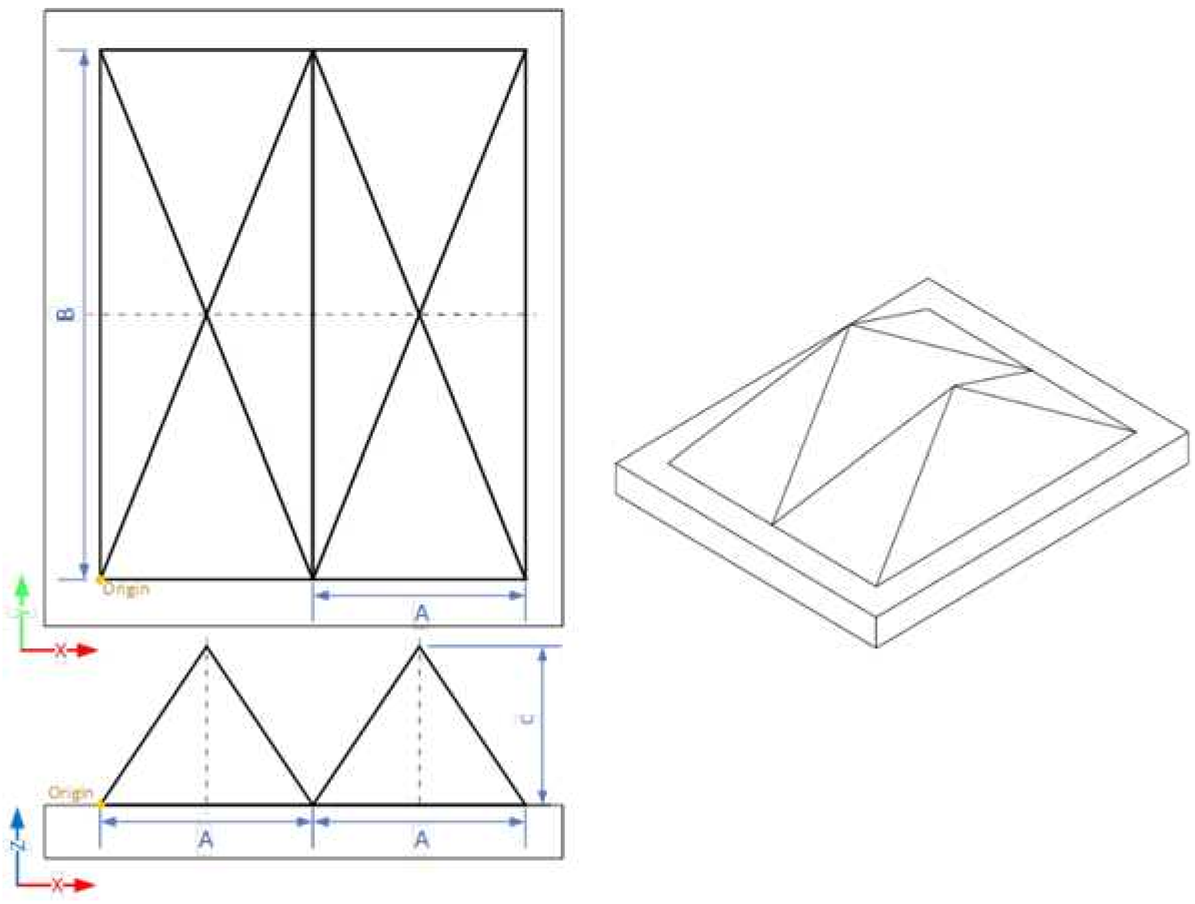
The calibration object

The general principle of **Easy3D** calibration is to match a scan of a known calibration object to its true geometric dimensions.

The double pyramid

TIP
In **Open eVision** 2.7 the “double truncated pyramid” calibration object is recommended over the "double pyramid" model.

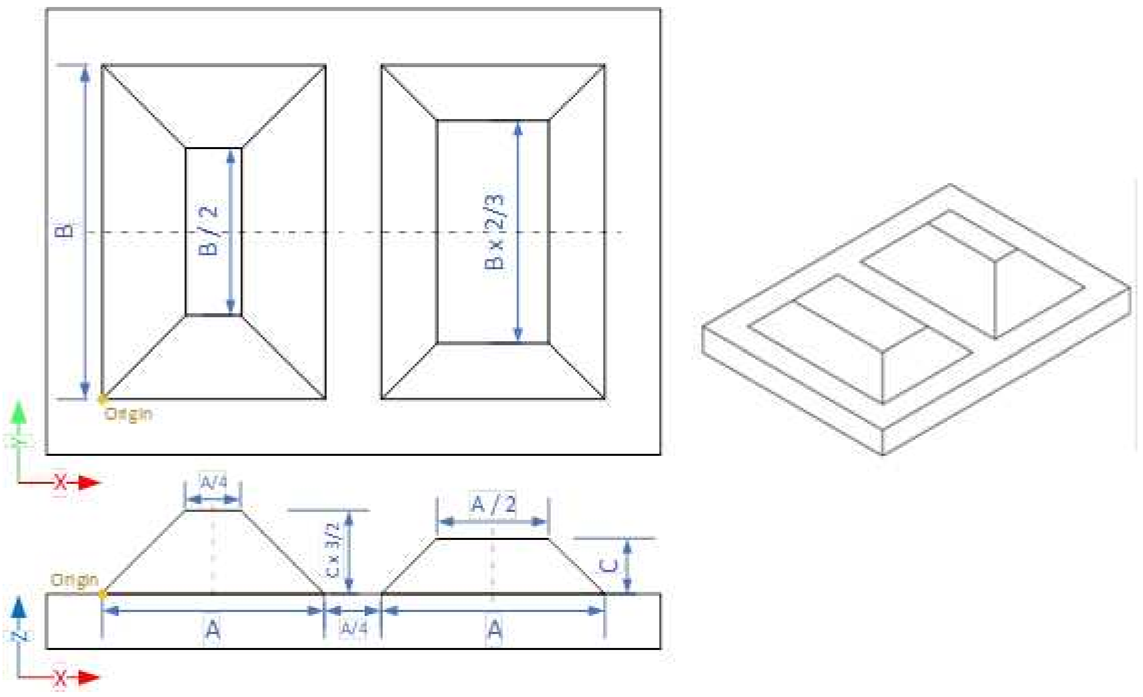
The dimensions of the “double pyramid” calibration object along the X-, Y- and Z-axes are named A, B and C respectively.



The "double pyramid" calibration model

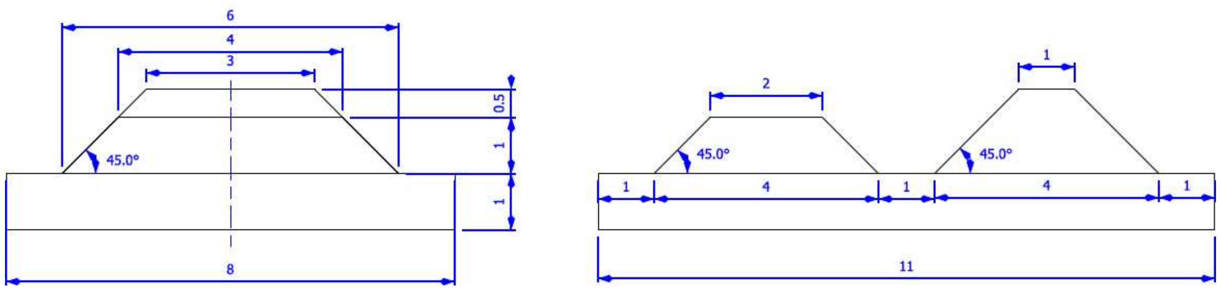
The truncated double pyramid

- The dimensions of the “double truncated pyramid” calibration object the X-, Y- and Z-axes are named A, B and C respectively.
- The design of the double truncated pyramid must follow the ratios given in the illustration below.



The "double truncated pyramid" calibration model (recommended)

- For example, the provided CAD files of the calibration object use A = 4 cm, B = 6 cm and C = 1 cm. The Calibration Object Size, required for the calibration process, are the values A, B and C.



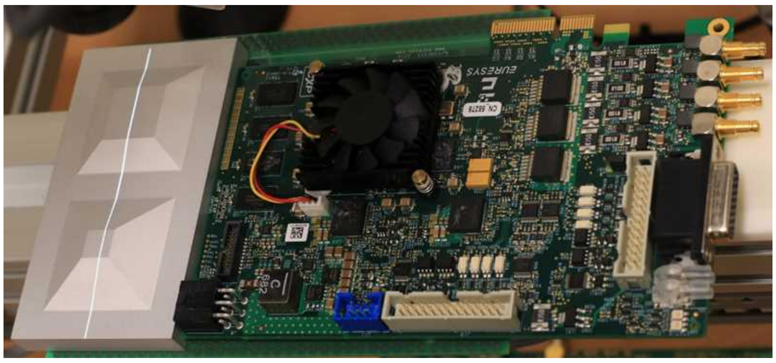
The "double truncated pyramid" calibration model with A = 4, B = 6 and C = 1

Building a calibration object

Overall dimensions

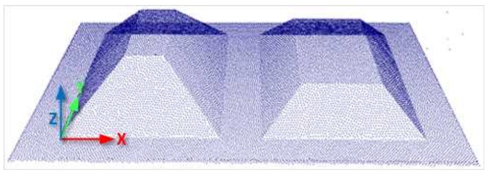
- Manufacture a calibration object that fits the working area of the project.
- For example, if the project targets the inspection of a PCB (a printed circuit board as illustrated), design your calibration object with:
  a. The dimension A or B (it does not matter) similar to the width of the PCB.
  b. The height (C) of only several millimeters.

TIP
This is not a strict requirement, if the scanned object is slightly larger or smaller than the calibration object, the calibration process is still valid.



A PCB scanning setup with the associated calibration object
The calibration object dimensions (A, B and C) match the width and the height of the PCB

TIP
There is no constraint on the orientation of the calibration object during the scan:
- The X-axis can be aligned with the motion direction or with the laser line.
- After the calibration process, the origin and axes of the 3D calibrated point cloud follow the conventions of the reference design.

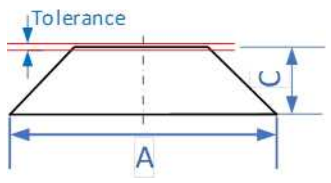


A calibrated point cloud with the origin and the axis of the coordinates system
The 3D origin is located at the external corner of the higher pyramid

Precision and tolerance

The relevant dimensions of the calibration object are the width, the length and the height of the pyramids (called A, B and C in the illustrations).

- The relative dimensions to A, B and C (B/2, A/4…) are important and you must execute them with the same precision.
- The dimensional tolerances are related to the overall expected precision.
  If you want to achieve measurements on the point cloud with a precision of 0.01 mm, the manufacturing of the calibration object must have the same precision.
- These tolerances only apply to the pyramids geometry, the calibration process does not use the dimensions of the support.
- The planar surfaces must be flat between 2 parallel planes separated by the target tolerance, as illustrated.



The tolerance of the pyramids sides is defined as the smallest distance between two parallel planes that contain the entire surface

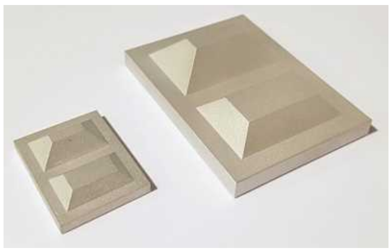
Material and surface finishing

TIP
The goal is to obtain the laser profile as thinnest as possible over the whole object surface with the largest reflected energy.

The build material and the surface finishing are also important and must have:

- A good reflectance, with diffuse reflection (no specular reflections).
- No transmission and limited diffusion inside the material.

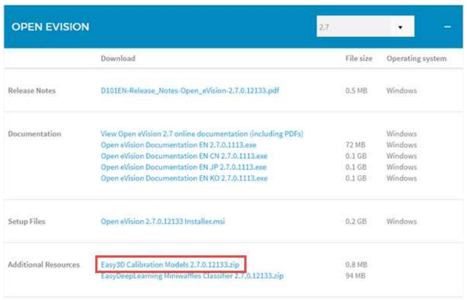
TIP
You can obtain a good surface finishing using aluminum material and blasting. Blasting gives the surfaces a satin gray finish.


2 aluminum machined calibration objects with a micro-abrasive blasting surface treatment

3D CAD models

The calibration object models are available in various 3D CAD format like STEP, OBJ and STL.

Download these files from the Open eVision download area in the Additional Resources section (www.euresys.com/Support).

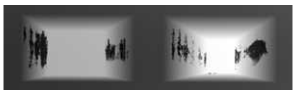


	Download	File size	Operating system
Release Notes	D101EN-Release_Notes-Open_eVision-2.7.0.12133.pdf	0.5 MB	Windows
Documentation	View Open eVision 2.7 online documentation (including PDFs)		Windows
	Open eVision Documentation EN 2.7.0.1113.exe	72 MB	Windows
	Open eVision Documentation EN CN 2.7.0.1113.exe	0.1 GB	Windows
	Open eVision Documentation EN JP 2.7.0.1113.exe	0.1 GB	Windows
	Open eVision Documentation EN KO 2.7.0.1113.exe	0.1 GB	Windows
Setup Files	Open eVision 2.7.0.12133 Installer.msi	0.2 GB	Windows
Additional Resources	Easy3D Calibration Models 2.7.0.12133.zip	0.8 MB	
	EasyDeepLearning Miniwaffles Classifier 2.7.0.12133.zip	94 MB	

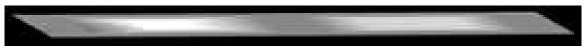
Download the calibration object models

Scanning the calibration object

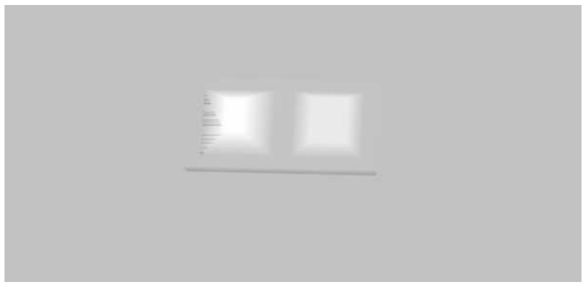
- The scan of the calibration object produces a depth map.
- To ensure a correct detection of the calibration object and a precise calibration model, you must fulfill the following criteria:
  - ◻ All faces of the calibration object must be visible on the depth map (this affects the orientation of both the camera and the laser).
  - ◻ No other object can be higher than the calibration object in the depth map.
  - ◻ The depth map must have at least 200 x 200 pixels.
  - ◻ The calibration object must cover at least 50% of the defined pixels of the depth map.
- Examples of bad scans:



Missing pixels on the side faces



Not enough lines



The calibration object is too small on the depth map

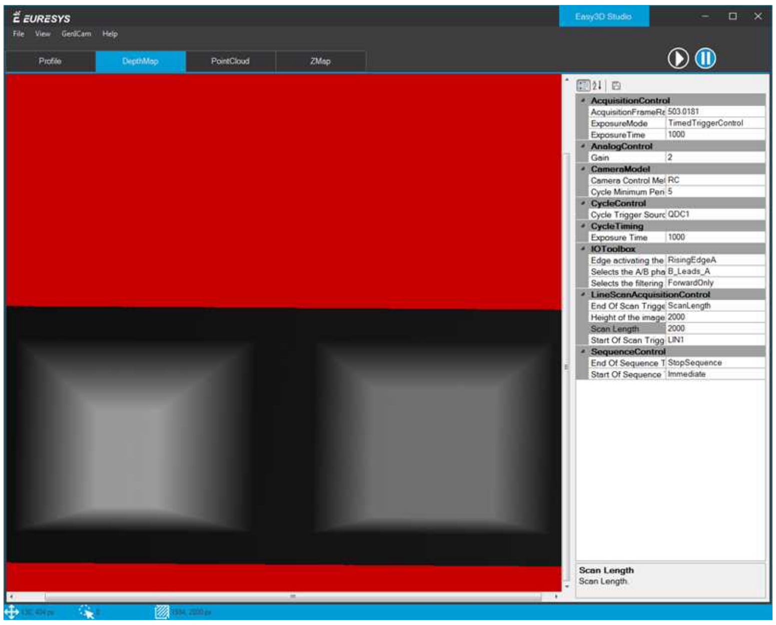
Calibration with Easy3D Studio

Easy3D Studio is a free application that helps you to set up a laser triangulation scanner. You can easily set the acquisition parameters of the **Coaxlink Quad 3D LLE** frame grabber and perform the calibration.

The DepthMap panel

This panel displays:

- The scanned image.
- The acquisition parameters on the right side.



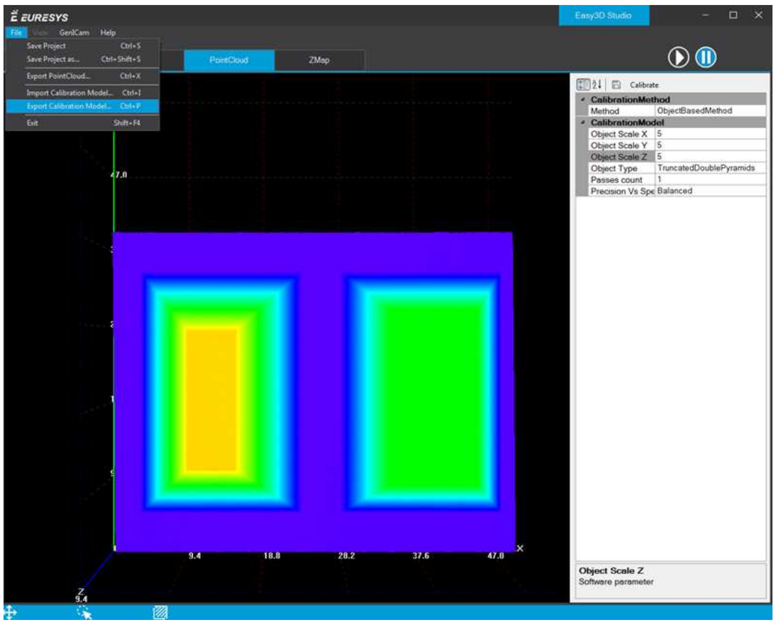
AcquisitionControl	
AcquisitionFrameRa	503.0181
ExposureMode	TimedTriggerControl
ExposureTime	1000
AnalogControl	
Gain	2
CameraModel	
Camera Control Me	RC
Cycle Minimum Peri	5
CycleControl	
Cycle Trigger Sourc	QDC1
CycleTiming	
Exposure Time	1000
IOToolbox	
Edge activating the	RisingEdgeA
Selects the A/B pha	B_Leads_A
Selects the filtering	ForwardOnly
LineScanAcquisitionControl	
End Of Scan Trigge	ScanLength
Height of the image	2000
Scan Length	2000
Start Of Scan Trigg	LIN1
SequenceControl	
End Of Sequence T	StopSequence
Start Of Sequence	Immediate

Scan Length
Scan Length.

The PointCloud panel

This panel displays:

- The depth map of the scanned image.
- The object-based calibration parameters on the right side.
- The Calibrate button computes the calibration model using the last scanned depth map.
- When the calibration model is ready, the depth map is transformed into a point cloud.
- You can export the calibration model for later use.


EURESYS
Easy3D Studio
File
GenICam
Help
Save Project Ctrl+S
Save Project as... Ctrl+Shift+S
Export PointCloud... Ctrl+X
Import Calibration Model... Ctrl+I
Export Calibration Model... Ctrl+P
Exit Shift+F4
PointCloud
ZMap
Calibrate
47.0
9.4
18.8
28.2
37.6
47.0
X
Z
9.4

CalibrationMethod	
Method	ObjectBasedMethod
CalibrationModel	
Object Scale X	5
Object Scale Y	5
Object Scale Z	5
Object Type	TruncatedDoublePyramids
Passes count	1
Precision Vs Spe	Balanced

Object Scale Z
Software parameter

Required parameters

The calibration based on a calibration object requires several parameters:

- Set the Object Type as DoublePyramid or TruncatedDoublePyramid.
  - The DoublePyramid object type is deprecated and not recommended.
- Set the Object Size to represent the real size of the calibration object.
  - If your calibration object has a base of 20 mm by 30 mm and a height of 5 mm, set these values in the Object Size A/B/C parameters.
  - The point cloud after the calibration uses coordinates in millimeters.
- Set the parameter Precision Vs Speed Trade Off to define the time spent on the calibration process.
  - The 3 possible values are Fast, Balanced and Precise.

- Set the parameter Passes count to define the number of iterations used to refine the calibration model.
  - Use 1 for the fastest processing.
  - Use up to 3 for slower but potentially better calibration model.

Using the calibration with Open eVision

- The class `EObjectBasedCalibrationModel` is the container for the object based calibration model.
- The class `EObjectBasedCalibrationGenerator` performs the computation of such a model using an EDepthMap8/16/32f as input.

The following code snippet illustrates the calculation of a calibration model:

```
// Initialize a depth map from an image of a double truncated pyramid
EDepthMap16 depth_map;
depth_map.LoadImage("ctx1 calibration object.png");     // from Easy3D sample images
depth_map.SetZResolution(1.f / (1 << 5));               // 11.5 fixed point pixel format

// Initialize the calibration generator
EObjectBasedCalibrationGenerator calib_generator;
calib_generator.SetCalibrationObjectType(EObjectBasedCalibrationType_
TruncatedDoublePyramid);
calib_generator.SetCalibrationObjectSize(40.f, 60.f, 10.f); // Size of the calibration
object

// Compute the calibration model
EObjectBasedCalibrationModel calib_model;
calib_model = calib_generator.Compute(depth_map);
float error = calib_model.GetCalibrationError();

// Save the calibration model
calib_model.Save("calib.model");
```

The following code snippet illustrates the use of a saved calibration model:

```
// Load the calibration model
EObjectBasedCalibrationModel calib_model;
calib_model.Load("calib.model");

// Load a depth map (captured in the same context)
EDepthMap16 depth_map;
depth_map.LoadImage("ctx1 shapes.png");
depth_map.SetZResolution(1.f / (1 << 5));

// Initialize a converter, use the loaded model
EDepthMapToPointCloudConverter converter;
converter.SetCalibrationModel(calib_model);

// Convert the depth map to a metric point cloud and save it
EPointCloud point_cloud;
converter.Convert(depth_map, point_cloud);
point_cloud.SavePCD("point_cloud.pcd");
```

To experiment and learn about the **Easy3D** calibration, a C++ sample called `3DCalibration` is provided with the source code in the **Open eVision** distribution.

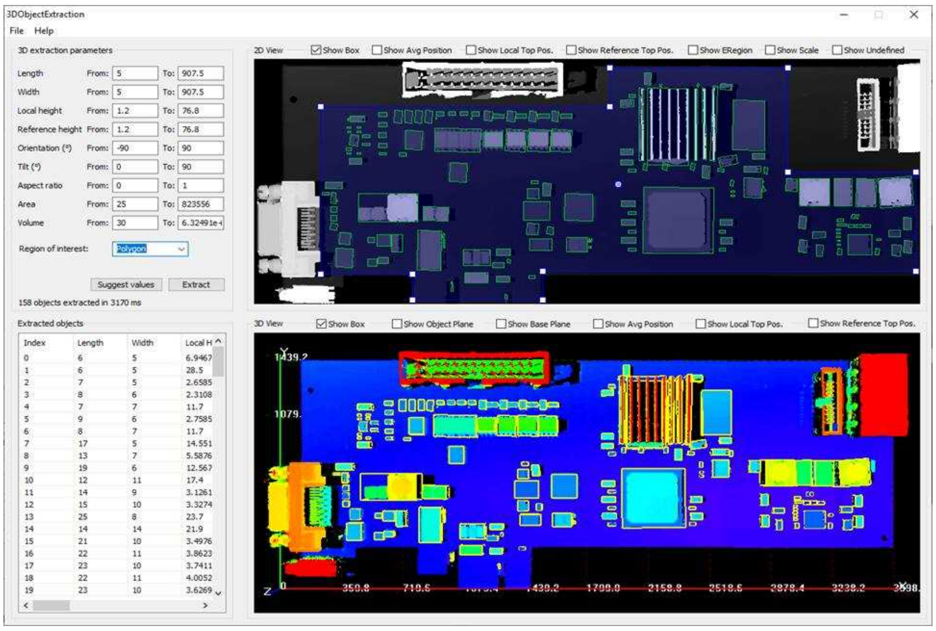
3.3. Easy3DObject - Extracting 3D Objects

Purpose and Workflow

Introduction

- The Easy3DObject tool extracts objects and their features from a ZMap.
  - The `E3DObjectExtractor` class uses a set of criteria to select the objects to extract.
  - The extracted objects are instances of the `ED3Object` class.
- Open eVision provides a demo application with C++ source code and 2 C++ / C# samples:

  This demo application exposes most of the features of the **Easy3DObject** tool.



3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

158 objects extracted in 3170 ms

Extracted objects

Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	7	5	2.6585
3	8	6	2.3108
4	7	7	11.7
5	9	6	2.7585
6	8	7	11.7
7	17	5	14.551
8	13	7	5.5876
9	19	6	12.567
10	12	11	17.4
11	14	9	3.1261
12	15	10	3.3274
13	25	8	23.7
14	14	14	21.9
15	21	10	3.4976
16	22	11	3.8623
17	23	10	3.7411
18	22	11	4.0052
19	23	10	3.6269

Library workflow

1. Load or build a ZMap (from an image or a point cloud).
2. Construct an `E3DObjectExtractor` instance.
3. Set the selection criteria of the `E3DObjectExtractor` instance.
4. Extract the 3D objects, with or without an `ERegion`.
5. Get and process the extracted objects list.

Load or build a ZMap

A ZMap is a grayscale image with a metric coordinate system. It is sometimes referred to as a "height map".

You can create a ZMap from an 8- or a 16-bit image or generate it from a point cloud.

- Before using an image as a ZMap, set the resolution.

> **TIP**
> The resolution is the metric size of a pixel (for example in mm / pixel) and the height difference between 2 consecutive grayscale levels.

- From a point cloud, use the `EPointCloudToZMapConverter` class to generate a ZMap. Choose the target ZMap resolution according to the point cloud sampling.
- Depending on the 3D scan precision, you can use a ZMap with 8- or 16-bit per pixel.

> **TIP**
> A 16-bit processing is more accurate but slower than an 8-bit processing.

Object Features

Units

Both the `E3DObjectExtractor` parameters and the `E3DObject` features are expressed in metric units.

- For example: if the resolution of the input `EZMap` is expressed in mm / pixel, the length parameter is expressed in mm.
- Use the `Resolution` accessors of the `EZMap` to query and change its resolution.

Angles are expressed in the unit defined by `Easy.AngleUnit`.

> **TIP**
> In this documentation, we use the default setting and all angles are expressed in degrees.

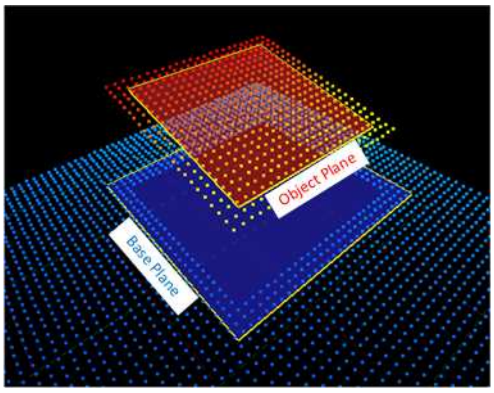
Object plane and base plane

The E3DObjectExtractor fits a plane to the pixels of each E3DObject output:

- Use E3DObject.Plane to access this plane.

The E3DObjectExtractor also tries to fit a plane to the pixels surrounding an E3DObject

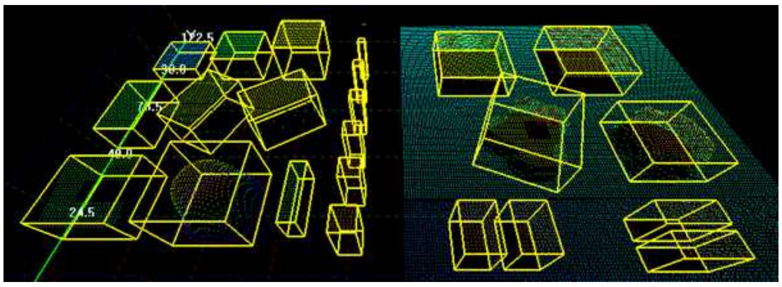
- This plane is called the *base plane*.
- It is an estimation of the local background around the object.
- If there are too many undefined pixels in this area, the base plane is equal to the reference plane of the input EZMap.



Bounding box

The *bounding box* is the minimal enclosing rectangle for all the object positions.

- It is oriented in the XY plane of the ZMap space (rotation around the Z axis of the ZMap).
- Its rotation is used as the orientation of the object (see E3DObject.GetOrientation).
- Its X and Y sizes are the object length and width (see E3DObject.GetLength and E3DObject.GetWidth).
- Its Z size is always in the Z axis of the ZMap direction.

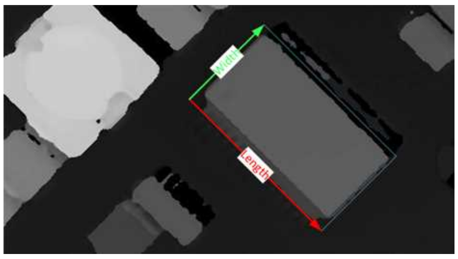


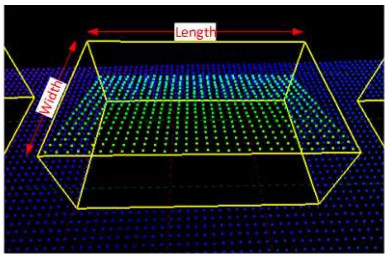
Length and width

The *length* of an object is the largest dimension on the XY plane in the ZMap space. It is the same as the size of the major axis of the bounding box.

The *width* of an object is the smallest dimension on the XY plane in the ZMap space. It is the same as the size of the minor axis of the bounding box.

Use the `E3DObjectExtractor.LengthRange` and the `E3DObjectExtractor.WidthRange` accessors to set the ranges of allowed values for the length and the width.





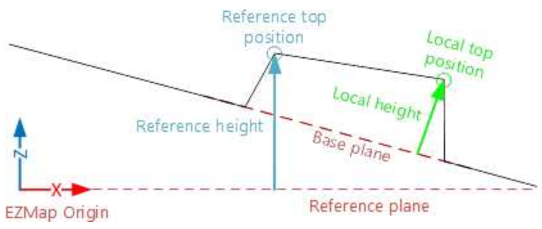
Local and reference top positions and heights

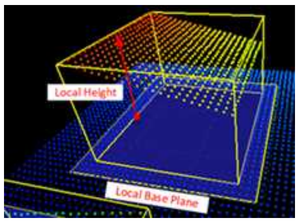
The *local top position* of an object is the position (3D coordinates) of the point in the `E3DObject` that is the furthest from the base plane.

The *local height* of an object is the distance between the local top position and the base plane.

The *reference top position* of an object is the position (3D coordinates) of the point in the `E3DObject` that is the furthest from the reference plane.

The *reference height* of an object is the distance between the reference top position and the reference plane.





If there are too many undefined pixels in the object surroundings:

- The base plane is equal to the reference plane of the input `EZMap`.
- The local top position is equal to the reference top position.
- The local height is equal to the reference height.

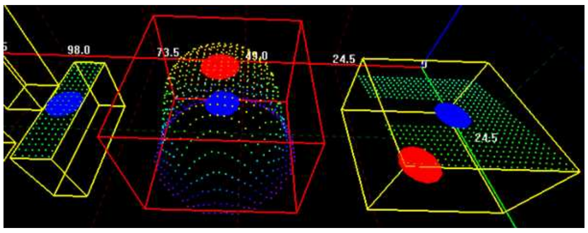
Use the `E3DObjectExtractor.LocalHeightRange` and the `E3DObjectExtractor.ReferenceHeightRange` accessors to set the ranges of allowed values for the local and the reference height.

Average position

The *average position* is the arithmetic mean of the 3D positions of the object, also known as the barycenter.

In the illustration below:

- The average position is displayed in blue.
- The top position is displayed in red.
- On the left object, the average and the top positions are at the same place.
- On the center object the average position is “inside” the object.

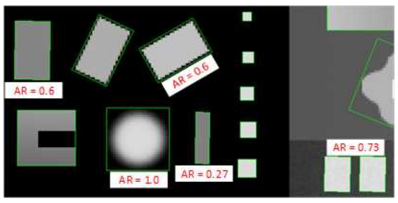


Aspect ratio

The *aspect ratio* is the width (the smallest dimension on the XY plane) divided by the length (the largest dimension).

- It lies between 0 and 1.
- The smaller the ratio, the more elongated the object is.
- A square has an aspect ratio of 1.

Use the `E3DObjectExtractor.AspectRatioRange` accessor to set the range of allowed values for the aspect ratio.

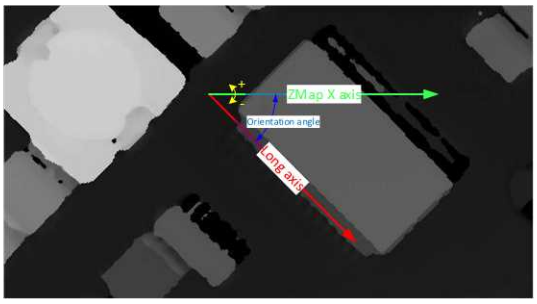


Orientation angle

The *orientation angle* is the angle between the X axis of the `EZMap` and the longest axis (the length) of the object.

- The angle is measured in the clockwise direction.
- The value must lie between -90° and +90°.

Use the `E3DObjectExtractor.OrientationRange` accessor to set the range of allowed values for the orientation angle.



Local and reference tilt angles

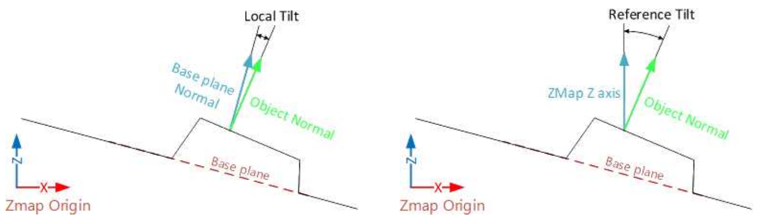
The *local tilt angle* is the angle between the base plane and the object plane.

- A value of 0 means that the object top surface is parallel to its base.
- The value must lie between 0° and +90°.

The *reference tilt angle* is the angle between the object plane and ZMap XY plane.

- A value of 0 means that the object top surface is parallel to its base.
- The value must lie between 0° and +90°.

Use the `E3DObjectExtractor.LocalTiltRange` and the `E3DObjectExtractor.ReferenceTiltRange` accessors to set the range of allowed values for the tilt angles.

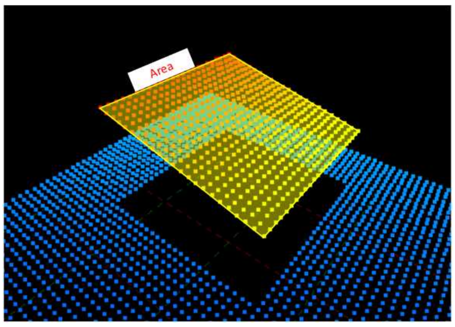


Area

The object *area* is the area of the top surface of the object projected on the reference plane of the EZMap.

- It is equal to [the number of pixels in the object] × [the x-resolution of the EZMap] × [the y-resolution of the EZMap].

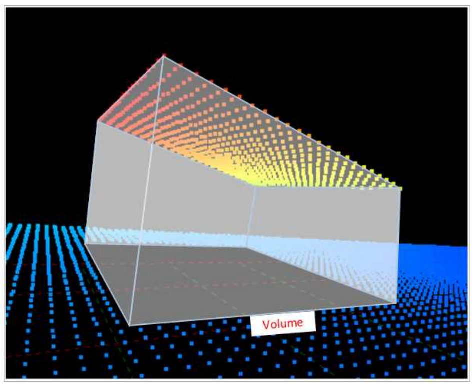
Use the E3DObjectExtractor.AreaRange accessor to set the range of allowed values for the area.



Volume

The object *volume* is the volume that lies between the top surface and the base plane of the object.

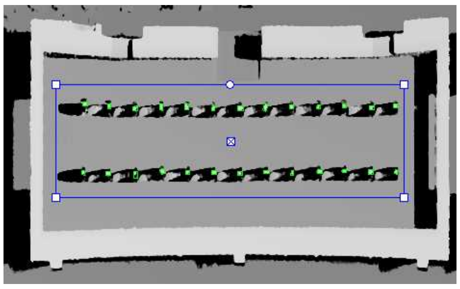
Use the E3DObjectExtractor.VolumeRange accessor to set the range of allowed values for the volume.



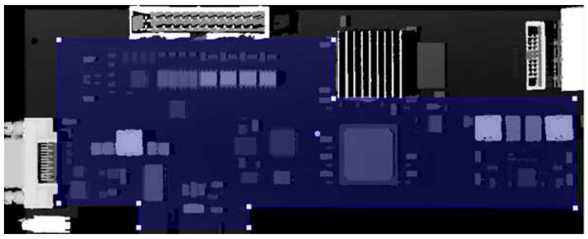
Extracting and Using Objects

Extracting the objects

Use the E3DObjectExtractor.Extract method to perform the objects extraction.



You can limit the extraction to an ERegion, for example to ignore parts of the ZMap that are not interesting and/or to speed up the extraction process.



The processing speed of the extraction depends directly on:

- The number of pixels in the ZMap or in the ERegion.
- The number of segmented objects.

TIP
Adjust the extraction ranges to reduce the number of objects and speed up the extraction process.

Using the objects

The E3DObjectExtractor.Extract method populates a list of E3DObject fulfilling your set criteria.

- Each E3DObject is a collection of descriptive features of the associated 3D points in the EZMap, such as its oriented bounding box, its local height and its volume.
- Call the associated E3DObject method to access a feature.
- The E3DObject list is sorted from the smallest area to the largest area.

The code snippet below provides an example for extracting features from the `E3DObject` list.

```
////////////////////////////////////////////////////
// This code snippet shows how to read a QR code  //
// and retrieve the decoded data.                 //
////////////////////////////////////////////////////

// get the extracted objects and loop over them
std::vector<Easy3D::E3DObject> objects = extractor.GetObjects();
int nObjects = objects.size();
for (int index = 0; index < nObjects; ++index)
{
  // inspect bounding box dimensions
  E3DPoint bbCenter = objects[index].GetBoundingBox().GetCenter();
  float bbHeight = objects[index].GetBoundingBox().GetXSize();
  float bbLength = objects[index].GetBoundingBox().GetYSize();

  // inspect object plane and base plane
  Easy3D::E3DPlane opjPlane = objects[index].GetPlane();
  Easy3D::E3DPlane basePlane = objects[index].GetBasePlane();

  // inspect the ERegion that exactly contains the object
  ERegion objRegion = objects[index].GetRegion();
}
```

Visualizing the objects

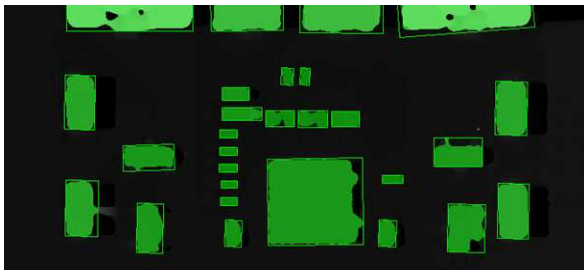
To visualize some of these features in 2D or 3D:

- Use the `E3DObject.Draw` method.
- Or submit a list of `E3DObject` to an `E3DViewer`.

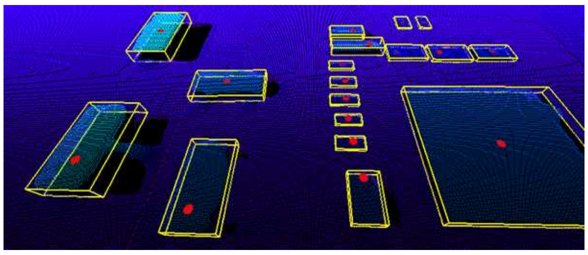
TIP
In an `E3DViewer`, use the `ERenderStyle` structure to choose your rendering style.

The following code snippets illustrate how to draw some object features:

- In a 2D graphic context: Drawing a 2D Feature from the List of E3DObjects



- In a 3D viewer: Drawing 3D Features from a List of E3DObjects

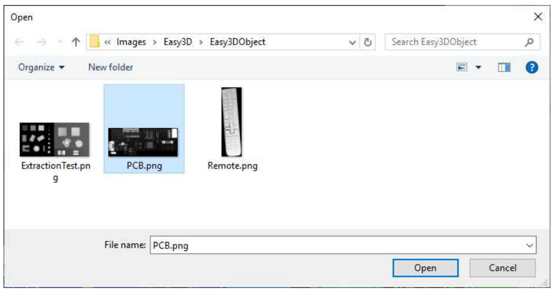


Use Case - Inspecting a PCB

The purpose of this use case is to test if all the components are present and correctly placed on the PCB.

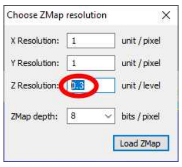
TIP
This example uses the sample image `Sample Images\Easy3D\Easy3DObject\PCB.png` and the illustrations are based on the **Easy3DObject** demo application.

1. Load the PCB image.



File name: PCB.png

2. Set the resolution.
   - The provided PCB sample is an 8-bit gray scale image.
   - Use a Z resolution of 0.3 metric unit per gray scale level for a realistic proportion.



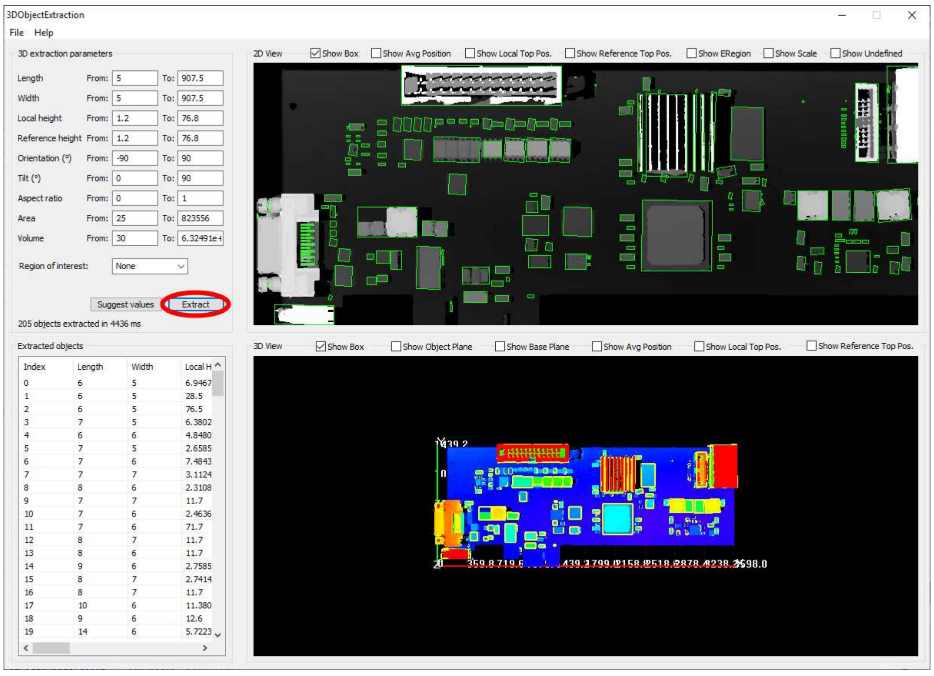
X Resolution: 1 unit / pixel
Y Resolution: 1 unit / pixel
Z Resolution: 0.3 unit / level
ZMap depth: 8 bits / pixel

3. Keep the suggested parameters for a first extraction.
   - The suggested parameters are set from the ZMap width, height and resolution.

4. Click on the Extract button to perform the extraction.

   When the extraction is done:

   - The object list is filled.
   - Click on a column title to sort the object list.
   - The various measures are displayed.
   - The 2D View and the 3D View show the extracted object bounding boxes.



3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

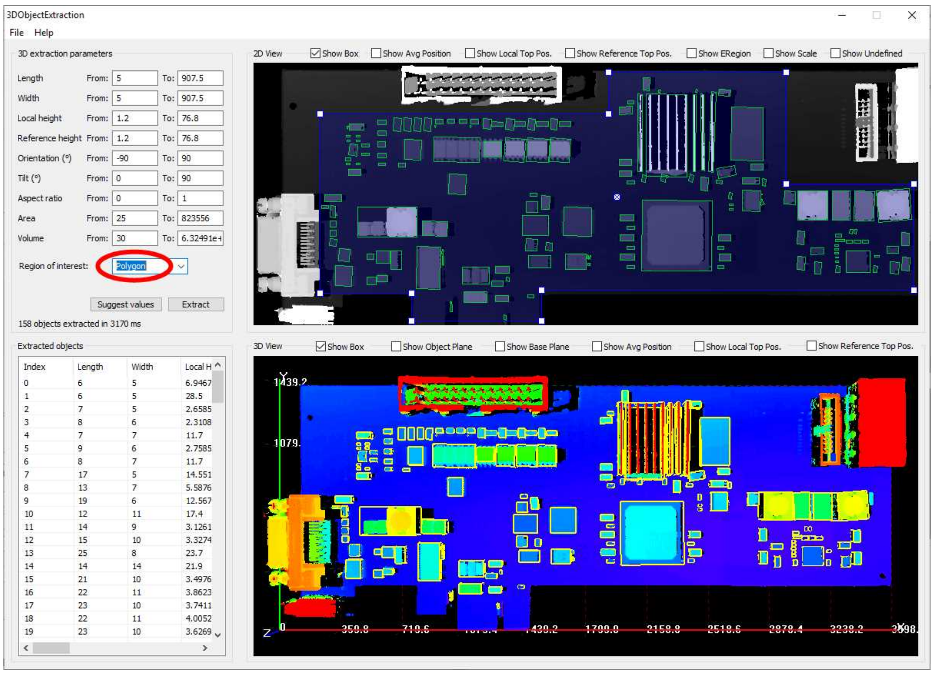
Region of interest: None

205 objects extracted in 4436 ms

Extracted objects

Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	6	5	76.5
3	7	5	6.3802
4	6	6	4.8480
5	7	5	2.6585
6	7	6	7.4843
7	7	7	3.1124
8	8	6	2.3108
9	7	7	11.7
10	7	6	2.4636
11	7	6	71.7
12	8	7	11.7
13	8	6	11.7
14	9	6	2.7585
15	8	7	2.7414
16	8	7	11.7
17	10	6	11.380
18	9	6	12.6
19	14	6	5.7223

5. Use a polygon region of interest to restrict the searched area.
   - You can limit the extraction to a region defined as a rectangle, a polygon or an ellipse in the demo application.
   - Use the Open eVision API, to define and use any `ERegion`.



3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

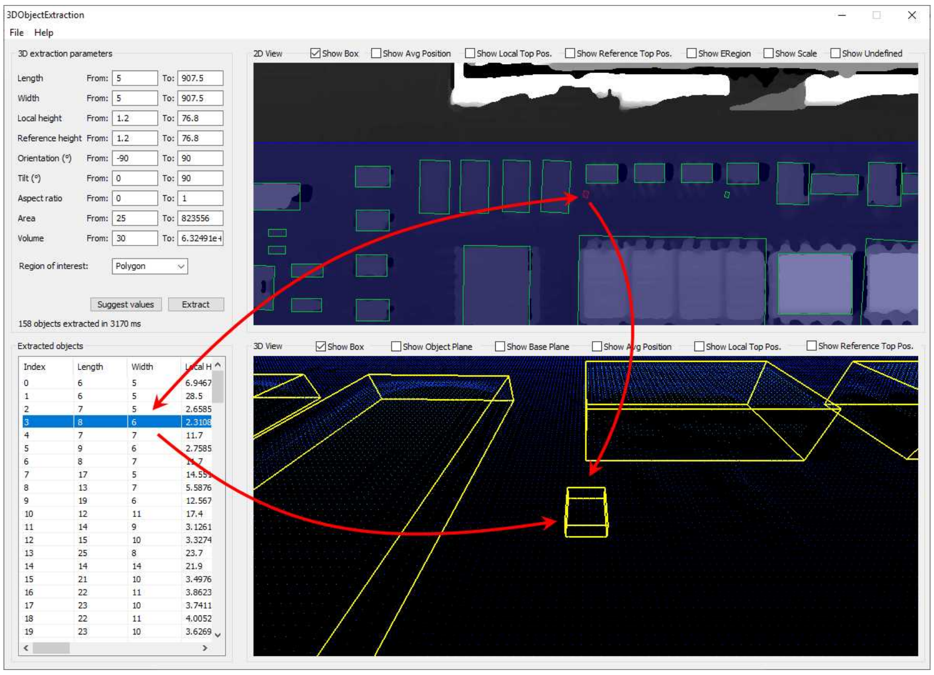
Region of interest: Polygon

158 objects extracted in 3170 ms

Extracted objects

Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	7	5	2.6585
3	8	6	2.3108
4	7	7	11.7
5	9	6	2.7585
6	8	7	11.7
7	17	5	14.551
8	13	7	5.5876
9	19	6	12.567
10	12	11	17.4
11	14	9	3.1261
12	15	10	3.3274
13	25	8	23.7
14	14	14	21.9
15	21	10	3.4976
16	22	11	3.8623
17	23	10	3.7411
18	22	11	4.0052
19	23	10	3.6269

6. Press again the Extract button to generate a new list of objects. Now, only the objects located inside the region are extracted.
7. The 2D View and 3D View automatically focus on the object selected in the list. You can also select an object by clicking on a bounding box in the 2D View.



3D extraction parameters

Length	From:	5	To:	907.5
Width	From:	5	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

158 objects extracted in 3170 ms

Extracted objects

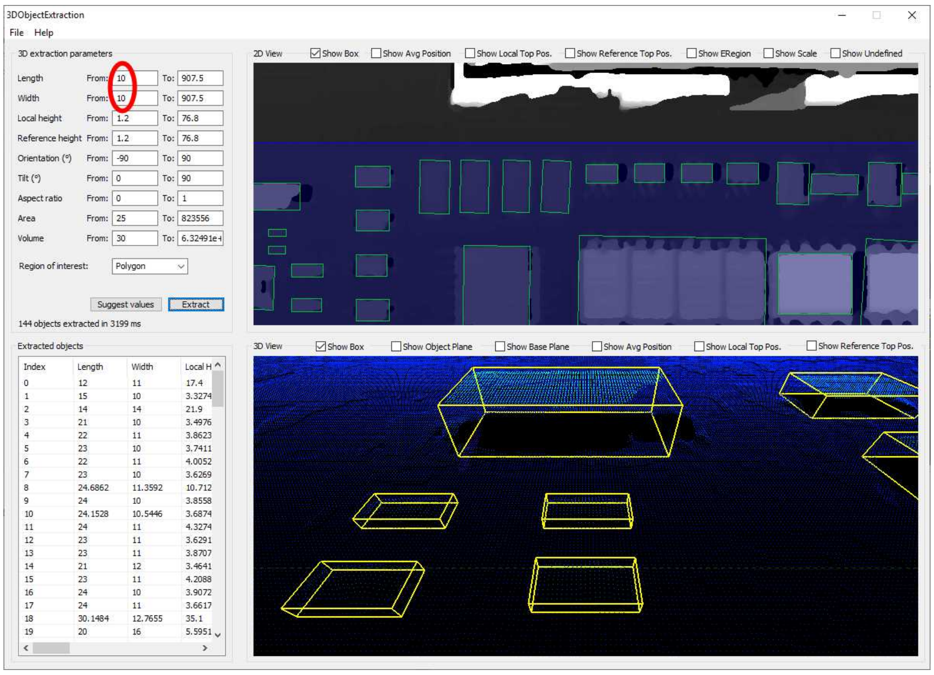
Index	Length	Width	Local H
0	6	5	6.9467
1	6	5	28.5
2	7	5	2.6585
3	8	6	2.3108
4	7	7	11.7
5	9	6	2.7585
6	8	7	11.7
7	17	5	14.551
8	13	7	5.5876
9	19	6	12.567
10	12	11	17.4
11	14	9	3.1261
12	15	10	3.3274
13	25	8	23.7
14	14	14	21.9
15	21	10	3.4976
16	22	11	3.8623
17	23	10	3.7411
18	22	11	4.0052
19	23	10	3.6269

8. Use the size ranges to discard the smaller components.

To add or remove objects:

- Change the extraction parameters, like the length and width ranges.
- In the illustration below, objects smaller than 10x10 metric unit are not extracted.

NOTE
After changing a parameter, press the Extract button to perform a new extraction.



3D extraction parameters

Length	From:	10	To:	907.5
Width	From:	10	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

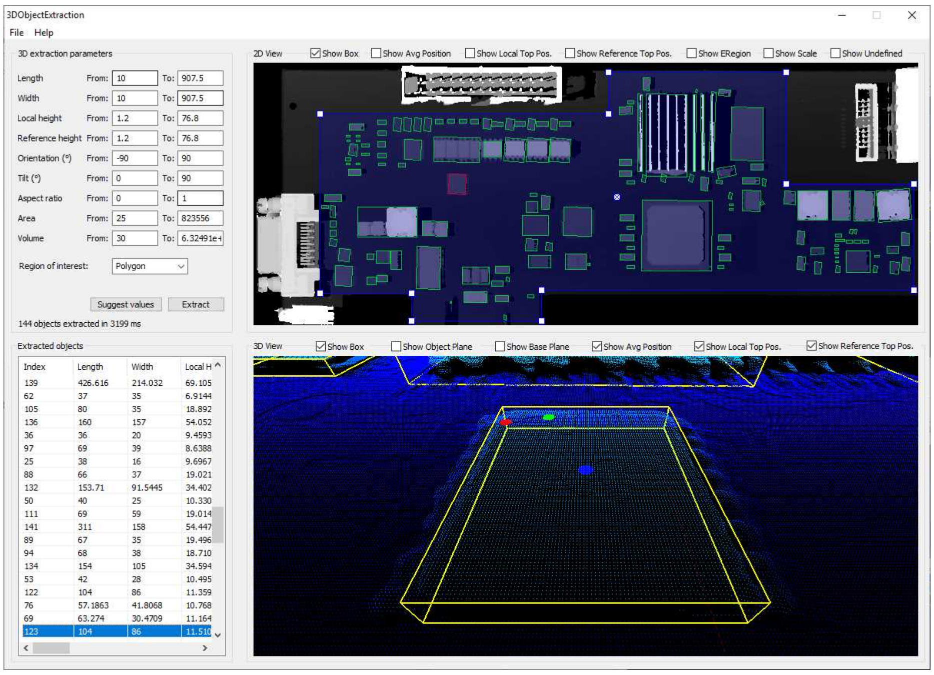
Region of interest: Polygon

144 objects extracted in 3199 ms

Extracted objects

Index	Length	Width	Local H
0	12	11	17.4
1	15	10	3.3274
2	14	14	21.9
3	21	10	3.4976
4	22	11	3.8623
5	23	10	3.7411
6	22	11	4.0052
7	23	10	3.6269
8	24.6862	11.3592	10.712
9	24	10	3.8558
10	24.1528	10.5446	3.6874
11	24	11	4.3274
12	23	11	3.6291
13	23	11	3.8707
14	21	12	3.4641
15	23	11	4.2088
16	24	10	3.9072
17	24	11	3.6617
18	30.1484	12.7655	35.1
19	20	16	5.5951

9. Check or uncheck the boxes at the top of the views to toggle the display of most of the object features, either in the 2D View or the 3D View.
    - In the illustration below, the object list is sorted by local height.
    - The first object is selected and displayed in both views.


3DObjectExtraction

File Help

3D extraction parameters

Length	From:	10	To:	907.5
Width	From:	10	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

144 objects extracted in 3199 ms

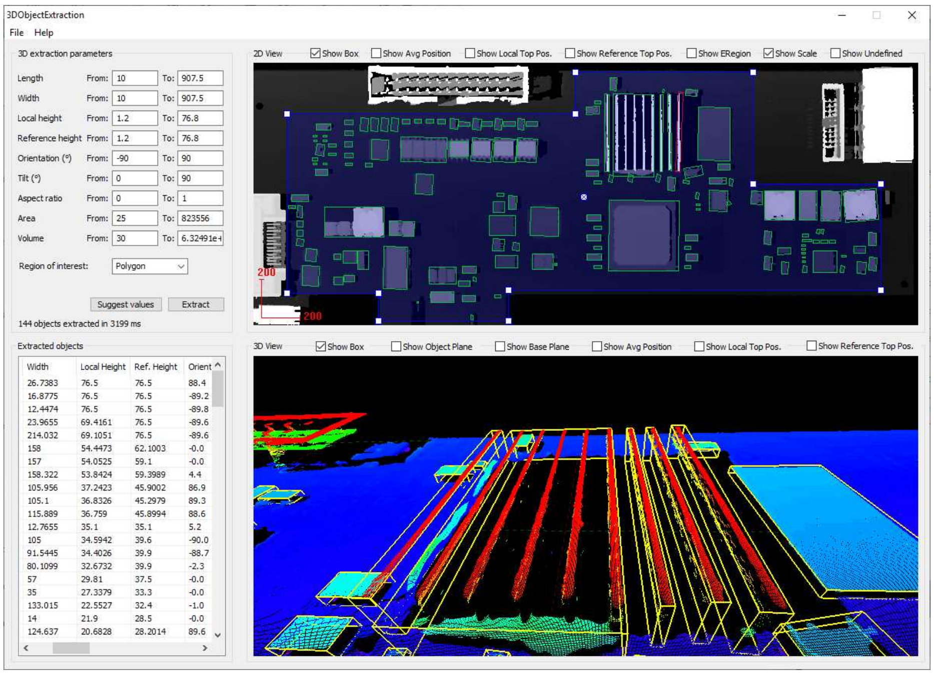
Extracted objects

Index	Length	Width	Local H
139	426.616	214.032	69.105
62	37	35	6.9144
105	80	35	18.892
136	160	157	54.052
36	36	20	9.4593
97	69	39	8.6388
25	38	16	9.6967
88	66	37	19.021
132	153.71	91.5445	34.402
50	40	25	10.330
111	69	59	19.014
141	311	158	54.447
89	67	35	19.496
94	68	38	18.710
134	154	105	34.594
53	42	28	10.495
122	104	86	11.359
76	57.1863	41.8068	10.768
69	63.274	30.4709	11.164
123	104	86	11.510

2D View: ✓ Show Box, ☐ Show Avg Position, ☐ Show Local Top Pos., ☐ Show Reference Top Pos., ☐ Show ERegion, ☐ Show Scale, ☐ Show Undefined

3D View: ✓ Show Box, ☐ Show Object Plane, ☐ Show Base Plane, ✓ Show Avg Position, ✓ Show Local Top Pos., ✓ Show Reference Top Pos.

10. Adjust the extraction parameters to accept or reject objects based on the results.


3DObjectExtraction

File Help

3D extraction parameters

Length	From:	10	To:	907.5
Width	From:	10	To:	907.5
Local height	From:	1.2	To:	76.8
Reference height	From:	1.2	To:	76.8
Orientation (°)	From:	-90	To:	90
Tilt (°)	From:	0	To:	90
Aspect ratio	From:	0	To:	1
Area	From:	25	To:	823556
Volume	From:	30	To:	6.32491e+

Region of interest: Polygon

144 objects extracted in 3199 ms

Extracted objects

Width	Local Height	Ref. Height	Orient
26.7383	76.5	76.5	88.4
16.8775	76.5	76.5	-89.2
12.4474	76.5	76.5	-89.8
23.9655	69.4161	76.5	-89.6
214.032	69.1051	76.5	-89.6
158	54.4473	62.1003	-0.0
157	54.0525	59.1	-0.0
158.322	53.8424	59.3989	4.4
105.956	37.2423	45.9002	86.9
105.1	36.8326	45.2979	89.3
115.889	36.759	45.8994	88.6
12.7655	35.1	35.1	5.2
105	34.5942	39.6	-90.0
91.5445	34.4026	39.9	-88.7
80.1099	32.6732	39.9	-2.3
57	29.81	37.5	-0.0
35	27.3379	33.3	-0.0
133.015	22.5527	32.4	-1.0
14	21.9	28.5	-0.0
124.637	20.6828	28.2014	89.6

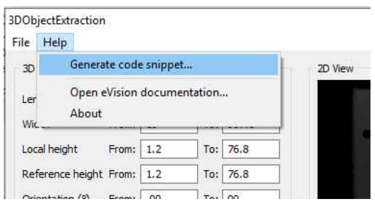
2D View: ✓ Show Box, ☐ Show Avg Position, ☐ Show Local Top Pos., ☐ Show Reference Top Pos., ☐ Show ERegion, ✓ Show Scale, ☐ Show Undefined

3D View: ✓ Show Box, ☐ Show Object Plane, ☐ Show Base Plane, ☐ Show Avg Position, ☐ Show Local Top Pos., ☐ Show Reference Top Pos.

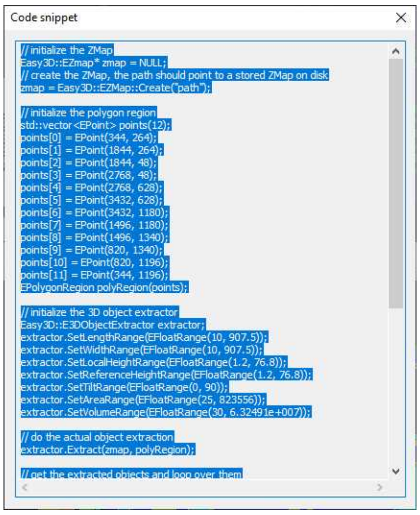
11. Open the Help menu and click on Generate code snippet to generate the C++ code corresponding to the current configuration.

    The generated code illustrates how you can:

    - Load a ZMap.
    - Define a region.
    - Set the configuration parameters of the extraction.
    - Start the extraction process.
    - Iterate through the resulting objects list.



Local height From: 1.2 To: 76.8
Reference height From: 1.2 To: 76.8



```
// initialize the ZMap
Easy3D::EZmap* zmap = NULL;
// create the ZMap, the path should point to a stored ZMap on disk
zmap = Easy3D::EZMap::Create("path");

// initialize the polygon region
std::vector<EPoint> points(12);
points[0] = EPoint(344, 264);
points[1] = EPoint(1844, 264);
points[2] = EPoint(1844, 48);
points[3] = EPoint(2768, 48);
points[4] = EPoint(2768, 628);
points[5] = EPoint(3432, 628);
points[6] = EPoint(3432, 1180);
points[7] = EPoint(1496, 1180);
points[8] = EPoint(1496, 1340);
points[9] = EPoint(820, 1340);
points[10] = EPoint(820, 1196);
points[11] = EPoint(344, 1196);
EPolygonRegion polyRegion(points);

// initialize the 3D object extractor
Easy3D::E3DObjectExtractor extractor;
extractor.SetLengthRange(EFloatRange(10, 907.5));
extractor.SetWidthRange(EFloatRange(10, 907.5));
extractor.SetLocalHeightRange(EFloatRange(1.2, 76.8));
extractor.SetReferenceHeightRange(EFloatRange(1.2, 76.8));
extractor.SetTiltRange(EFloatRange(0, 90));
extractor.SetAreaRange(EFloatRange(25, 823556));
extractor.SetVolumeRange(EFloatRange(30, 6.32491e+007));

// do the actual object extraction
extractor.Extract(zmap, polyRegion);

// get the extracted objects and loop over them
```

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;
EImageBW8 dstImage;
```

```
// 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;
```

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

```
//Pointer to the third-party image buffer
EBW8* imgPtr;
```

```
// ...
```

```
// 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 (fastest)  //
// to access the pixel values in a BW8 image                 //
////////////////////////////////////////////////////////////////
```

```
EImageBW8 img;
```

```
OEV_UINT8* pixelPtr;
OEV_UINT8* rowPtr;
OEV_UINT8  pixelValue;
OEV_UINT32 rowPitch;
OEV_UINT32 x, y;
```

```
rowPtr = reinterpret_cast <OEV_UINT8*>(img.GetImagePtr());
rowPitch = img.GetRowPitch();
```

```
for (y = 0; y < height; y++)
{
    pixelPtr = rowPtr;

    for (x = 0; x < width; x++)
    {
        pixelValue = *pixelPtr;

        // Add your pixel computation code here

        *pixelPtr = pixelValue;
        pixelPtr++;
    }

    rowPtr += rowPitch;
}
```

ROI Placement

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

```
// Image constructor
EImageBW8 parentImage;
```

```
// ROI constructor
EROIBW8 myROI;
```

```
// ...
```

```
// 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;
```

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

```
// Fill the vector with increasing values
for(int i= 0; i < 128; i++)
{
        ramp.AddElement((EBW8)i);
}
```

```
// Retrieve the 10th element value
EBW8 value= ramp[9];
```

Exception Management

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

```
try
{
        // Image constructor
        EImageC24 srcImage;
```

```
        // ...
```

```
        // Retrieve the pixel value at coordinates (56, 73)
        EC24 value= srcImage.GetPixel(56, 730);
}
```

```
catch(Euresys::Open_eVision_1_1::EException exc)
{
        // Retrieve the exception description
        std::string error = exc.What();
}
```

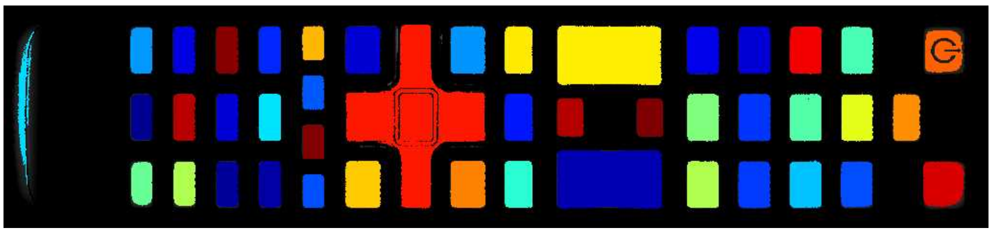
5. Application Examples

5.1. Measuring a Remote Controller

This topic presents a complete 3D processing workflow, featuring a TV remote controller as sample object.

Introduction

The remote controller on the laser triangulation acquisition setup



The remote controller after 3D processing and projection on a 2D image

The proposed process is the sequence of the following operations:

One time calibration process:

1. Load a depth map representing the calibration object.
2. Perform the calibration model computation.
3. Store the calibration model.

For each object:

1. Load the object depth map.
2. Apply the calibration model to get the world point cloud.
3. (Optional) Save the point cloud to a PCD file.
4. Search for a reference plane by either:
   - Choosing 3 points on the remote depth map to be used to compute a plane.
   - Using the `3DPlaneFitter` function.
5. Choose 2 points to define an orientation.
6. Build a ZMap using the reference plane and the orientation vector.
7. (Optional) Save the ZMap as an image.
8. Query the ZMap to get world space measurement.
9. Process the ZMap with 2D image function.

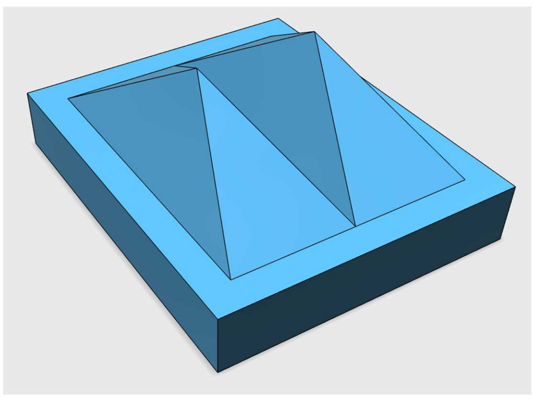
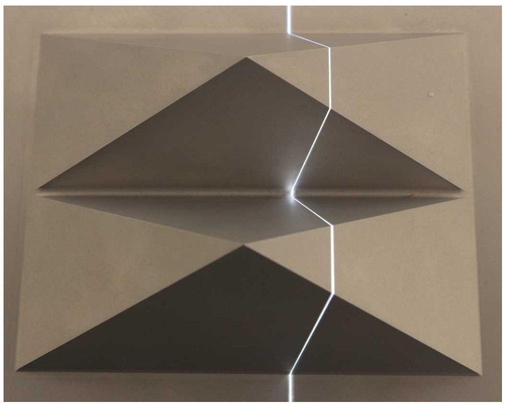
TIP
For easier reading, the code snippets in this example do not show exception catching and error checking.

Calibration

The calibration process is mandatory to find the exact transformation from the depth map to the real world, metric, coordinate system.

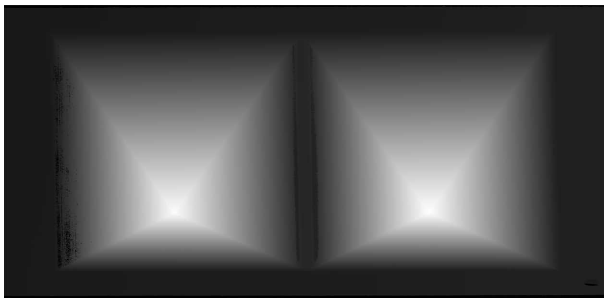
Open eVision features an object based calibration process: the scan of a reference object of known geometry is used in the calibration calculation.

TIP
The recommended calibration object is the double truncated pyramid.

CAD model of the double pyramid and calibration object scanned

Using a **Euresys Coaxlink 3D LLE** frame grabber, the captured image is directly a depth map. Depth maps are 8 bits or 16 bits grayscale images with the pixel values representing the height of the laser profile.



A depth map (gray scale image) of a double pyramid, used as the calibration object

The code snippet below shows how to load the depth map representing the double pyramid object, using it to compute a calibration model and save the result for later use.

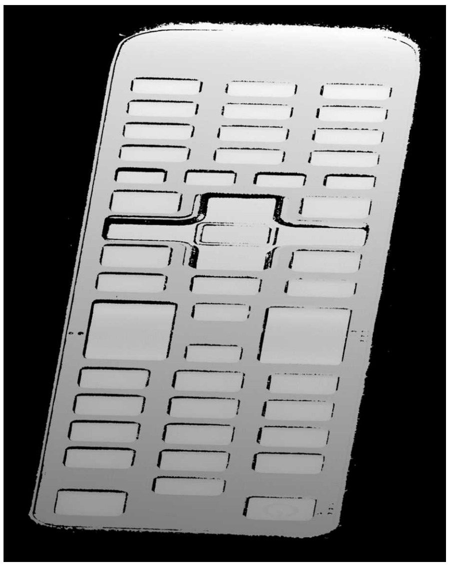
```
  // Load the depthmap used for the calibration process
  EDepthMap16 calibration_depthmap;
  calibration_depthmap.Load("calibration.tiff");
  // Set the Z resolution from the number of bits for the fractional part, depends on
the depth map acquisition
  calibration_depthmap.SetZResolution(1.f / (1<<5));

  // Declare the object based calibration generator
  EObjectBasedCalibrationGenerator calibrator;
  // set the real world scale of the scanned object
  calibrator.SetCalibrationObjectType(EObjectBasedCalibrationType_DoublePyramid);
  calibrator.SetCalibrationObjectScale(10.f);

  // Declare an object based calibration model
  EObjectBasedCalibrationModel calibration_model;

  // Perform the calibration process (can take some time, like 10 seconds)
  calibration_model = calibrator.Compute(calibration_depthmap);

  // Check the calibration result
  if(calibration_model.IsInitialized())
    {
      printf("Calibration succeeded with score: %g\n", calibration_
model.GetCalibrationError());
      // Save the model for later use
      ESerializer* serializer = ESerializer::CreateFileWriter("calibration.model");
      calibration_model.Save(serializer);
      delete serializer;
    }
    else
    {
      printf("Calibration failed\n");
    }
```

Processing the object in 3D

This section exposes the 3D workflow, from the source depth map to metric measurement.

Acquiring and calculating the 3D point cloud

The calibration model previously calculated is used to transform the depth map data to real world 3D point cloud.



A depth map of the TV remote controller
(the object is distorted and scaled while black pixels represent undefined regions, that is parts of the object that were not seen by the camera or lit by the laser)

The code snippet below:

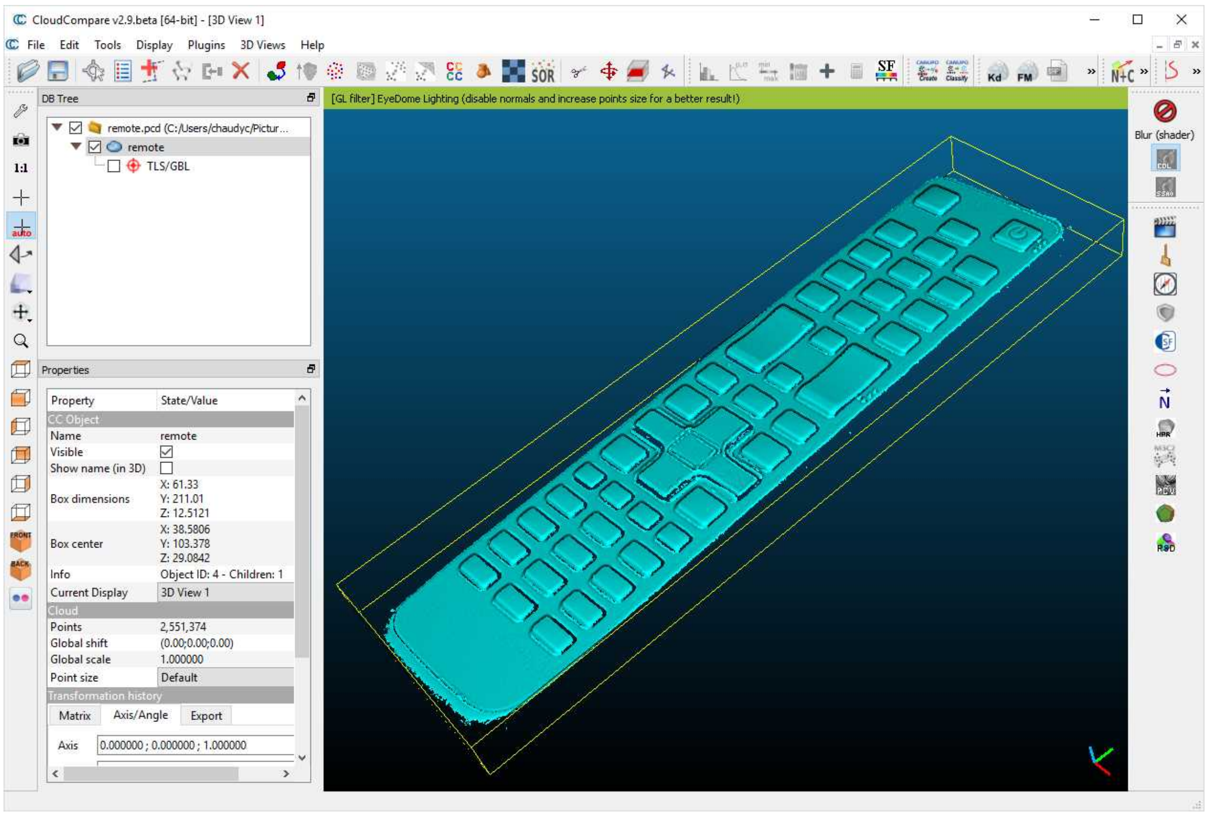
1. Loads a calibration model.
2. Transforms the depth map to a 3D point cloud.
3. Saves the point cloud to a PCD file.

```
// Read an abstract calibration model from file
ECalibrationModel* calibration_model;
ESerializer* serializer = ESerializer::CreateFileReader("calibration.model");
// must be desallocated later
calibration_model = ECalibrationModel::Create(serializer);
delete serializer;

// Declare a depth map to point cloud converter
EDepthMapToPointCloudConverter dm2pc;
// attach a calibration model to the converter
dm2pc.SetCalibrationModel(*calibration_model);

// Generate the point cloud
EPointCloud point_cloud;
dm2pc.Convert(object_depthmap, point_cloud);

printf("Point cloud size : %d\n", point_cloud.GetNumPoints());

// Save to point cloud to a PCD file
point_cloud.SavePCD("point_cloud.pcd");
```

TIP

The PCD file is a simple 3D point container, used by the PCL framework (www.pointclouds.org). Such file can be loaded in the PCL viewer of other tools like Cloud Compare (www.cloudcompare.org).

As shown on the screen shots below, data in point cloud are in world coordinate system, expressed in the calibration object units. Distances and angles are correct and then metric measures are possible. However, processing on point clouds can be difficult and costly and then the ZMap representation is an alternative allowing 2D processing on metric world space.

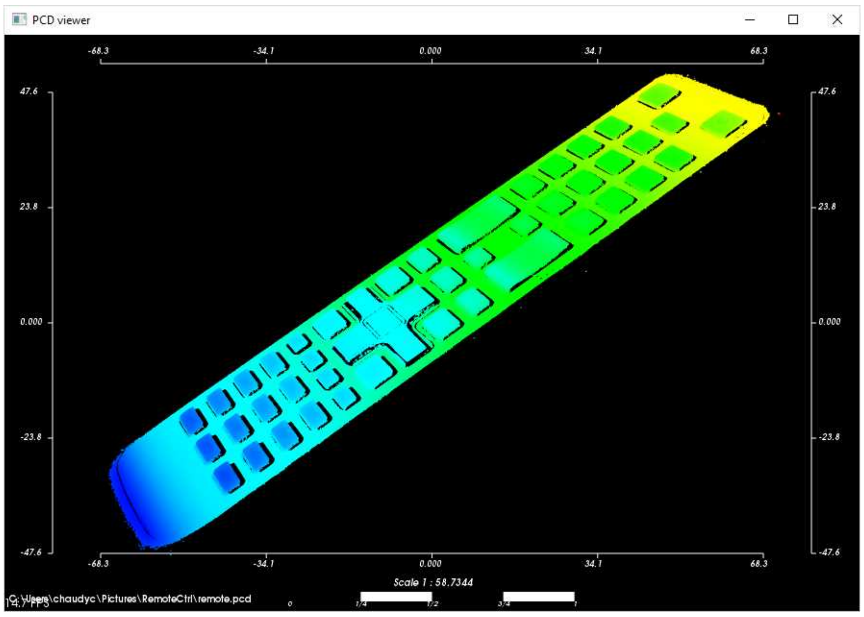

CloudCompare v2.9.beta [64-bit] - [3D View 1]
File Edit Tools Display Plugins 3D Views Help
DB Tree
remote.pcd (C:/Users/chaudyc/Pictur...
remote
TLS/GBL
Properties
[GL filter] EyeDome Lighting (disable normals and increase points size for a better result!)
Blur (shader)

Property	State/Value
CC Object	
Name	remote
Visible	☑
Show name (in 3D)	☐
Box dimensions	X: 61.33 Y: 211.01 Z: 12.5121
Box center	X: 38.5806 Y: 103.378 Z: 29.0842
Info	Object ID: 4 - Children: 1
Current Display	3D View 1
Cloud	
Points	2,551,374
Global shift	(0.00;0.00;0.00)
Global scale	1.000000
Point size	Default
Transformation history	

Matrix | Axis/Angle | Export

Axis 0.000000 ; 0.000000 ; 1.000000

The resulting point cloud viewed in the Cloud Compare application



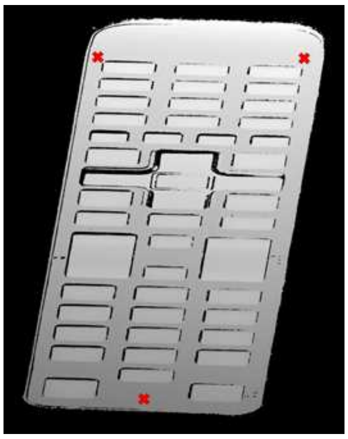
The resulting point cloud in the PCL viewer; the color ramp shows that the main body plane of the remote is not aligned with an axis

Searching for a reference plane

To perform measurements and processing, it is usually mandatory to transform the data to a reference frame. For the remote controller, we want to level to the main plane supporting the keys and orient the points along the remote edge.

If the depth map has been previously registered (by a consistent acquisition process or by the detection of fiducial markers), it may be possible to build the reference plane from 3 chosen points (points to be known as part of the reference plane).

```
  // Use 3 known points to build the reference plane
  E3DPoint p1(600.5, 450.5, object_depthmap.GetZValue(600, 450));
  E3DPoint p2(1700.5, 470.5, object_depthmap.GetZValue(1700, 470));
  E3DPoint p3(840.5, 2300.5, object_depthmap.GetZValue(840, 2300));

  // convert these points from depth map space to world space, using the calibration
model
  E3DPoint w1, w2, w3;
  w1 = calibration_model.Apply(p1);
  w2 = calibration_model.Apply(p2);
  w3 = calibration_model.Apply(p3);

  // build the world plane using 3 points
  E3DPlane reference_plane(w1, w2, w3);
```

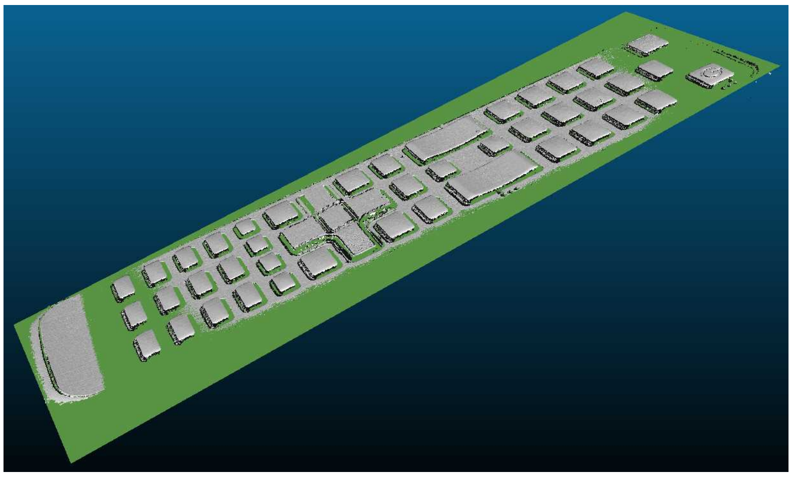


The original object depth map and the position of 3 points used for the reference plane calculation

Another option, if the depth map is not registered, is to use the `E3DPlaneFitter` class. This function tries to find the main plane from a point cloud using a probabilistic approach. It finds a subset of the point cloud that lies on a plane, given a user defined threshold tolerance.

The distance tolerance is a parameter of `E3DPlaneFitter` and must be adapted depending on the scale, noise and curvature of the plane in the point cloud.

```
// Use a E3DPlaneFinder with a distance tolerance of 0.1 (world space coordinate)
float distance_tolerance=0.1f;
EPlaneFinder plane_finder(distance_tolerance);
E3DPlane reference_plane=plane_finder.Find(point_cloud);
```



The principal plane (in green) extracted from the point cloud by `E3DPlaneFinder` class

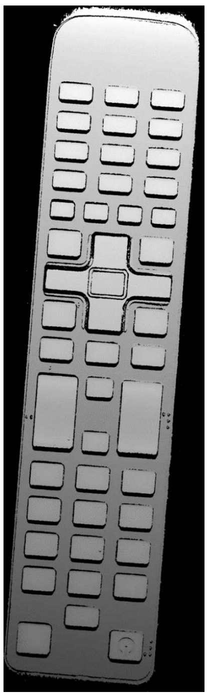
Building the ZMap

The ZMap is a gray scale 2D image, representing the projection of the 3D points to a reference plane. The value of the pixels of the ZMap is the distance of the 3D point to the reference plane, coded in a fixed point representation.

The ZMap also supports the “undefined pixel” specific value, when no point is projected on a pixel and there is no valid value at that position.

As shown below, the point cloud to ZMap conversion can be made using default values for all parameters: resolution and scale of the ZMap, reference plane, orientation, origin will all be chosen automatically.

```
// Create the converter
EPointCloudToZMapConverter pc2zmap;

// Create a 16 bits ZMap and fill it with point cloud points
EZMap16 zmap;
pc2zmap.Convert(point_cloud, zmap);

// Save the ZMap as a PNG image
zmap.SaveImage("zmap.png");
```

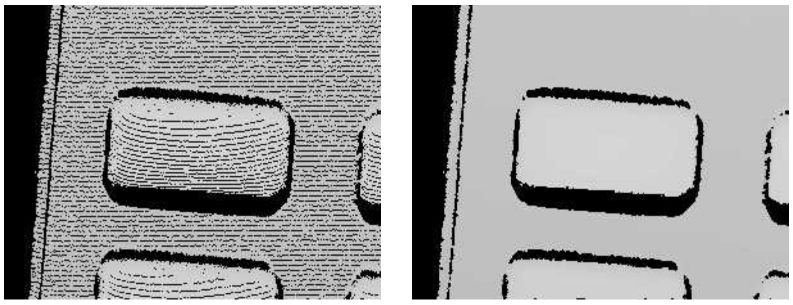


The generated depth map with default parameters

The body of the remote is not leveled and the object is not aligned. Nevertheless, comparing to the depth map, the ZMap is a calibrated representation of the object. Metric distances can be evaluated on the ZMap.

The current implementation of the ZMap converter simply projects 3D points on the ZMap image. Thus, depending on the point density and projection parameters, undefined pixels and region may appear in the ZMap.

The ZMap converter automatically perform a filling algorithm on undefined pixels. Disable it with the `EnableFillMode(false)`.

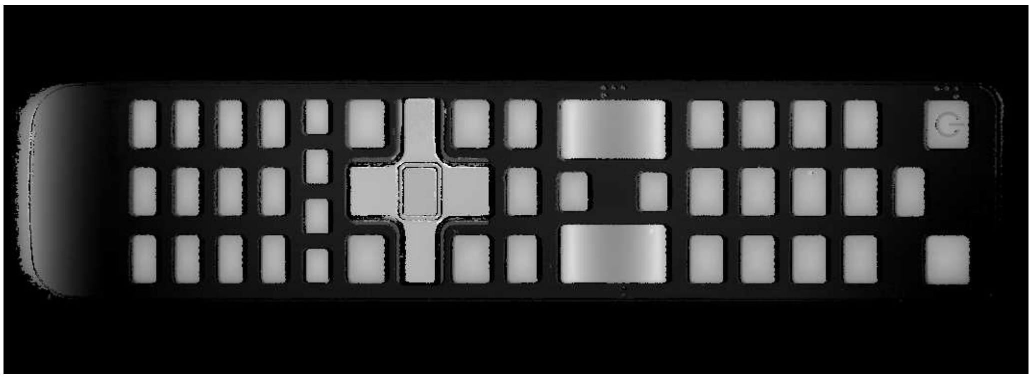


`EnableFillMode(false)` and `EnableFillMode(true)`

To avoid undefined pixels, choose a target scale for the ZMap. The method `SetScale()` changes the target X and Y resolution (in metric unit per pixel).

With the default configuration, the ZMap generator uses a horizontal reference plane. Use the `SetReferencePlane()` method to “level” the object and use the main body as the reference plane. `SetOrientationVector()` specifies the direction of the X (width) axis of the ZMap. The orientation vector allow to “rotate” the object around the reference plane normal.

```
// level the object by defining a reference plane
pc2zmap.SetReferencePlane(reference_plane);

// align to the world Y axis
pc2zmap.SetOrientationVector(E3DPoint(-0.07501, 0.9964, -0.03761));

// choose a resolution of 0.2mm per pixel
pc2zmap.SetMapXYResolution(0.2f);

// generate the ZMap
pc2zmap.Convert(point_cloud, zmap);
```

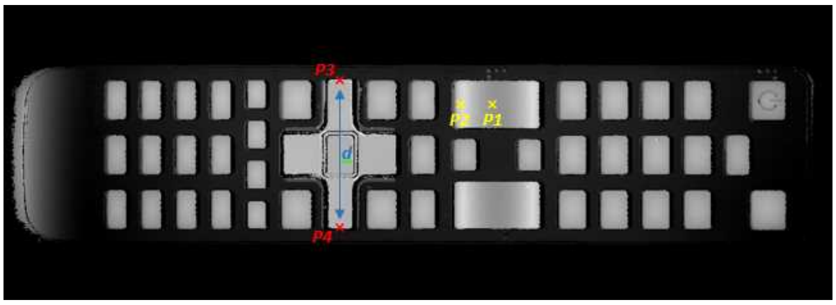


The ZMap with a reference plane previously calculated, an orientation to align the object and a reduced resolution

Use queries on ZMap to retrieve metric coordinates (for example, to measure the size and/or the height of a feature). Useful functions are `GetWorldPositionFromPixelPosition(x,y)` and `GetZValue(x,y)`.

```
float h1 = zmap.GetZValue(638, 128);        // get “height” at position P1
float h2 = zmap.GetZValue(595, 128);        // get “height” at position P2
```

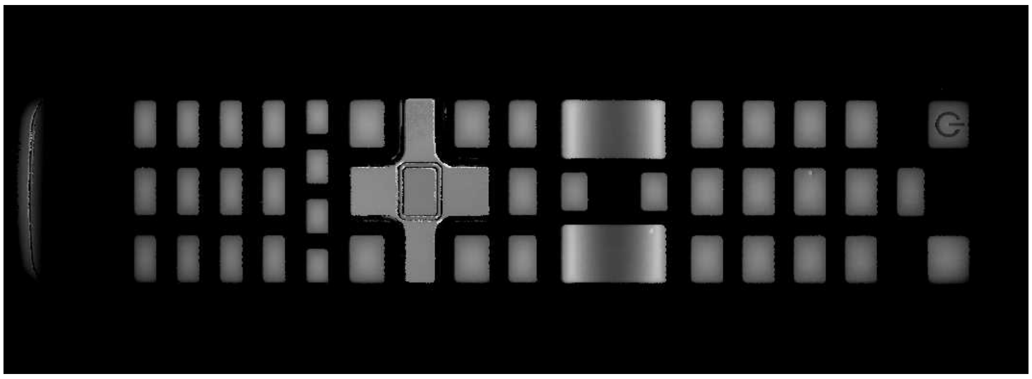
```
  E3DPoint p3 = zmap.GetWorldPositionFromPixelPosition(437, 98);  // get world
position at p3
  E3DPoint p4 = zmap.GetWorldPositionFromPixelPosition(437, 288); // get world
position at p4
  float d = p3.DistanceTo(p4);                          // world distance between p3 and p4
```



- h1 (1.15601) and h2 (1.83618) are distance above the ZMap reference plane. These are values in millimeters, the difference evaluates the “curvature” of the key.
- P1 (58.843, 84.7838, 32.1084) and P2 (20.9479,81.9271, 32.0041) are positions in the original 3D world space.
- The distance d (38.0028) represents the width of the remote keyboard in millimeters.

The reference plane can be shifted (translated) to remove the remote controller body and keep only the keys in the ZMap. An 8 bits ZMap must be used to be compatible the other Open eVision 2D libraries.

```
  // shift the plane by 1mm in the normal direction
  float d = reference_plane.GetSignedDistanceFromOrigin();
  reference_plane.SetSignedDistanceFromOrigin(d + 1.f);
  pc2zmap.SetReferencePlane(reference_plane);
  // choose the scale for the Z axis (2mm for 256 grey scale values)
  pc2zmap.SetMapZResolution(2.f / 256);
  // convert the point cloud to a 8 bits ZMap
  EZMap8 zmap8;
  pc2zmap.Convert(point_cloud, zmap8);
```



ZMap with shifted reference plane: only the keys remain visible while other pixels are set to 0 (undefined value)

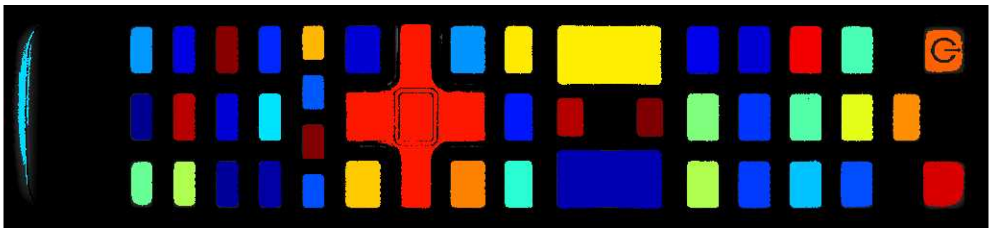
TIP
This image can be used in 2D libraries like EasyImage, EasyObject or EasyGauge.

Processing the ZMap

On a ZMap, the gray value of a pixel is the distance (height) above the reference plane. Threshold, filters, morphology and other image operators can be used directly.

Here is an example of a region segmentation using EasyObject:

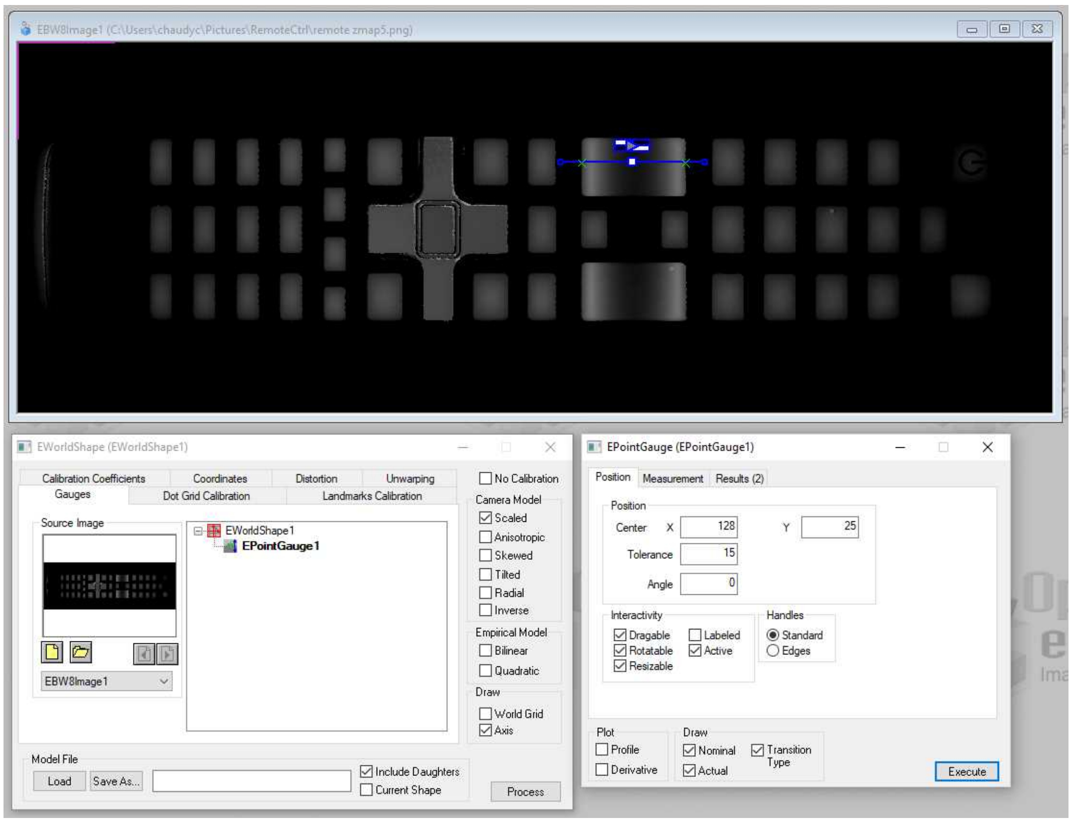
```
  // segment the objects of the ZMap (default is Minimum Residue segmentation method)
  ECodedImage2 coded_image;
  EImageEncoder image_encoder;
  image_encoder.Encode(zmap8.AsEImage(), coded_image);

  // filter the object by area
  EObjectSelection object_selection;
  object_selection.AddObjects(coded_image);
  object_selection.RemoveUsingUnsignedIntegerFeature(EFeature_Area, 100,
ESingleThresholdMode_Less);
```



The extracted objects with EasyObject on the ZMap image

The ZMap includes a `EWorldShape` object that you can use for EasyGauge measurement. The `EWorldShape` class represents the scale between the image space and the world space.

```
  // get the world shape from the ZMap
  const EWorldShape& world_shape = zmap8.GetWorldShape();

  // setup a point gauge using that world shape
  EPointGauge pointGauge;
  pointGauge.Attach(&world_shape);

  // set gauge center point and tolerances in world space (mm)
  pointGauge.SetCenterXY(128.f, 25.f);  // 128mm and 25mm from the upper left corner
  pointGauge.SetTolerances(15.f, 0.f);  // 15mm, half gauge size

  // perform the measurement on the ZMap
  pointGauge.Measure(&zmap8.AsEImage());

  // get the 2 points and calculate the length of the key, values are in millimeters
  EPoint p1 = pointGauge.GetMeasuredPoint(0);
  EPoint p2 = pointGauge.GetMeasuredPoint(1);
  float length = p1.Distance(p2);      // return 21.619mm
```



EBW8Image1

No Calibration ☐
Camera Model: Scaled ✓, Anisotropic ☐, Skewed ☐, Tilted ☐, Radial ☐, Inverse ☐
Empirical Model: Bilinear ☐, Quadratic ☐
Draw: World Grid ☐, Axis ✓
Include Daughters ✓
Current Shape ☐

Position: Center X 128, Y 25; Tolerance 15; Angle 0
Interactivity: Dragable ✓, Labeled ☐, Rotatable ✓, Active ✓, Resizable ✓
Handles: [✓ Standard] Edges
Plot: Profile ☐, Derivative ☐
Draw: Nominal ✓, Actual ✓, Transition Type ☐

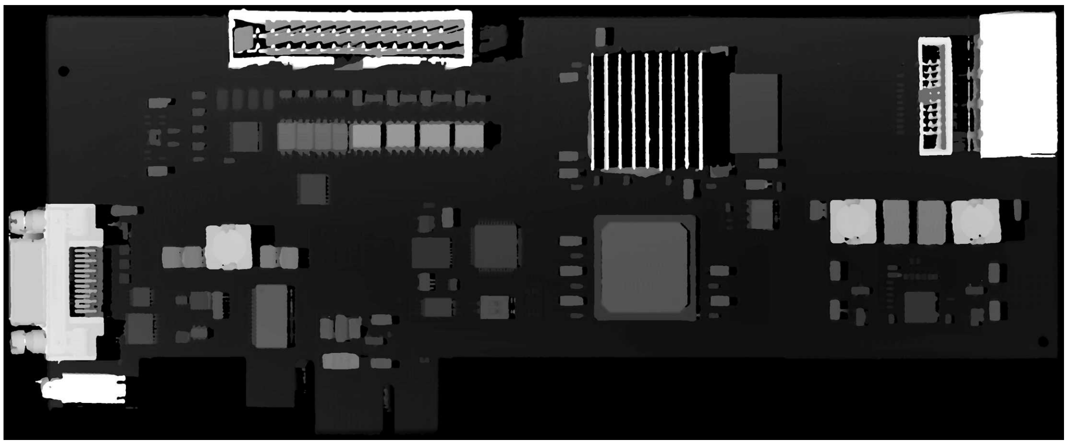
A point gauge on a ZMap in Open eVision Studio: the parameters are the same as the code snippet above

5.2. Inspecting a PCB

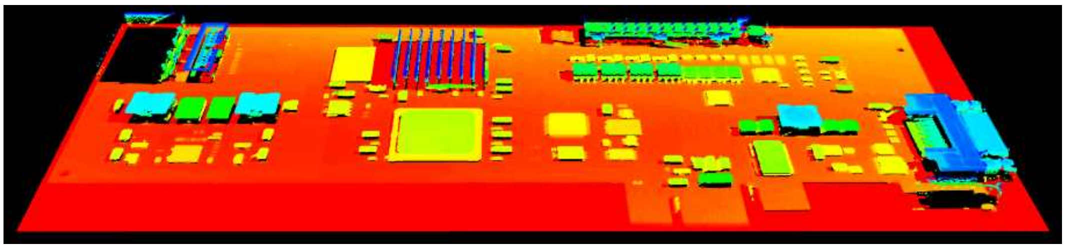
With Easy3D, it is possible to use depth maps for PCB inspection. This section presents a simple detection of missing or misplaced components on a PCB. The processing is done entirely with 2D images but use depth maps as inputs.

The workflow is as follow:

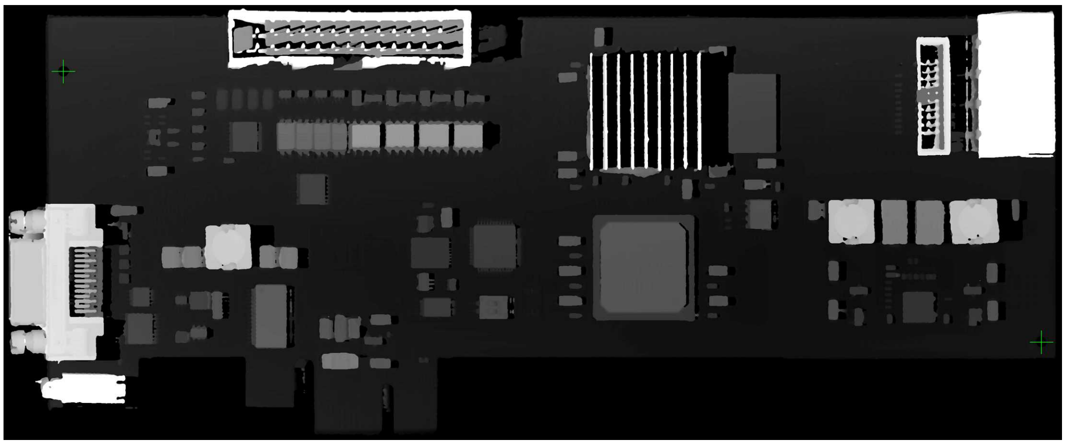
1. Perform a 3D acquisition or create the depth map with software laser line extraction (`ELaserLineExtractor` class). Retrieve the grayscale image corresponding to the depth map (`EDepthMapROI8.AsEImageBW8()` method).
2. Align the image using fiducial markers (`EMatcher` class).
3. Search for the PCB plane and subtract it from the aligned image, only the components and the connectors remain (`EasyImage::Oper(EArithmeticLogicOperation_Subtract...)` function).
4. Compare the processed image to a golden sample to detect missing or misplaced components (`EasyImage::Oper(EArithmeticLogicOperation_Compare...)` or `EChecker`).



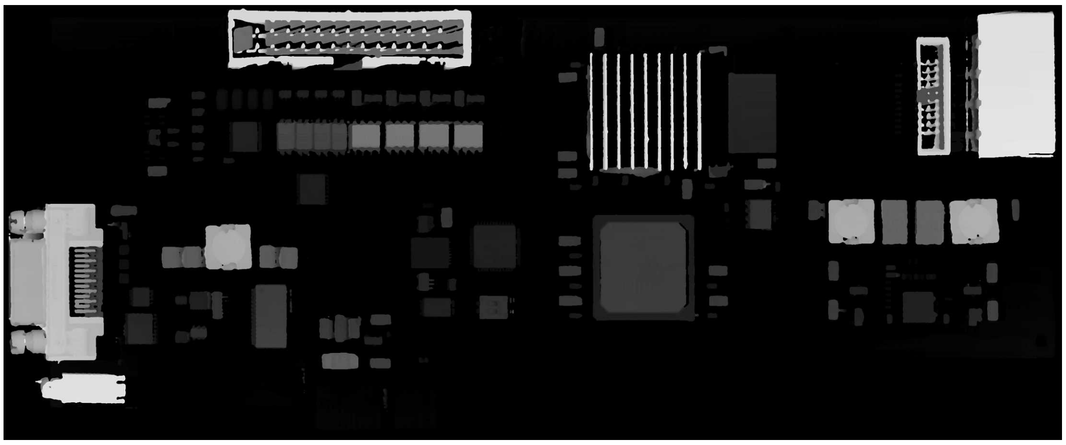
The source depth map of a PCB, outputs of CoaxLink Quad 3D-LLE

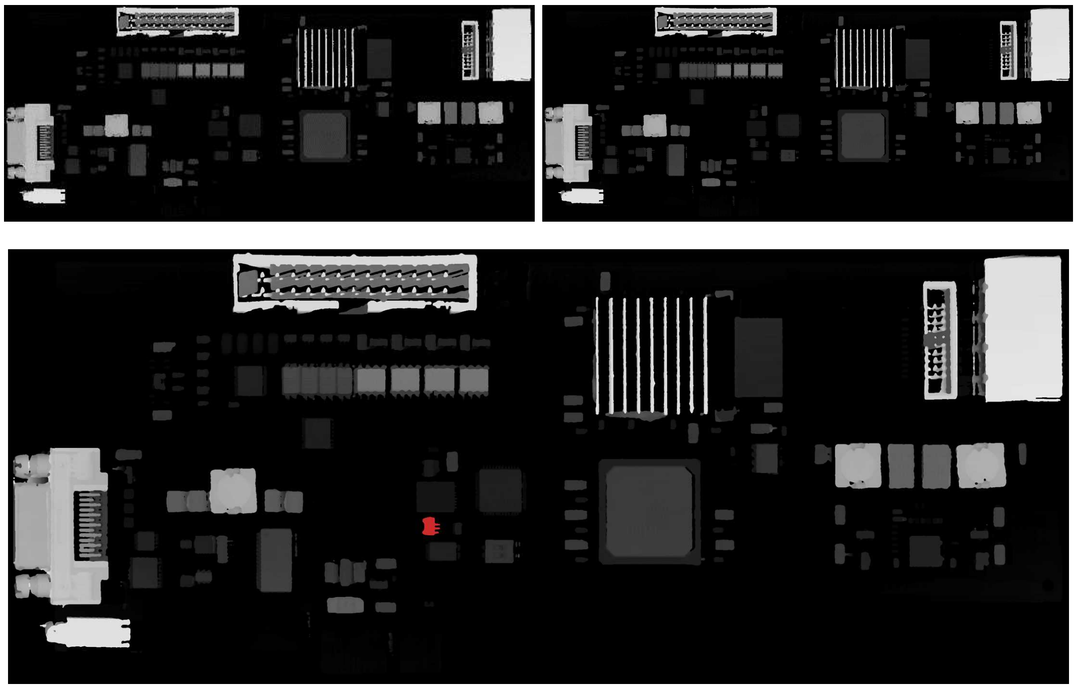


The same depth map displayed as a 3D point cloud with false colors



Align the image using fiducial markers (2 holes)


The image with reference plane subtracted, leaving only the components


The comparison of the image (left) with the golden sample (right, processed with the same workflow) shows the missing component (in red)

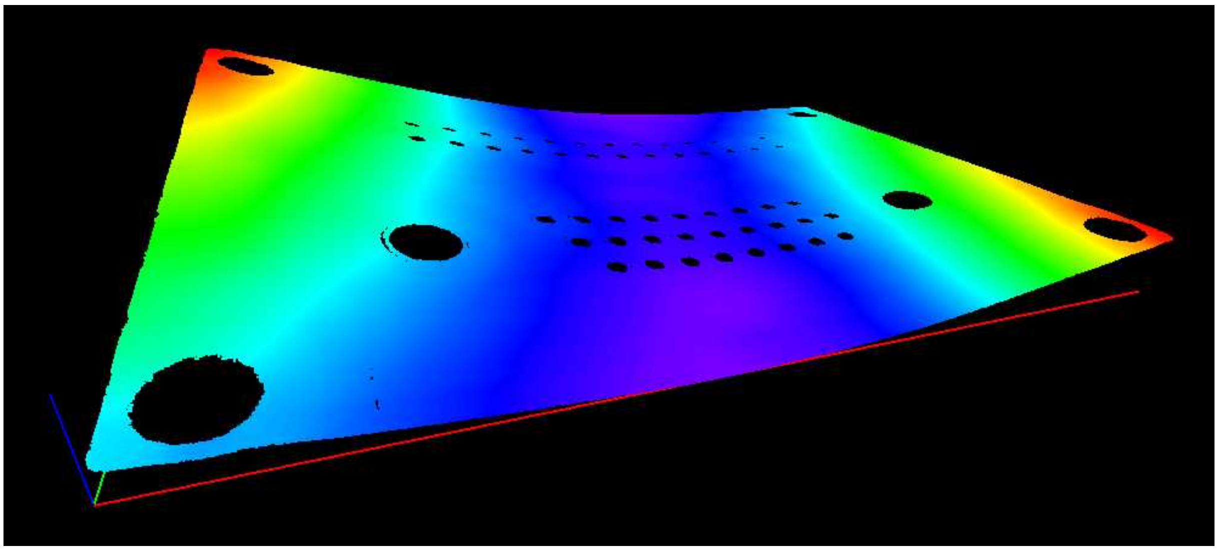
5.3. Measuring the Warpage of a PCB

Source code and images

This application example is provided as a C++ sample program, named `3DPCBWarpage`:

- The source code is located in `Sample Programs\MsVc samples\3DPCBWarpage`.
- The sample images are located in `Sample Images\Easy3D\PCB Warpage`.

NOTE
To run this program, you need the Easy3D and EasyGauge licenses.



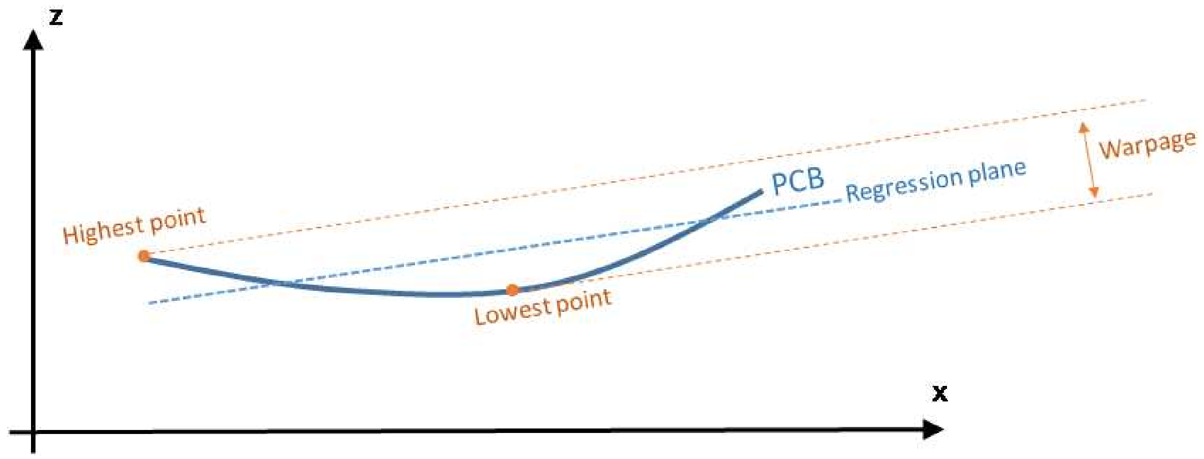
Display output of the 3D PCB Warpage application example

NOTE
The calibration process is not described in this document. We assume that a calibration has been done and that the metric unit used is the mm.

Application objective

This application demonstrates how to measure the warpage of a PCB.

TIP
In this application, we define the warpage as the difference between the highest and the lowest surface points when the PCB regression plane is oriented horizontally. In other words, we fit a plane through the PCB points and take the perpendicular distance between the highest point above and the lowest point below this regression plane.



The main steps of the process are:

1. Making an initial ZMap using a first estimation of an average PCB plane.
2. Filtering the data by:
   - Removing the points that show a large height deviation with respect to their neighbors.
   - Applying a smooth filter on the remaining points.

   For the generation of the final ZMap, the application will fit a regression plane through the remaining / filtered data.
3. On the ZMap, detecting the corners of the PCB for the alignment (optional).
4. Producing the final ZMap that the application will use to compute the warpage of the PCB.

TIP
The (optional) alignment on the PCB edges ensures that the resulting ZMap is always the same, independently of the orientation of the PCB during the measurement.

TIP
Moreover, on the final ZMap, the horizon is parallel with the PCB regression plane. This is necessary to have a well-defined warpage value.

Generating the initial ZMap

1. Using a calibration model (`model_` in the code) and a depth map (« `dm_` »), the application produces a point cloud (« `pc_` »).

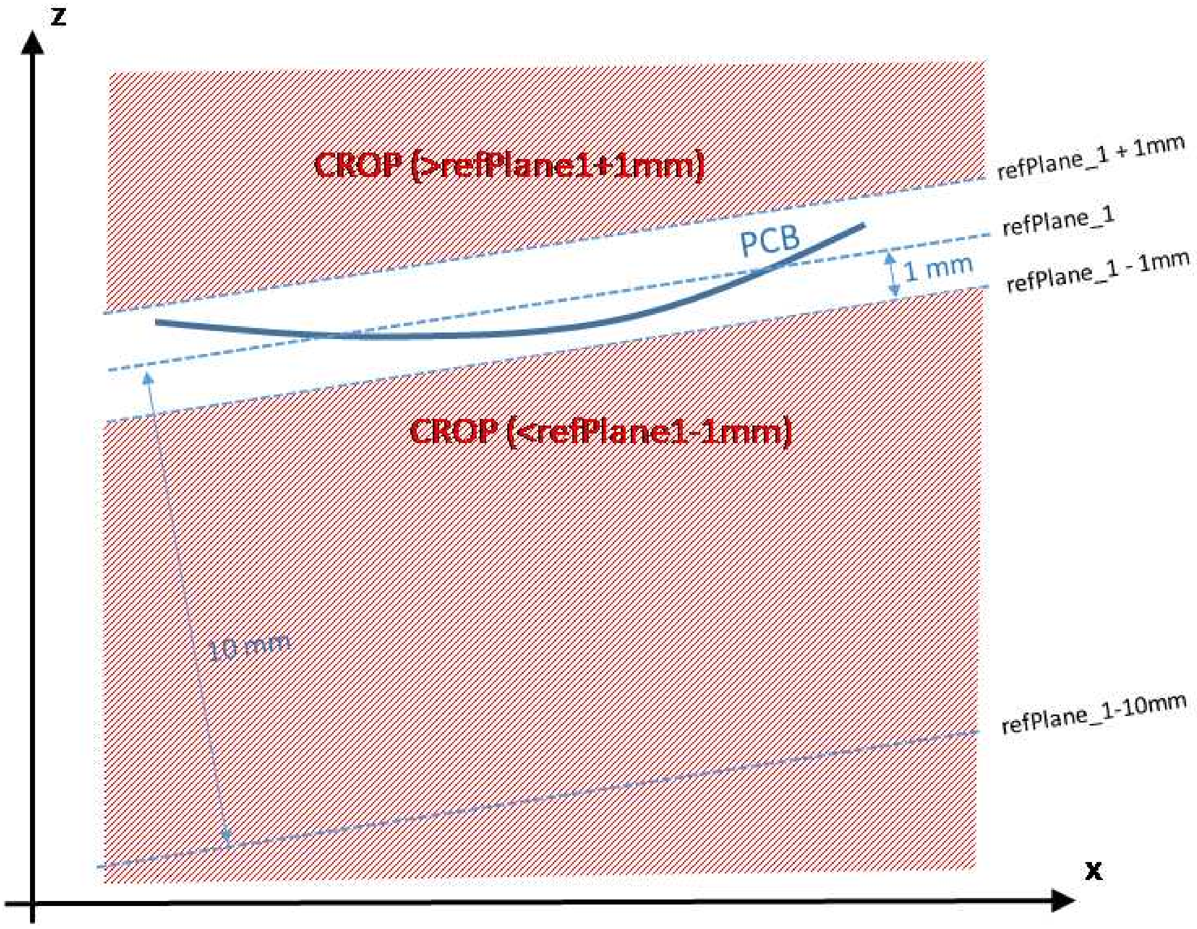
TIP
Alternatively, you can load directly a point cloud from a file.

2. In this point cloud, the application localizes the PCB plane by searching for the largest plane, using a plane finder object of the type `E3DPlaneFinder`. As we want to handle a curved PCB, a large tolerance (+/- 1mm) is used by the plane finder object.

This step produces a reference plane called `refPlane_1` of the type `E3DPlane`.

3. The application crops any data point that is farther from the reference plane than 1 mm.

4. It uses `refPlane_1` » to generate the first ZMap (called `zmapBeforeAlignment`). Actually, the reference plane used for the ZMap generation is 10 mm below `refPlane_1` so that all pixel values of the ZMap are positive. This is illustrated on the figure below.

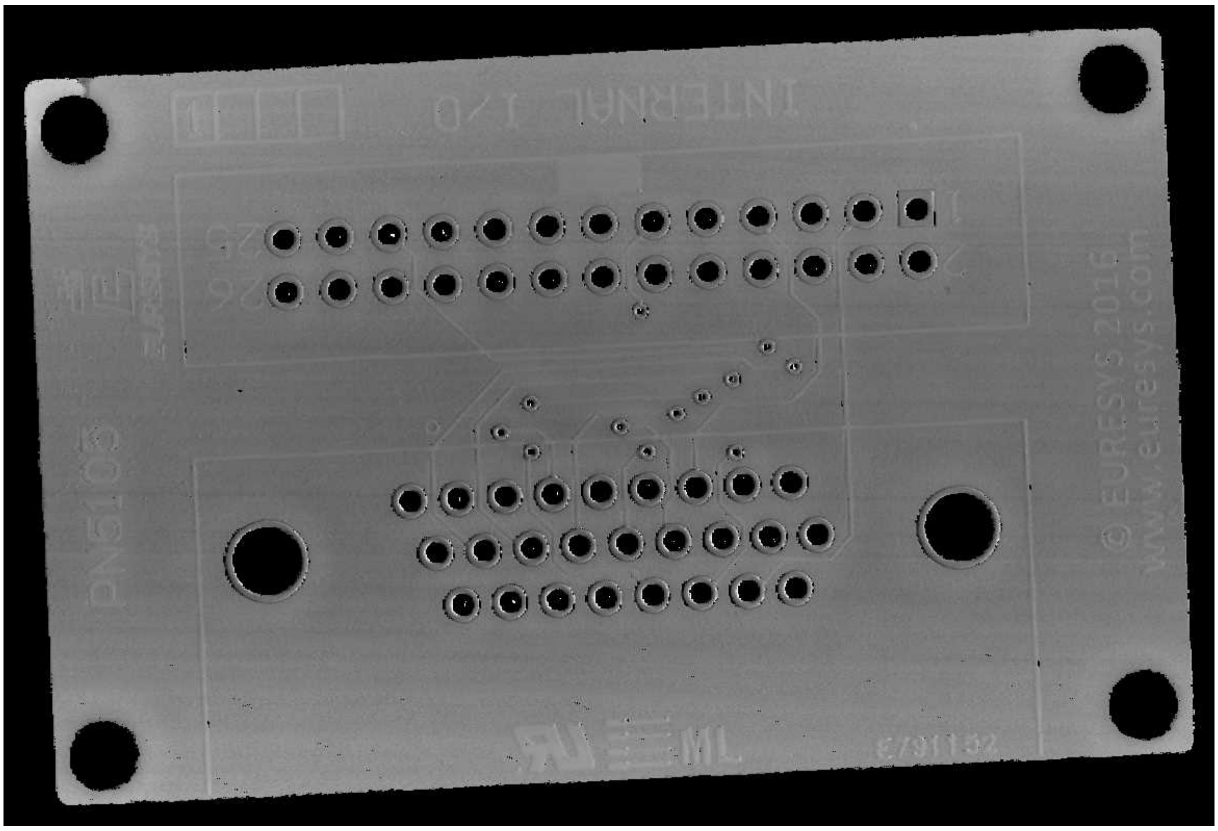


The code to produce the ZMap `zmapBeforeAlignment` is shown below.

```
// Apply calibration to the depth map, a metric 3D point cloud is generated
E3DDepthMapToPointCloudConverter converter;
converter.SetCalibrationModel(model_);
converter.Convert(dm_, pc_);

// Search for the PCB plane
float maximumDistanceToPlane = 1.0f; // 1.0 mm
E3DPlaneFinder finder(maximumDistanceToPlane); // tolerance for plane search = 1.0 mm
E3DPlane refPlane_1 = finder.Find(pc_); // finds the largest plane in the point cloud

// Crop any point distant from PCB plane
E3DPlaneCropper cropperPCB(refPlane_1); // 'refPlane_1' = ref. plane for the cropper
E3DPointCloud pcPlaneOnly; // only keep points close to the ref. plane
cropperPCB.Crop(pc_, pcPlaneOnly, EPlaneCropperType_KeepClose, maximumDistanceToPlane);
```

```
// ZMap projection of PCB plane, using reference plane and fixed resolution
float zMapPixelSize = 0.050f; // ZMap horizontal resolution: 1 pixel = 50µm = 0.050 mm
float zMapVerticalResolution = 0.001f; // ZMap vertical resol.: 1 gray value = 1µm (= 0.001 mm)
float zMapOffset = 10.0f; // 10 mm
E3DZMapGenerator zmapGenerator;
EZMap16 zmapBeforeAlignment;
zmapGenerator.SetScale(zMapPixelSize); // horizontal resolution = 50µm/pixel
zmapGenerator.SetZScale(zMapVerticalResolution); // vertical resolution = 1 µm/GV
zmapGenerator.SetReferencePlane(refPlane_1 - zMapOffset); // 10mm below refPlane_1
zmapGenerator.SetOrientationVectorMode(EZMapOrientationVectorMode_XAxis);
zmapGenerator.SetExtension(1.f);  // add a 1mm 3D extension, create a border around the ZMap
zmapGenerator.Convert(pcPlaneOnly, zmapBeforeAlignment);
int zmapWidth = zmapBeforeAlignment.GetWidth();
int zmapHeight = zmapBeforeAlignment.GetHeight();
```

The following image shows this first ZMap with a 16-bit per pixel resolution.



ZMap `zmapBeforeAlignement`

In this image:

- Background pixels have the value "0" (black). This value is reserved for “undefined pixels” (nothing detected). Pixels having a value different from zero are “valid pixels”.
- The horizontal resolution is set to 50 µm/pixel and the vertical resolution is set to 1 µm for one gray value.
  Because the reference plane of the ZMap is 10 mm below the PCB, the average gray value in the image is 10000 (10000 gray values = 10000 µm).
- The size of this ZMap is not specified; it is automatically computed from the size of the bounding box, enlarged by 1 mm so that there is a black border around the PCB. This enlargement is specified by the method `SetExtension` of the ZMap generator. The black border around the PCB helps for the detection of the edges.

Reducing and filtering the noise

On the ZMap, the application should ignore the following points during the computation of the warpage:

- Isolated noisy points on the background.
- Noisy points inside the holes or on the pads.

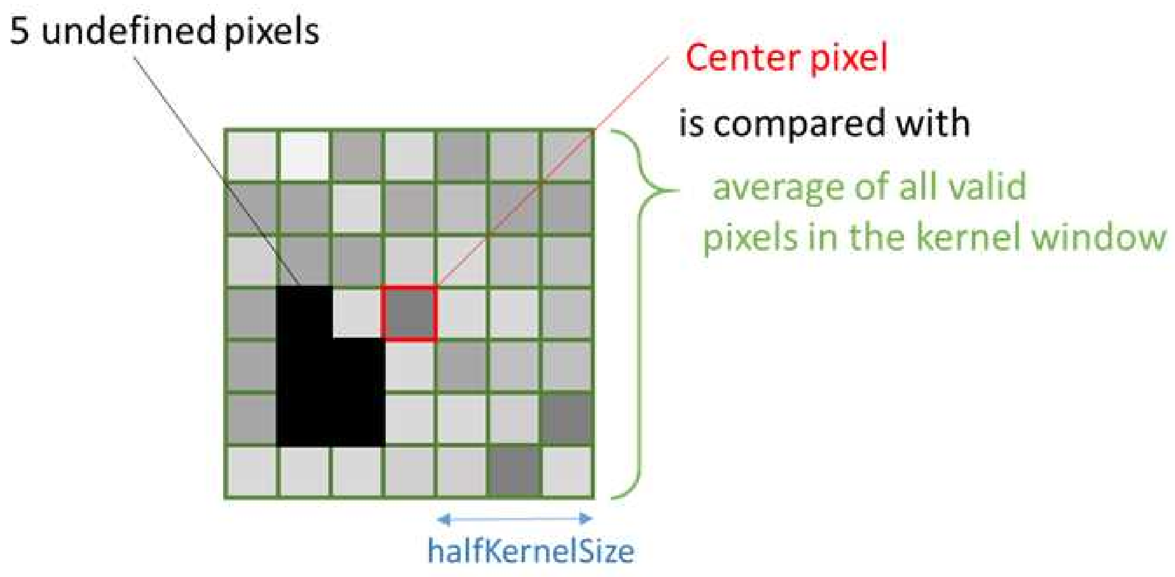
The application removes points and applies a low-pass filter on the remaining points to detect only the slow / global PCB deformations. This filtering is done in 2 steps.

Filter 1: removing the outliers

1. The filter `RemoveNoise` defines a kernel window that is moved over every pixel of the ZMap.
2. For each pixel of the ZMap image, the kernel window is centered on this pixel and the filter condition determines if the center pixel is either kept or removed.

   The filter condition used here is `E3DNoiseRemovalMethod_AbsoluteDifferenceFromMean`; it compares the value of the pixel in the center with the average of all valid pixels in the kernel window.

This is illustrated on the figure below.



The application evaluates the condition to keep or remove the center pixel as follow :

- If a window has less than 25% of defined pixels (see `ratioValidPixelsFilter` in the code) or if the center pixel is already undefined, this center pixel is marked as undefined.
- The application computes the height deviation of the center point with respect to all the valid points in the kernel window and, if the resulting height deviation exceeds 30 µm (see `thresholdFilter` in the code), the application removes the pixel in the center of the kernel window (it actually replaces it by 0, meaning « undefined pixel »).

Filter 2: applying an averaging filter

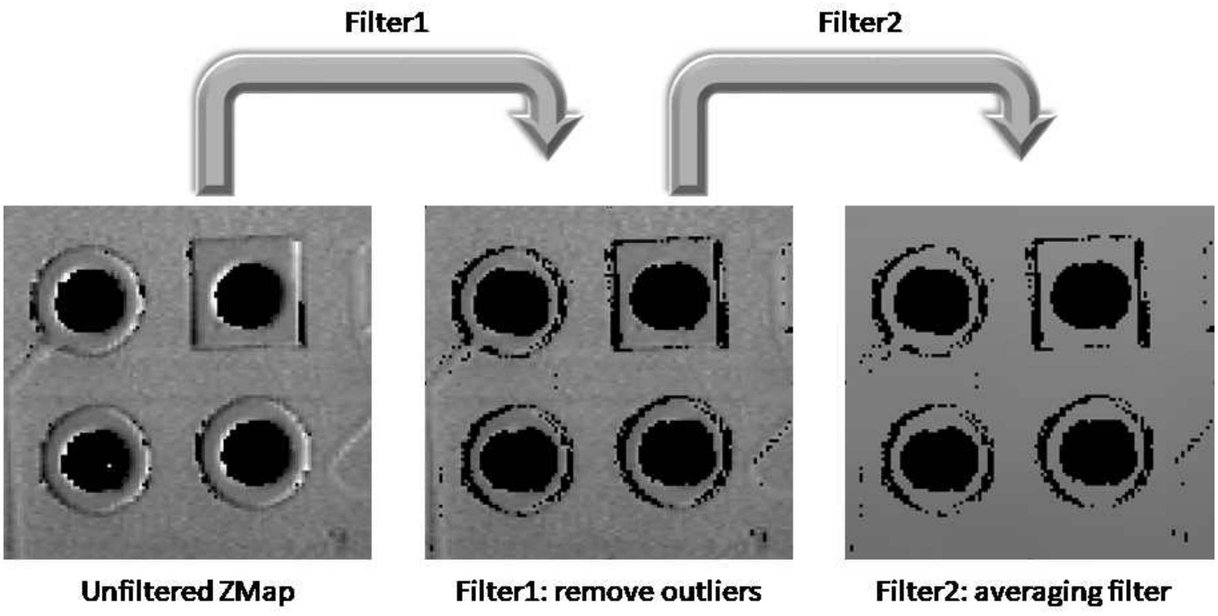
1. In a second step, the application applies an averaging filter (low-pass) in order to remove the random noise.
2. The filter `ComputeAverageMap` also defines a kernel window that is moved over every pixel.
3. If there are enough valid pixels within the window (at least 25% in this case) and if the center pixel is valid, this center pixel is replaced by the average of all pixels within the kernel window.

Filtering code

The code used for the filtering of the ZMap data is:

```
// Process the ZMap to remove noise and small scale structures
// Filters parameters
int halfKernelSizeFilter = 25; // kernel size = 2 x halfKernelSizeFilter + 1 pixel = 51
float thresholdFilter = 0.030f; // threshold = maximum deviation from mean
float ratioValidPixelsFilter = 0.25f; // requires at least 25% of valid points in the kernel

// Apply noise removal filter
EZMap16 zmapBeforeAlignmentFilter1; // output of the filter
zmapBeforeAlignmentFilter1.SetSize(zmapBeforeAlignment);
Easy3D::RemoveNoise(zmapBeforeAlignment, zmapBeforeAlignmentFilter1,
                    E3DNoiseRemovalMethod_AbsoluteDifferenceFromMean,
                    halfKernelSizeFilter, thresholdFilter, ratioValidPixelsFilter, false);

// Apply low-pass filter
EZMap16 zmapBeforeAlignmentFilter2; // output of the second filter
zmapBeforeAlignmentFilter2.SetSize(zmapBeforeAlignment);
Easy3D::ComputeAverageMap(zmapBeforeAlignmentFilter1, zmapBeforeAlignmentFilter2,
                          halfKernelSizeFilter, ratioValidPixelsFilter);
```

The different steps of the filtering are illustrated below:



The 2 steps of the ZMap filtering

Aligning the ZMap on the PCB edges (optional)

This optional step consists in finding the orientation of the PCB (detection of a rectangle).

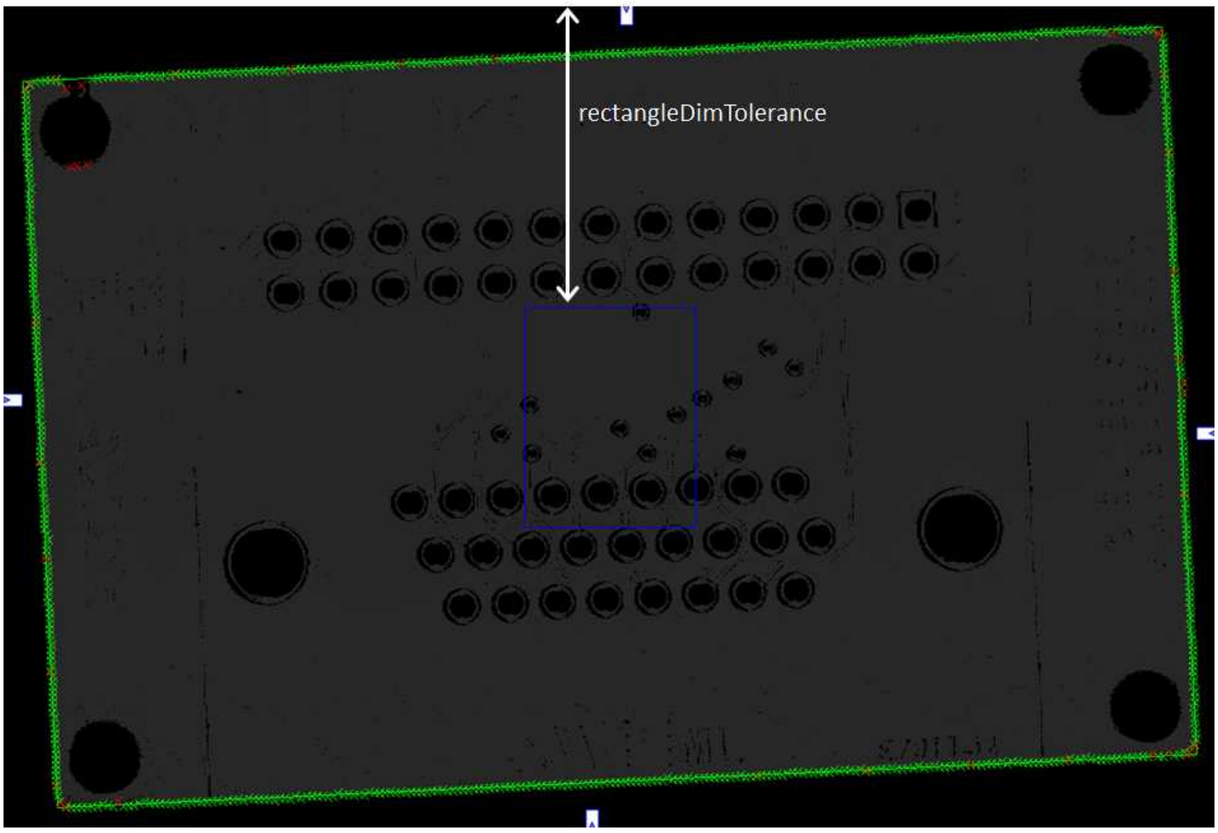
TIP
We could skip this step for the computation of the warpage but aligning on the edges makes the comparison of different ZMaps with each other easier.

To generate a ZMap aligned on the PCB edges:

1. The application converts the ZMap to an 8-bit image. This is necessary to use the 2D tool set of Open eVision.
2. Using EasyGauge, it detects the edges of the PCB by fitting a rectangle on the ZMap image.

The application uses the transitions between the undefined (0) and the valid (non-zero) pixels to detect the position of the edges of the rectangle.

The use of a rectangular gauge on the ZMap image is illustrated below:



The corresponding code is:

```
// From the ZMap create an 8-bit image for alignment
EImageBW8 imageBeforeAlignment8(zmapWidth, zmapHeight);
// conversion 16-bits to 8-bits
EasyImage::Convert(&zmapBeforeAlignmentFilter2.AsEImage(), &imageBeforeAlignment8);

// Search for the PCB "rectangle" in 8-bits image
float rectangleDimTolerance = 500.f; // 500 pixels tolerance on the rectangle's dimension
ERectangleGauge ERectangleGauge1;
ERectangle measuredRectangle;
ERectangleGauge1.SetTolerance(rectangleDimTolerance);
ERectangleGauge1.SetSize((float)(zmapWidth), (float)(zmapHeight));
ERectangleGauge1.SetThreshold(20);
ERectangleGauge1.SetCenterXY((float)(zmapWidth / 2.0), (float)(zmapHeight / 2.0));
ERectangleGauge1.SetTransitionChoice(ETransitionChoice_NthFromBegin);
ERectangleGauge1.SetTransitionIndex(0); // take the first transition ...
ERectangleGauge1.SetTransitionType(ETransitionType_Bw); // ... from black to white
ERectangleGauge1.SetNumFilteringPasses(10);
ERectangleGauge1.SetFilteringThreshold(5.0f); // threshold = 5 x the mean deviation
ERectangleGauge1.Measure(&imageBeforeAlignment8);
measuredRectangle = ERectangleGauge1.GetMeasuredRectangle();
float rectangleSizeX = ceil(measuredRectangle.GetSizeX()); // will be the width of the final ZMap
float rectangleSizeY = ceil(measuredRectangle.GetSizeY()); // will be the height of the final ZMap
```

```
// Store the 3D coordinates of the aligned rectangle corners
EPoint corner_2D[4];
E3DPoint corner_3D[4];
measuredRectangle.GetCorners(corner_2D[0], corner_2D[1], corner_2D[2], corner_2D[3]);
for (int i = 0; i<4; i++)
{
  E3DPoint cornerPtZMap;
  cornerPtZMap = E3DPoint(corner_2D[i].GetX(), // X pixel position
                          corner_2D[i].GetY(), // Y pixel position
         zMapOffset / zMapVerticalResolution); // = 10000 (height of the ZMap reference plane)
  zmapBeforeAlignmentFilter2.PixelToWorld(cornerPtZMap, corner_3D[i]);
}
```

Creating the filtered point cloud

1. In order to create a new ZMap with a well-defined reference plane (parallel with the regression plane through the filtered points), the application generates a point cloud from the smoothed ZMap.

In the code below, the filtered point cloud is named `filteredPc`:

```
// From the filtered ZMap, generate a new 3D point cloud
// Use that point cloud to estimate a better reference plane
E3DPointCloud filteredPc;
zmapBeforeAlignmentFilter2.ToPointCloud(filteredPc);
E3DPlaneFitter planeFitter;
E3DPlane refPlane_2 = planeFitter.Fit(filteredPc);  // find the best fit plane
```

2. The application fits a plane (the final regression plane called `refPlane_2`) through the filtered data points.

3. This new reference plane, that is very close to `refPlane_1`, is the new « horizontal » reference for the generation of the final ZMap on which the warpage is computed.

This plane is much less sensitive to the noise and outliers found in the original data.

Computing the warpage of the PCB

1. The application creates the final ZMap using the filtered point cloud.

The position of this new ZMap is based on:

- The plane `Ref_plane2` minus 10 mm as the reference plane.
- The 3D position of the corners to determine the reference (lower-left) corner and the new horizontal orientation (X axis). Using the corner positions, we specify the origin and the orientation vector for the generation of the new ZMap, assuming that the two planes are very close to each other.

The code that generates this aligned ZMap is:

```
// Generate the new ZMap with the filtered data
EZMap16 zmap_;
zmapGenerator.SetOrigin(corner_3D[2]);
zmapGenerator.SetReferencePlane(refPlane_2 - zMapOffset); // refPlane_2 - 10mm
E3DPoint horizDirection(corner_3D[1].X - corner_3D[0].X,
                        corner_3D[1].Y - corner_3D[0].Y,
                        corner_3D[1].Z - corner_3D[0].Z);
zmapGenerator.SetOrientationVector(horizDirection);
zmapGenerator.SetResolution((int)rectangleSizeX, (int)rectangleSizeY);
zmapGenerator.SetExtension(0.f); // no border
zmapGenerator.Convert(filteredPc, zmap_);
```

2. To compute the warpage of the PCB, the application only needs the maximum and the minimum height from the aligned ZMap.

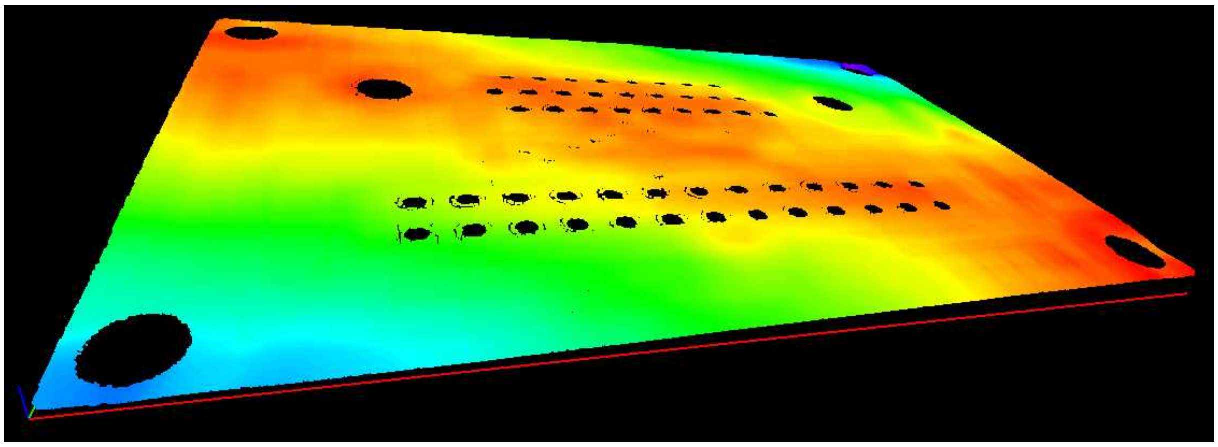
   It retrieves these values with the **static method** `ComputeStatistics` in the class `Easy3D`.

The corresponding code is:

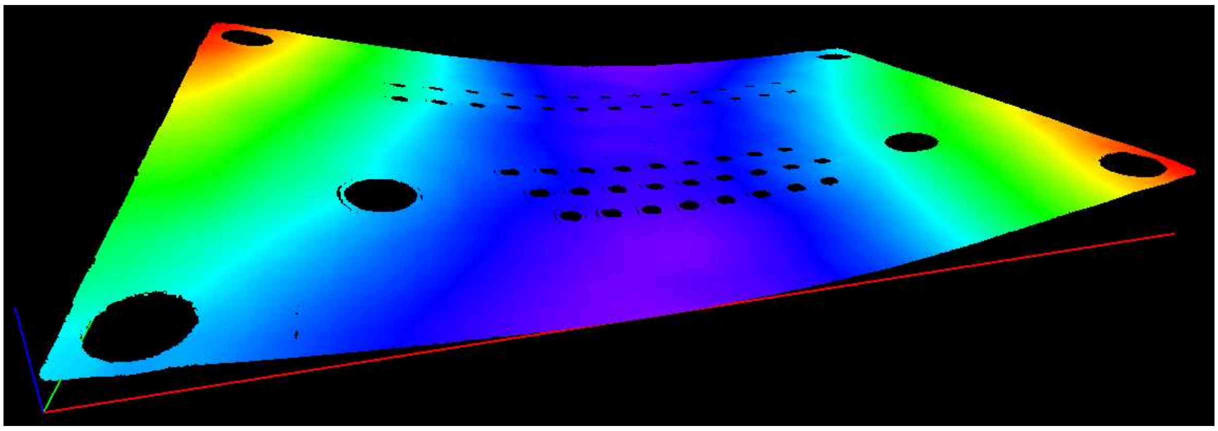
```
// Calculate warpage in metric unit
UINT32 validCount;
float minValue, maxValue, averageValue;
Easy3D::ComputeStatistics(zmap_, validCount, minValue, maxValue, averageValue);
float warpage = maxValue - minValue;
```

- The sample application displays the PCB in 3D and applies a color map.
- The 3D viewer applies a scale factor of 10 on Z axis to emphasize the warpage.

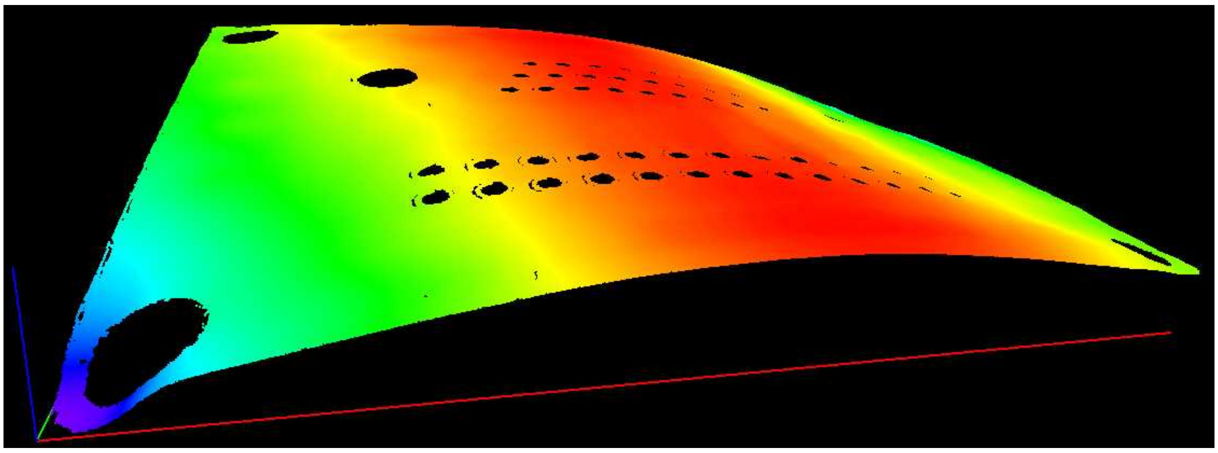
Sample 3D renderings of the resulting depth maps (showing the warpage) and their corresponding warpage measurement are illustrated below:



Warpage = 0.145 mm



Warpage = 0.407 mm



Warpage = 0.620 mm