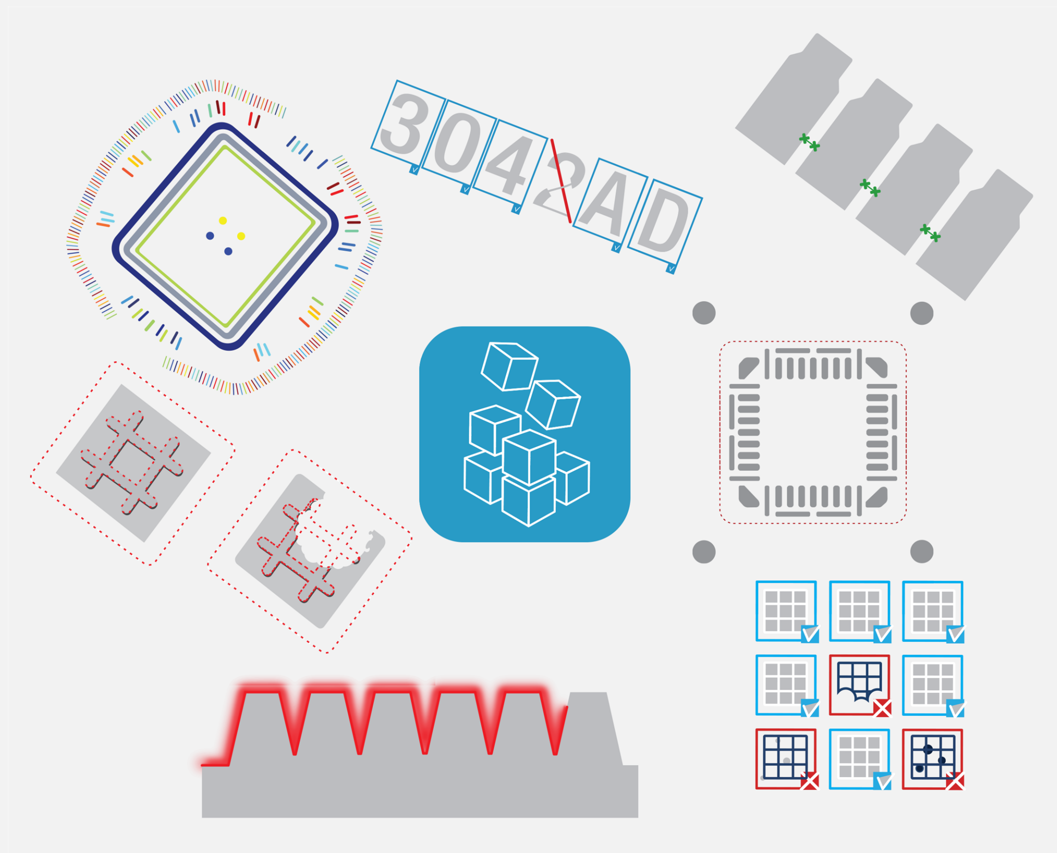
euresys
Empowering Computer Vision

USER GUIDE

Open eVision

Easy3D Compatibility with Automation Technology 3D Sensors



© EURESYS s.a. 2021 - Document D190EN-Easy3D Compatibility with Automation Technology 3D Sensors-2.16.1.1155 built on

This documentation is provided with Open eVision 2.16.1 (doc build 1155).
www.euresys.com

Easy3D Compatibility with Automation Technology 3D Sensors

Introduction

AT compact 3D products are integrated laser triangulation sensors. The **cx3D** library distributed with the **cxSupportPackage** supports various operations on 2D and 3D structures, like the calibration of the laser triangulation setup.

The specifications are available on the manufacturer website:
`https://www.automationtechnology.de/cms/en/3d-laser-sensors-for-measurements-by-means-of-laser-triangulation/`

- This document explains how to use the 3D data coming from these sensors with Open eVision**Open eVision** 3D libraries and tools.
- A sample application distributed with source code demonstrates that integration. This application is freely available in the Easy3D Sensors Compatibility additional resources package on Euresys**Euresys** web site.

Resources

This document and the sample applications are based on the following resources:

- C5-2040CS sensor, but the sample application is compatible with the entire C5 and MCS series.
- **AT** cxSupportPackage 2.5 RC3 64 bits
- **Open eVision** 2.12
- Microsoft Visual Studio 2017

Features

- The following data formats are exposed within **cx3D** (in **Genicam** Standard Features Naming Convention):
  - `UncalibratedC`: uncalibrated 2.5D depth image (or range map). The distance data does not represent a physical unit and may be non-linear.
  - `CalibratedABC`: Point cloud image with three coordinates in grid organization. The pixel coordinates are calibrated and transformed to the world coordinate system (WCS). Invalid pixels are marked with an invalid z-value.
  - `RectifiedC`: rectified 2.5D depth image. The distance data has been rectified to a uniform sampling pattern in X and Y direction.
- These formats support various pixel formats and organizations.

- **Easy3D** exposes also similar structures:
  - • A depth map (EDepthMap8/16/32f) is a container for uncalibrated 2.5D data. 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 point cloud (EPointCloud) is a set of 3D points (x, y and z coordinates) representing the scanned object in the world metric space.
  - • A ZMap (EZMap8/16/32f) is a grayscale images like depth maps but represent metric and corrected 3D points.
- The correspondences between the **cx3D** and the **Easy3D** classes are:

AT cx3D objects	Euresys Easy3D objects
cx::Image (UncalibratedC) Formats: □ CX_PF_MONO_8 □ CX_PF_MONO_10 □ CX_PF_MONO_10p □ CX_PF_MONO_12 □ CX_PF_MONO_12p □ CX_PF_MONO_16 (preferred) □ CX_PF_MONO_32 □ CX_PF_MONO_64	**Easy3D::EDepthMap** Formats: □ EDepthMap8 for unsigned 8-bit integer □ EDepthMap16 for unsigned 16-bit integer □ EDepthMap32f for 32-bit float.
cx::c3d::CPointCloud (CalibratedABC) Formats: □ CX_PF_COORD3D_ABC8_PLANAR □ CX_PF_COORD3D_ABC8 □ CX_PF_COORD3D_ABC16 □ CX_PF_COORD3D_ABC16_PLANAR □ CX_PF_COORD3D_ABC32f (preferred) □ CX_PF_COORD3D_ABC32f_PLANAR	**Easy3D::EPointCloud**: □ A continuous buffer of E3DPoint.E3DPoint is a struct (x,y,z) with 3 floats.
cx::c3d::ZMap (Rectified) Formats : □ CX_PF_COORD3D_C16 □ CX_PF_COORD3D_C32f (preferred)	**Easy3D::EZMap** Formats: □ EZMap8 for unsigned 8-bit integer □ EZMap16 for unsigned 16-bit integer □ EZMap32f for 32-bit float.

Sample C++ codes for the generic conversions from **cx3D** to **Easy3D** formats is provided at the end of this document:

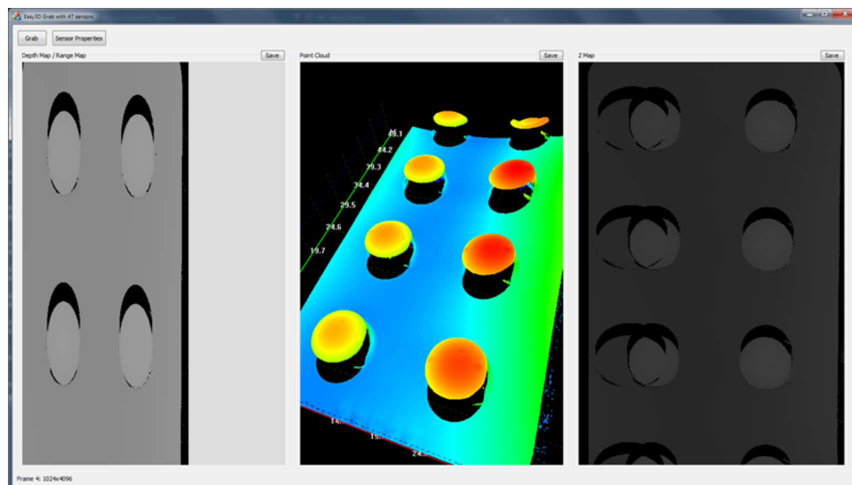
- `Easy3D::EDepthMap* DepthMapConvertion(const AT::cx::Image& atRangeMap)`
  - Create a EDepthMap8 or EDepthMap16 object depending on the format of the cx::Image.
  - For better efficiency, the data are not copied but referenced in the EDepthMap container.
- `Easy3D::EPointCloud* PointCloudConvection(const AT::cx::c3d::PointCloud& atPC, float idv)`
  - Create an Easy3D::EPointCloud and copy the 3D points.
  - Undefined values (when Z==idv) are not copied.
  - As Easy3D::EPointCloud contains world space points, the scale and offset are applied to the valid c3d::PointCloud positions.

- `Easy3D::EZMap* ZMapConvertion(const AT::cx::c3d::ZMap& atZmap, float idv)`
  - Create a `EZMap16` or `EZMap32f` object depending on the format of `c3d::ZMap`.
  - Undefined values (when `Z==idv`) are converted to `Easy3D::EZMap` undefined values.
  - The scale and offset are kept in the `Easy3D::EZMap` object.

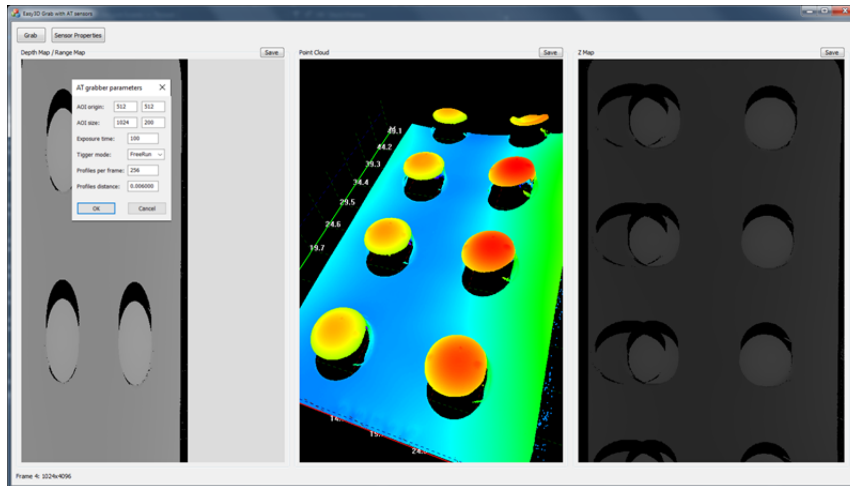
TIP
The **Easy3DGrab** sample application implements the acquisition and conversion for selected formats.

Easy3DGrab sample application

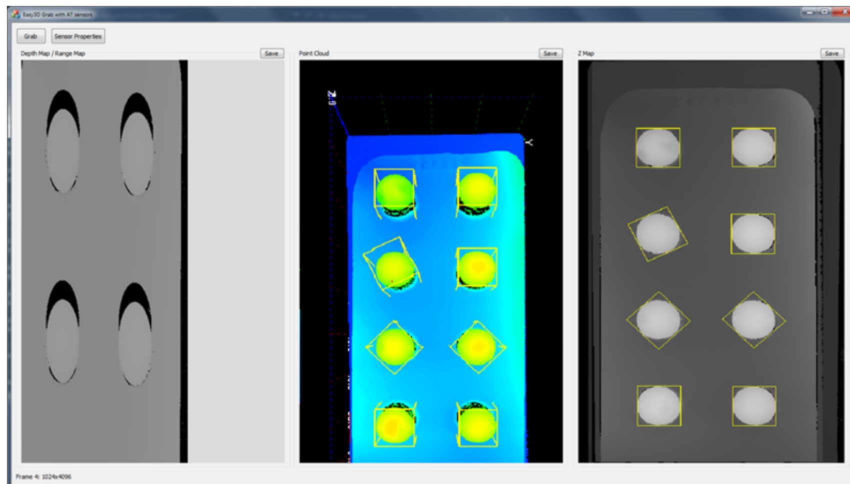
Easy3DGrab is distributed with C++ source code as an **Open eVision** additional resource.
- It features the acquisition of **AT** 3D data, the conversion to depth maps, point clouds and ZMaps.
- You can save these representations.
- Click on the Grab button to acquire a new image.
- Open the Sensor Properties dialog to access some of the device parameters.
- The Object extraction function is exposed but you can use it only with the Easy3DObject license.



The Easy3DGrab application: EDepthMap (left), EPointCloud (center), EZMap (right)



Tuning the C5-CS sensor parameters to filter the range data



Automatic extraction of 3D objects with Easy3DObject library

C++ code sample to convert cx3D to Easy3D objects

Converting an AT range map (`UncalibratedC`) to `Easy3D::EDepthMap`

```
// Create an Open eVision EDepthMap8 or EDepthMap16 depending on the format of the AT
range map.
// The pixel buffer of the given range map is directly referenced in the depth map,
no copy is performed.
// The EDepthMap must be explicitly released after usage.

Easy3D::EDepthMap* DepthMapConvertion(const AT::cx::Image& atRangeMap)
{
  Easy3D::EDepthMap* eDepthMap = NULL;

  switch (atRangeMap.pixelFormat())
  {
  case cx_pixel_format::CX_PF_MONO_8:
    eDepthMap = new Easy3D::EDepthMap8();
    eDepthMap->SetBufferPtr(atRangeMap.width(), atRangeMap.height(), atRangeMap.data
(), (int)atRangeMap.linePitch() * 8);
    break;

  case cx_pixel_format::CX_PF_MONO_16:
    eDepthMap = new Easy3D::EDepthMap16();
    eDepthMap->SetBufferPtr(atRangeMap.width(), atRangeMap.height(), atRangeMap.data
(), (int)atRangeMap.linePitch() * 8);
    break;

  default:
    throw std::exception("RangeMap format not supported");
    break;
  }
  return eDepthMap;
}
```

Converting an AT point cloud (PointCloud) to Easy3D::EPointCloud

```
// Create an Open eVision EPointCloud depending on the format of the AT PointCloud.
// The 3D positions are converted in "world space" by applying scale and offset.
// Undefined point are discarded.
// The EPointCloud object must be explicitly released after usage.


Easy3D::EPointCloud* PointCloudConvection(AT::cx::c3d::PointCloud& atPC, float idv)
{
  Easy3D::EPointCloud* ePC = new Easy3D::EPointCloud();
  std::vector<Easy3D::E3DPoint> vector3DPoints;
  vector3DPoints.reserve(atPC.points.size());

  Easy3D::E3DPoint scale(atPC.scale.x, atPC.scale.y, atPC.scale.z);
  Easy3D::E3DPoint offset(atPC.offset.x, atPC.offset.y, atPC.offset.z);

  switch (atPC.points.pixelFormat())
  {
  case cx_pixel_format::CX_PF_COORD3D_ABC8_PLANAR:
  {
    int8_t* atPCp = static_cast<int8_t*>(atPC.points.data());
    int dataSize = atPC.points.planePitch() / sizeof(int8_t);
    for (int ii = 0; ii < dataSize; ++ii)
    {
      if (atPCp[ii + 2 * dataSize] != idv)
      {
        Easy3D::E3DPoint p(atPCp[ii] * scale.X + offset.X, atPCp[ii + dataSize] *
scale.Y + offset.Y, atPCp[ii + 2 * dataSize] * scale.Z + offset.Z);
        vector3DPoints.push_back(p);
      }
    }
    break;
  }
  case cx_pixel_format::CX_PF_COORD3D_ABC8:
  {
    for (unsigned y = 0; y < atPC.points.height(); y++)
    {
      Point3i8* row = atPC.points.row < Point3i8>(y);
      for (unsigned x = 0; x < atPC.points.width(); x++)
      {
        const Point3i8 atP = row[x];
        if (atP.z != idv)
        {
          Easy3D::E3DPoint p(atP.x * scale.X + offset.X, atP.y * scale.Y + offset.Y,
atP.z * scale.Z + offset.Z);
          vector3DPoints.push_back(p);
        }
      }
    }
    break;
  }
  case cx_pixel_format::CX_PF_COORD3D_ABC16:
  {
    for (unsigned y = 0; y < atPC.points.height(); y++)
    {
      Point3i16* row = atPC.points.row < Point3i16>(y);
      for (unsigned x = 0; x < atPC.points.width(); x++)
      {
        const Point3i16 atP = row[x];
        if (atP.z != idv)
        {
          Easy3D::E3DPoint p(atP.x * scale.X + offset.X, atP.y * scale.Y + offset.Y,
atP.z * scale.Z + offset.Z);
          vector3DPoints.push_back(p);
```

```
        }
      }
    }
    break;
  }
  case cx_pixel_format::CX_PF_COORD3D_ABC16_PLANAR:
  {
    int16_t* atPCp = static_cast<int16_t*>(atPC.points.data());
    int dataSize = atPC.points.planePitch() / sizeof(int16_t);
    for (int ii = 0; ii < dataSize; ++ii)
    {
      if (atPCp[ii + 2 * dataSize] != idv)
      {
        Easy3D::E3DPoint p((float)atPCp[ii] * scale.X + offset.X, (float)atPCp[ii + 
dataSize] * scale.Y + offset.Y, (float)atPCp[ii + 2 * dataSize] * scale.Z + 
offset.Z);
        vector3DPoints.push_back(p);
      }
    }
    break;
  }
  case cx_pixel_format::CX_PF_COORD3D_ABC32f:
  {
    for (unsigned y = 0; y < atPC.points.height(); y++)
    {
      AT::cx::Point3f* row = atPC.points.row < AT::cx::Point3f>(y);
      for (unsigned x = 0; x < atPC.points.width(); x++)
      {
        const AT::cx::Point3f atP = row[x];
        if (!isnan(atP.z) && atP.z != idv)
        {
          Easy3D::E3DPoint p(atP.x * scale.X + offset.X, atP.y * scale.Y + offset.Y, 
atP.z * scale.Z + offset.Z);
          vector3DPoints.push_back(p);
        }
      }
    }
    break;
  }
  case cx_pixel_format::CX_PF_COORD3D_ABC32f_PLANAR:
  {
    float* atPCp = static_cast<float*>(atPC.points.data());
    int dataSize = atPC.points.planePitch() / sizeof(float);
    for (int ii = 0; ii < dataSize; ++ii)
    {
      if (!isnan(atPCp[ii + 2 * dataSize]) && atPCp[ii + 2 * dataSize] != idv)
      {
        Easy3D::E3DPoint p(atPCp[ii] * scale.X + offset.X, atPCp[ii + dataSize] * 
scale.Y + offset.Y, atPCp[ii + 2 * dataSize] * scale.Z + offset.Z);
        vector3DPoints.push_back(p);
      }
    }
    break;
  }
  default:
  {
    //need to convert or ordinate AT value
    throw std::exception("PointCloud format not supported");
    break;
  }
  }

  ePC->AddPoints(vector3DPoints);

  return ePC;
```

Converting an AT rectified map (`ZMap`) to `Easy3D::EZMap`

```
// Create an Open eVision EZMap depending on the format of the AT ZMap.
// The EZMap object must be explicitly released after usage.


Easy3D::EZMap* ZMapConvertion(AT::cx::c3d::ZMap& atZmap, float idv)
{
  Easy3D::EZMap* eZmap = NULL;

  unsigned int width = atZmap.img.width();
  unsigned int height = atZmap.img.height();

  switch (atZmap.img.pixelFormat())
  {
  case cx_pixel_format::CX_PF_COORD3D_C16:
  {
    Easy3D::EZMap16* eZmap16 = new Easy3D::EZMap16(width, height);
    int16_t Easy3DUndefinedValue = eZmap16->GetUndefinedValue().Value;

    for (unsigned int yy = 0; yy < height; ++yy)
    {
      int16_t* r = atZmap.img.row<int16_t>(yy);
      for (int xx = 0; xx < width; ++xx)
      {
        if (r[xx] == idv)
          eZmap16->SetPixel(Easy3DUndefinedValue, xx, yy);
        else
          eZmap16->SetPixel(r[xx], xx, yy);
      }
    }

    eZmap = eZmap16;
    break;
  }
  case cx_pixel_format::CX_PF_COORD3D_C32f:
  {
    Easy3D::EZMap32f* eZmap32f = new Easy3D::EZMap32f(width, height);
    float Easy3DUndefinedValue = eZmap32f->GetUndefinedValue().Value;

    for (unsigned int yy = 0; yy < height; ++yy)
    {
      float* r = atZmap.img.row<float>(yy);
      for (int xx = 0; xx < width; ++xx)
      {
        if (r[xx] == idv)
          eZmap32f->SetPixel(Easy3DUndefinedValue, xx, yy);
        else
          eZmap32f->SetPixel(r[xx], xx, yy);
      }
    }

    eZmap = eZmap32f;
    break;
  }
  default:
    //need to convert or ordinate AT value
    throw std::exception("Zmap format not supported");
    break;
  }

  eZmap->SetResolution(atZmap.scale.x, atZmap.scale.y, atZmap.scale.z);

  return eZmap;
}
```