Documentation Azure Kinect in Halcon

All necessary information on the code and applications used to scan window frames.

[1. Overview 2](#_Toc155977415)

[1.1 The problem (current situation) 2](#_Toc155977416)

[1.2 Requirements 2](#_Toc155977417)

[1.3 Overview of tested methods 2](#_Toc155977418)

[2. Using Azure Kinect to detect and localise a window frame 4](#_Toc155977419)

[2.1 External application to generate point clouds 4](#_Toc155977420)

[2.2 Alternative application to generate point clouds 4](#_Toc155977421)

[2.3 File conversion from .pts to .ply 4](#_Toc155977422)

[2.4 Halcon script to grab point clouds using KinectCloud and CloudCompare. 5](#_Toc155977423)

[2.5 Camera discontinued 6](#_Toc155977424)

[3. Filtering and analysing point cloud data 7](#_Toc155977425)

[3.1 Rough explanation of necessary output data 7](#_Toc155977426)

[3.2 Input variables 7](#_Toc155977427)

[3.3 Thresholding 8](#_Toc155977428)

[3.4 Noise filtering 8](#_Toc155977429)

[3.5 Generating a bounding box 9](#_Toc155977430)

[3.6 Translating and rotating pose 10](#_Toc155977431)

[3.7 Writing data to file 10](#_Toc155977432)

[4. Current status and potential next steps 12](#_Toc155977433)

[4.1 Data exchange between PC and cobot 12](#_Toc155977434)

[4.2 Path generation based on vision data 12](#_Toc155977435)

[4.3 Using vision to detect multiple objects 12](#_Toc155977436)

[4.4 Setting up an HMI (Human Machine Interface) 12](#_Toc155977437)

# Overview

## 1.1 The problem (current situation)

This project is all about scanning, detecting and locating objects. These objects can be doors, door frames, windows and window frames. This provides quite a challenge, since TiFa Lemele does not (yet) work with bar codes or other product information methods. If they had, it would have made the vision side of things much easier, since it would’ve been possible to simply load the CAD data and give it offsets based on distance and rotation. When a product arrives at the painting station, there is no data.

## 1.2 Requirements

Fabrication requirements:

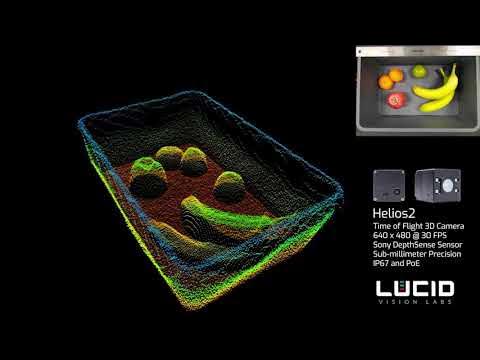
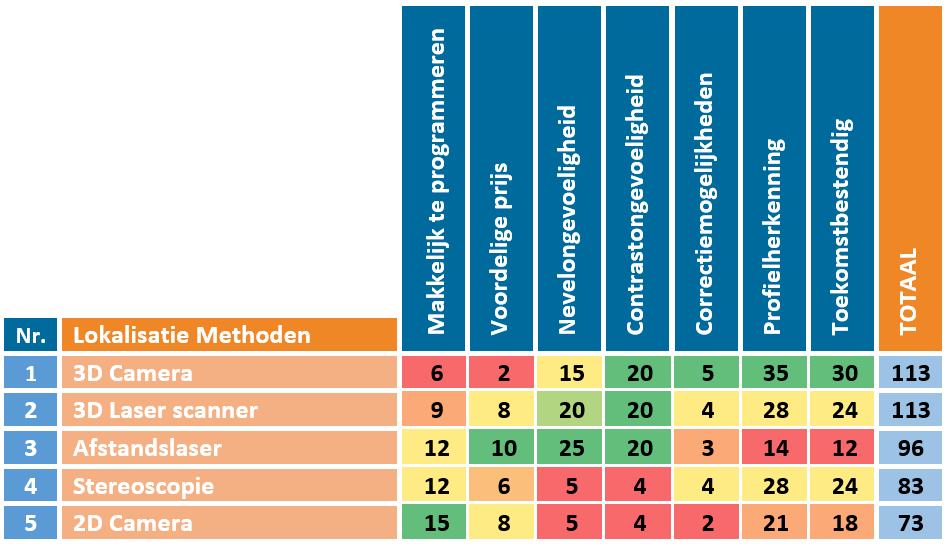
* Since there is no prior information on the product, it **must** be scanned in “6D”. This consists of three dimensions (X, Y, Z) and the rotation of each of these. This way, it will be possible to determine what and where the product is.
* Accuracy of the scan **should** be within ±5mm.
* The system **should** be compliant with the *ISO 10218-1* and *ISO 10218-2* standards to ensure safety for the operator.
* The system **must** be able to be changed to support growing production.
* The complete automated spray-painting installation **must** cost less than €150 000.

User requirements:

* The product **must** not sway.
* The product profile (shape of the beams) **must** be selected by hand.
* The system **must** be able to support products of up to 4 by 6 meters.

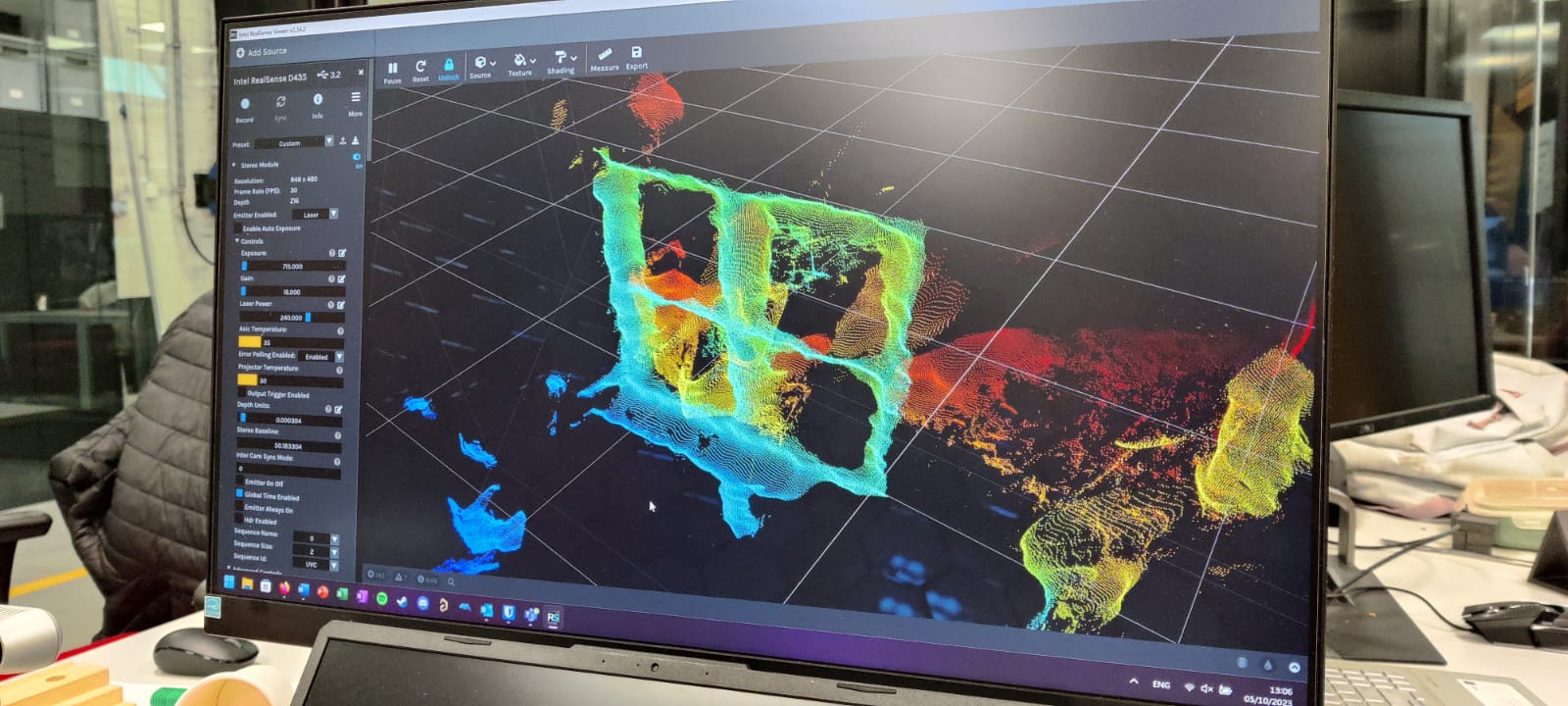
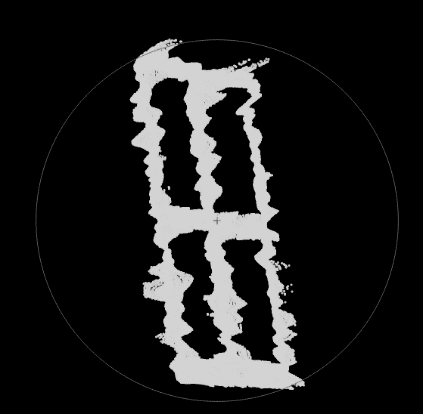
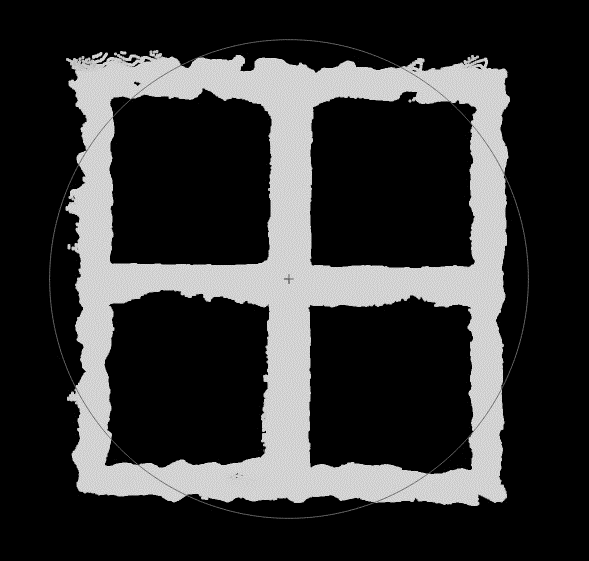
## 1.3 Overview of tested methods

Throughout the experimental phase of the development of this system, various detection methods have been compared and tested. Once it was clear that “6D” scanning is absolutely necessary, a list of possible solutions has been made to judge using the Kesselring method. 3D laser scanners and 3D cameras seem to be the best options here. However, this list is incomplete. Methods that weren’t included are LIDAR and 3D stereoscopy. These methods have been considered later on.



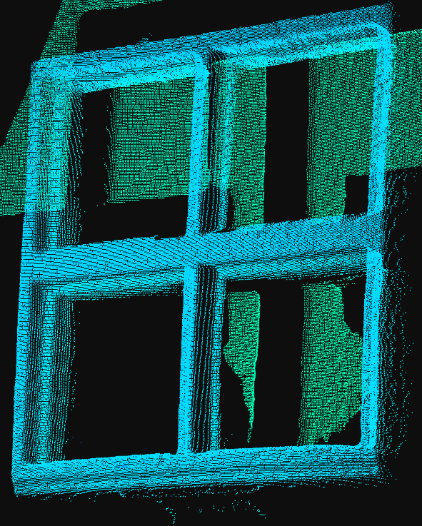
*Figure 1: Results Kesselring method*

Based on the results of this chart, information was gathered about which products would be most suitable for testing. There are multiple types of 3D cameras and some 3D scanners. The main problem, however, is the cost. Most of the scanners and cameras cost well over €1 500, some even in the direction of €10 000. This is simply too expensive for the very limited budget. Some 3D cameras and 3D scanners were already available at Perron 038. There was one camera, the Intel RealSense D435, which seemed ideal for this project. Costing around €350, this entry level USB 3D camera would be an amazing solution. After a lot of testing, filtering and post processing, the camera appeared to be way too inaccurate for the required ±5mm. Most measurements were over 5cm off. Below, images are shown of a thoroughly filtered and processed point cloud generated with a D435:



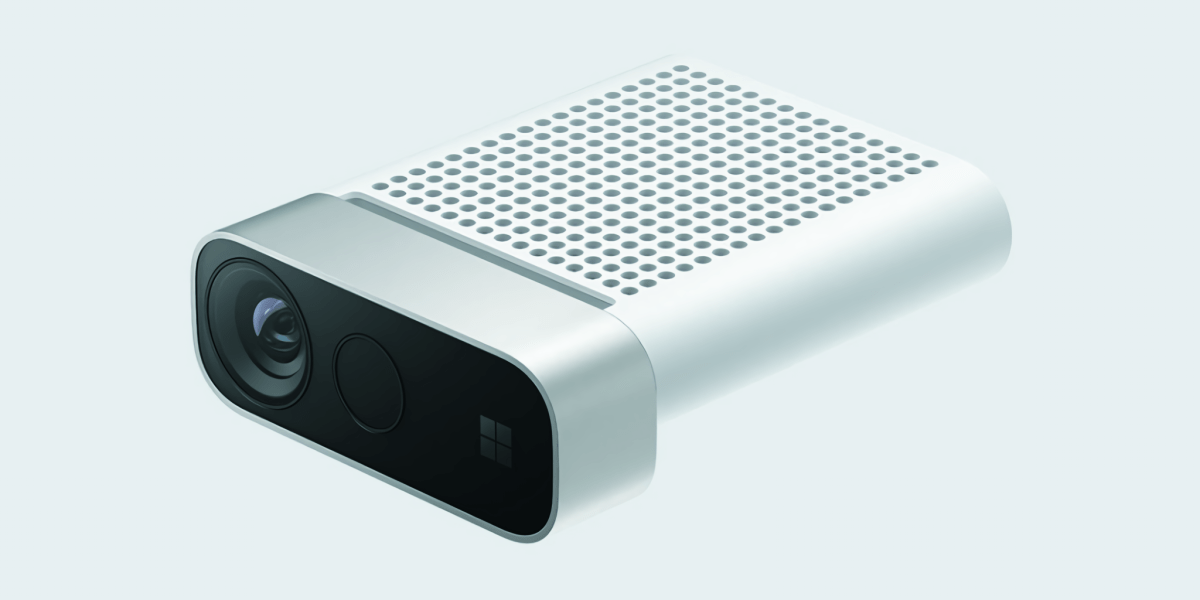
*Figure 2: front and side views of filtered window frame and a demonstrative image of the camera noise using the Intel RealSense D435*

This method has proven to be inadequate. After a tip from one of the teachers who had access to a relatively cheap (€1 100) LIDAR sensor, the Sick TiM551, that was up next for testing. During testing, it became clear that this particular sensor wouldn’t be sufficient for this project. Its only output is a single digital bit. It is pretty much impossible to generate a point cloud that way. More expensive LIDAR sensors would be able to generate proper point clouds, but those are well above budget.



*Figure 3: Azure Kinect camera in SDK Viewer*

The next plan was to test 3D stereoscopy using 2D cameras. However, none were available at Perron 038 anymore. The only thing they had laying around for testing was a new Microsoft Azure Kinect DK. No one had worked with this 3D camera yet, but based on specifications, this seemed to be another good option. Its 2D and 3D resolutions are much higher than the D435, while costing around €400. After setting the camera up in the Azure Kinect SDK Viewer application, it was clear that this camera has a lot of potential.



*Figure 4: Microsoft Azure Kinect camera*

# Using Azure Kinect to detect and localise a window frame

After some initial tests within the Azure Kinect SDK Viewer, the next step was to use it with Halcon. There was one large problem: the camera isn’t directly compatible with Halcon. There seemed to be no way to generate still point clouds using the SDK Recorder application either.

## 2.1 External application to generate point clouds

Since the SDK’s recorder doesn’t export any filetype other than .mkv video, a different solution is necessary. Luckily, there’s an example program that can be found on the SDK’s GitHub page[[1]](#footnote-2). It’s called “fastpointcloud”. In order to use it, certain dependencies have to be installed and added to PATH. Once the GitHub repository is cloned, the application can be built using cmake. Once built, it’ll be possible to call fastpointcloud.exe from the command prompt. Simple point cloud data can be saved as .ply or .stl. All camera parameters must be edited in the source code, after which the application will have to be rebuilt. This makes configuring and testing settings rather tedious. Another thing that’s missing is colour data.

## 2.2 Alternative application to generate point clouds

The company *Virtual Environments Group* and a user named *bennywwg* have developed an application named KinectCloud[[2]](#footnote-3). It makes it possible to generate coloured point clouds. Even better is that all camera parameters can be adjusted using command line arguments (list of options can be found on the GitHub page). This makes testing and configuring the camera a lot easier. There is only one major downside: files are exported as .pts. Halcon doesn’t recognise this filetype, so the files must be converted.

## 2.3 File conversion from .pts to .ply

One of the dangers of file conversion is data compression and/or data loss. Luckily, there is little to worry about when it comes to the conversion of .pts files to .ply. The only difference between the two is that .ply files have extra headers. The rest of the data is the same. A simple way to properly convert these files is to use an application called CloudCompare[[3]](#footnote-4). This application can be called from the command prompt and uses command line arguments to set up tasks. Documentation on the command lines can be found over at their wiki[[4]](#footnote-5). Using the following command, .pts files can be loaded in and properly exported as .ply files:

CloudCompare.exe -SILENT -O {filename} -NO\_TIMESTAMP -C\_EXPORT\_FMT PLY -SAVE\_CLOUDS FILE "{filename}"

In this command, the arguments have the following functionality: -SILENT prevents a window from being opened. -O {filename} opens a specified file. -NO\_TIMESTAMP prevents a timestamp from being added to the export filename. -C\_EXPORT\_FMT PLY defines the export format, .ply. -SAVE\_CLOUDS FILE “{filename}” specifies the export path and name.

## 2.4 Halcon script to grab point clouds using KinectCloud and CloudCompare.

Halcon uses a function named system\_call() to run commands through the command prompt. The command will have to be inserted as a string. This makes it possible to define all command line arguments as variables within the Halcon script. This way, it’s relatively easy to quickly configure settings and camera parameters. The code to configure and run the KinectCloud application looks as follows:

\* Device IDs (Centre C)

IDC := '000237925112'

\* Color exposure [-ce] (time in nanoseconds)  
Exposure := ' -ce 70000'

\* White balance [-cw] in Kelvin (steps of 10)

WhiteBalance := ' -cw 3600'

\* Declare device topology [-dt] (Standalone [a], Master [m], Slave [s])

Topology := ' -dt ' + IDC + ' a'

\* Declare color resolution [-dra {res}] for all, [-dr ser {res}] for target (res: 720p, 1080p, 1440p, 1536p, 2160p, 3072p)

Resolution := ' -dra 1080p'

\* Declare depth mode [-dma {mode}] for all, [dm ser {mode}] for target (mode: NFOV\_2X2BINNED, NFOV\_UNBINNED, WFOV\_2X2BINNED, WFOV\_UNBINNED)

DepthMode := ' -dma NFOV\_UNBINNED'

\* Set file path and name [-o] (prefix [-o] and suffix {%s.pts} in system\_call) (%s = camera serial number, %f = frame number)

Path := 'C:/TestScan\_'

\* Set KinectCloud directory

PathKinectCloud := 'C:/source/repos/KinectCloud/x64/Debug'

\* Take point cloud snapshot using KinectCloud (.pts file) (-s for snapshot, -da to apply params to all devices)

set\_current\_dir (PathKinectCloud)

system\_call ('KinectCloud.exe -s -da' + Exposure + WhiteBalance + Topology + Resolution + DepthMode + ' -o ' + Path + '%s.pts')

The only arguments that haven't been explained in the commentary lines are -s and -da. The argument -s is used to take a snapshot instead of a video recording. -da is used to apply general parameters such as exposure and white balance to all connected devices. Since files are exported as .pts, they must be converted to .ply. Calling CloudCompare to convert the file saved with the code above, looks like this:

\* Set CloudCompare directory

PathCloudCompare := 'C:/Program Files/CloudCompare'

\* Load point clouds in CloudCompare and export as .ply

set\_current\_dir (PathCloudCompare)

system\_call ('CloudCompare.exe -SILENT -O ' + Path + IDC + '.pts' + ' -NO\_TIMESTAMP -C\_EXPORT\_FMT PLY -SAVE\_CLOUDS FILE "' + Path + IDC + '.ply"')

All command line arguments have been explained in [paragraph 2.3](#_2.3_File_conversion). This is the code necessary to capture and convert usable point clouds with colour data for the Microsoft Azure Kinect cameras.

## 2.5 Camera discontinued

In August 2023, Microsoft announced the discontinuation of the Azure Kinect cameras[[5]](#footnote-6). They were only available for sale until the end of October 2023. Luckily, this doesn’t mean this technology is done for. One of Microsoft’s partner companies, Orbbec, will continue to produce and develop this camera system, albeit with some changes. Orbecc’s Femto Bolt[[6]](#footnote-7) uses the same camera system with the same ToF technology for a cost of around US$400. Some of the changes that stand out when comparing the Femto Bolt to the Azure Kinect are the Bolt’s much smaller housing and the use of more advanced connectors for multi-camera setups instead of the Kinect’s two 3.5mm jack ports. Another major difference is that the Bolt doesn’t appear to have a microphone array for surround sound capture.



*Figure 5: Orbbec Femto Bolt*

The Femto Bolt does use a different SDK. Orbbec has developed their own SDK[[7]](#footnote-8), and Microsoft’s will no longer be used. The good news is that there is a wrapper to make the Azure Kinect compatible with Orbbec’s SDK and to make the Femto Bolt compatible with the Azure Kinect SDK.

# Filtering and analysing point cloud data

In this chapter, the process of filtering and then analysing the data gathered from the point clouds will be explained.

## 3.1 Rough explanation of necessary output data

In order to properly adjust the robot/cobot’s programming, a few things are necessary:

1. Measurements of the product itself. These three variables will be used to determine the length of and/or offset the robot paths.
2. The orientation of the product is another required output. This orientation plane will be translated into a feature plane for the robot program.
3. The location of the product compared to the robot. This will provide the positioning data for the feature plane.
4. Information on what the product looks like is required. Does it have cross beams, is it a door, etc.

During this semester, requirements 1 through to 3 have been successfully accounted for. The output data currently consists of 3D measurements (X, Y, Z) and a 6D pose (TransX, TransY, TransZ, RotX, RotY, RotZ).

## 3.2 Input variables

In order to be able to properly threshold and filter the acquired data, a few things must be configured. First off: the “box” in which the product will appear. We can use this data to remove any points that exist outside this area. The code looks like this:

\* Scanning box measurements (in meters)

Xmin := -0.6

Xmax := 0.6

Ymin := -0.4

Ymax := 1.1

Zmin := 1.55

Zmax := 1.95

From the camera’s perspective, the axes are as follows: X to the right, Y down, Z forwards. The output data must be translated and rotated to match the coordinate system of the robot/cobot. This is the second set of input variables. It looks like this:

\* Pose translation (camera to robot/cobot) (in meters/degrees)

TransX := -0.149514

TransY := -0.971785

TransZ := 0.241117

RotX := -90

RotY := 0

RotZ := 180

These are the values of the translation between camera and cobot with our final demo setup.

Next, the captured and converted point cloud data must be loaded in. This is done using the following line of code:

\* Read-in data (Centre)

read\_object\_model\_3d (Path + IDC + '.ply', [], [], [], PointCloudC, Status)

## 3.3 Thresholding

Now that the data has been loaded in and a scanning area has been configured, data can be removed using thresholding. The code looks like this:

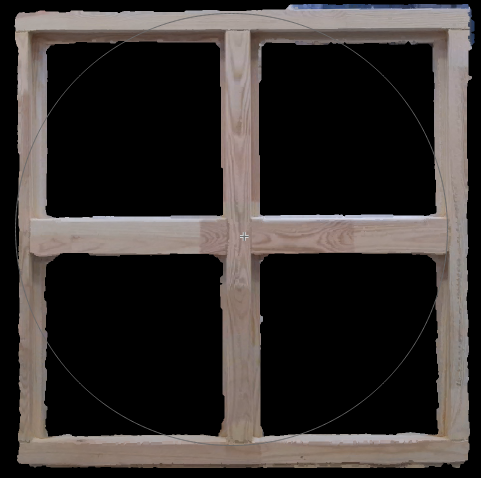
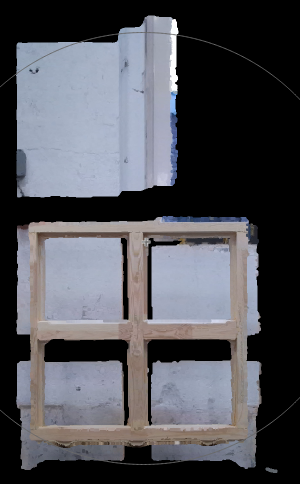
\* Coarse filter data (Centre)

select\_points\_object\_model\_3d (PointCloudC, 'point\_coord\_x', Xmin, Xmax, PointCloudCThresholded)

select\_points\_object\_model\_3d (PointCloudCThresholded, 'point\_coord\_y', Ymin, Ymax, PointCloudCThresholded)

select\_points\_object\_model\_3d (PointCloudCThresholded, 'point\_coord\_z', Zmin, Zmax, PointCloudCThresholded)

The argument ‘point\_coord\_x’ lets the select\_points\_object\_model\_3d select all points that are within the range specified with the Xmin and Xmax variables. The same is done for the Y- and Z-axes. The results look like this:



No thresholding

X-axis

Y-axis

Z-axis

*Figure 6: thresholding steps visualised*

Thresholding the data removes by far the most unwanted data.

## 3.4 Noise filtering

There still is some noise and unwanted data that needs to be filtered out. Examples are small groups of noisy points that exist outside the window frame or the chains holding the product. This can be done using a “blob filter”. The code looks like this:

\* Filter out noise (Centre)

get\_object\_model\_3d\_params (PointCloudCThresholded, 'num\_points', NumPoints)

NumNeighbors := 15

InlierRate := 99

get\_object\_model\_3d\_params (PointCloudCThresholded, 'neighbor\_distance ' + NumNeighbors, DistanceDistribution)

Distance := sort(DistanceDistribution)[|DistanceDistribution| \* InlierRate / 100]

select\_points\_object\_model\_3d (PointCloudCThresholded, 'num\_neighbors ' + Distance, 15, NumPoints, PointCloudCThresholdedFiltered)

Similarly to the thresholding, only points that meet certain requirements will be selected, everything else will be removed. Getting the object parameters ‘num\_points’ and ‘neighbor\_distance’ generates a list of distances between points. By sorting this list and selecting all points that have a lot of other points nearby, blobs of noise can be filtered out. This configuration also worked to filter out the chains.

## 3.5 Generating a bounding box

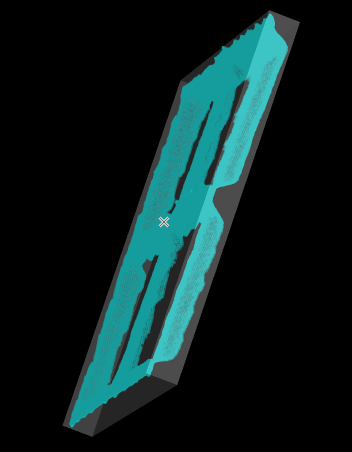
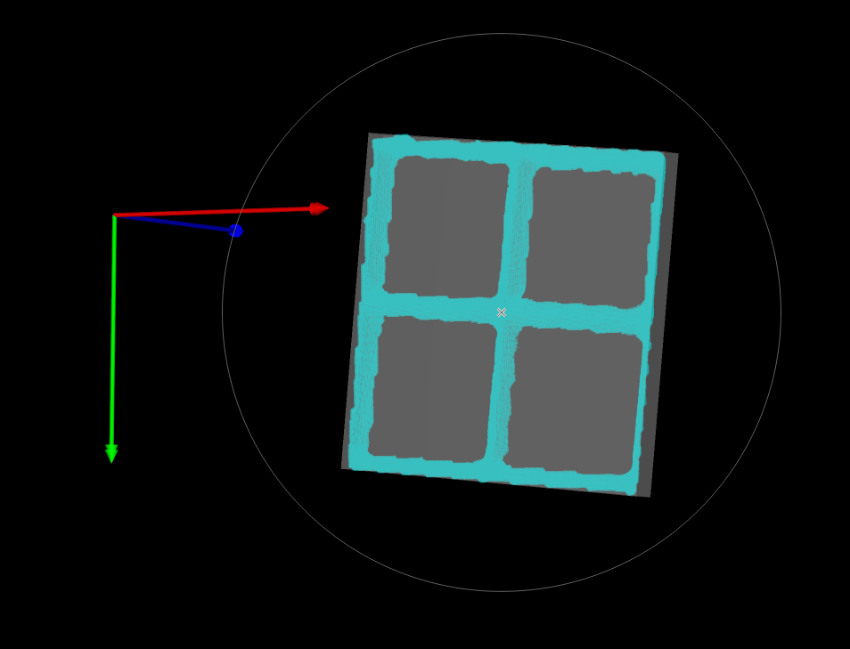
The next step is to generate an as small as possible bounding box around the remaining data. This is done using the following code:

\* Get bounding box and pose

smallest\_bounding\_box\_object\_model\_3d (PointCloudCThresholdedFiltered, 'oriented', BoundingBoxPose, LengthX, LengthY, LengthZ)

gen\_box\_object\_model\_3d (BoundingBoxPose, LengthX, LengthY, LengthZ, BoundingBox)

The function smallest\_bounding\_box\_object\_model\_3d generates a bounding box around the remaining data. It has four sets of output data: the pose of the bounding box and the length, width and height (LengthX, LengthY, LengthZ). Using these four outputs, a model can be generated using the function gen\_box\_object\_model\_3d. When comparing the bounding box to the thresholded and filtered data, it looks like this:



*Figure 7: Final point cloud (PointCloudCThresholdedFiltered) (blue) inside the smallest bounding box (BoundingBox) (grey)*

The pose and measurements are incredibly accurate. To test this, the output data was used to make a plane for the cobot. The cobot was manually moved using the axes of the pose and it stayed flush with the window frame. The X and Y measurements of the window frame are very accurate too (real measurements: 1m x 1m. Bounding box output: 0.9923m x 0.9911m). The Z measurement is less accurate (0.0819m instead of 0.113m) This is not a problem, since the profile type (and therefore depth) will be manually selected.

## 3.6 Translating and rotating pose

Next, the pose must be translated from the camera’s origin to the robot/cobot’s origin. With the final test setup, the translation and rotation looks like the configured variables seen in [paragraph 3.2](#_3.2_Input_variables). the code looks as follows:

\* Translate pose as seen from camera to pose as seen from origin robot/cobot

create\_pose (0, 0, 0, RotX, RotY, RotZ, 'Rp+T', 'abg', 'point', PoseRot)

rigid\_trans\_object\_model\_3d (BoundingBox, PoseRot, BoundingBoxTrans)

get\_object\_model\_3d\_params (BoundingBoxTrans, 'primitive\_pose', BoundingBoxTransPose)

BoundingBoxTransPose[0] := BoundingBoxTransPose[0] + TransX

BoundingBoxTransPose[1] := BoundingBoxTransPose[1] + TransY

BoundingBoxTransPose[2] := BoundingBoxTransPose[2] + TransZ

First, a pose is created using the rotational offsets. It’s important to use the argument ‘abg’ instead of ‘gba’ to make sure the rotation is calculated using RPY (Roll, Pitch, Yaw). Next, the bounding box is transformed to fit the new axes. After getting the pose parameters from the transformed bounding box, the translational values for X, Y and Z can be added to the pose’s X, Y and Z values.

## 3.7 Writing data to file

The last step is to export the data. This is done by writing it to a file.

\* Write output data to file

open\_file (SaveTxt, 'output', FileHandle)

fwrite\_string (FileHandle, 'The detected measurements (BoundingBox) are as follows:' + '\n' + '\n')

fwrite\_string (FileHandle, 'Measured X axis (width in m):' + '\n' + LengthX + '\n' + '\n')

fwrite\_string (FileHandle, 'Measured Y axis (height in m):' + '\n' + LengthY + '\n' + '\n')

fwrite\_string (FileHandle, 'Measured Z axis (depth in m, inaccurate):' + '\n' + LengthZ + '\n' + '\n')

fwrite\_string (FileHandle, '\n' + 'The detected translated pose values (BoundingBoxTransPose) are as follows:' + '\n' + '\n')

fwrite\_string (FileHandle, 'Transformation on the x axis (in m):' + '\n' + BoundingBoxTransPose[0] + '\n' + '\n')

fwrite\_string (FileHandle, 'Transformation on the y axis (in m):' + '\n' + BoundingBoxTransPose[1] + '\n' + '\n')

fwrite\_string (FileHandle, 'Transformation on the z axis (in m):' + '\n' + BoundingBoxTransPose[2] + '\n' + '\n')

fwrite\_string (FileHandle, 'Rotation around the x axis (in degrees):' + '\n' + BoundingBoxTransPose[3] + '\n' + '\n')

fwrite\_string (FileHandle, 'Rotation around the y axis (in degrees):' + '\n' + BoundingBoxTransPose[4] + '\n' + '\n')

fwrite\_string (FileHandle, 'Rotation around the z axis (in degrees):' + '\n' + BoundingBoxTransPose[5] + '\n' + '\n')

close\_file (FileHandle)

First, a file is opened according to the path declared for SaveTxt. Using the argument ‘output’, any existing file with that exact path and name will be overwritten. All measurements are written to the file using the LengthX, LengthY and LengthZ variables. The pose is written to the file using the table entries of the pose parameters in BoundingBoxTransPose. The string '\n' inserts and enter (endline). The contents of the output text file look like this:

The detected measurements (BoundingBox) are as follows:

Measured X axis (width in m):

0.992326

Measured Y axis (height in m):

0.991102

Measured Z axis (depth in m, inaccurate):

0.0818862

The detected translated pose values (BoundingBoxTransPose) are as follows:

Transformation on the x axis (in m):

-0.143818

Transformation on the y axis (in m):

-2.71663

Transformation on the z axis (in m):

0.0980508

Rotation around the x axis (in degrees):

269.918

Rotation around the y axis (in degrees):

0.400857

Rotation around the z axis (in degrees):

267.537

# Current status and potential next steps

**Disclaimer: all explanations/suggestions for future steps are what we, as the first team working on this project, find logical solutions. With more research, simpler or better solutions may very well be found. This chapter is meant to be some sort of guiding hand for the next team to get started.**

Just a few days ahead of the demo day, this is where the project is at regarding machine vision. Currently, the output values can be used to manually create a feature plane on the cobot. Combined with manual inputs of the length and width, it is possible to create proper programs for the cobot to paint the window frame.

## 4.1 Data exchange between PC and cobot

A very important next step would be to automatically load the output variables onto the cobot. Whether this is done by creating variables and uploading them in a UR script, or by writing the variables to a file that is used by a script to generate paths to be uploaded to the cobot, or any other way. For now, the output variables are written to a .txt file with some supportive text around it to make it easy to manually copy the values to the cobot.

## 4.2 Path generation based on vision data

This may sound scarier than it is. A simple way to approach this could be to build default functions for actions that must be taken regardless and make them all dependent on the variables gotten from the vision application. From here on out, more functionality and options can be added to support a wider range of products.

## 4.3 Using vision to detect multiple objects

For now, the vision system is able to detect a single object. It can be improved further by making it possible to detect multiple objects and more complex shapes (like cross beams). Currently, manual selection of profile type and product type can resolve the need for more complex machine vision, but it may be a nice way to improve the system in the future.

## 4.4 Setting up an HMI (Human Machine Interface)

To make the system more user friendly, integrating an HMI into it may be essential. Once set up, an HMI can really show how an operator will work with the system. Not only will this give the project team more clarity on how the system works, it will show stakeholders how easy it is to work with it. A big obstacle for stakeholders without/with little experience with automated systems, is knowing whether it’ll be easy to use.

1. Microsoft Azure Kinect SDK GitHub: <https://github.com/microsoft/Azure-Kinect-Sensor-SDK> [↑](#footnote-ref-2)
2. KinectCloud GitHub: <https://github.com/widVE/KinectCloud> [↑](#footnote-ref-3)
3. CloudCompare website: <https://www.cloudcompare.org/> [↑](#footnote-ref-4)
4. CloudCompare wiki: <https://www.cloudcompare.org/doc/wiki/index.php/Main_Page> [↑](#footnote-ref-5)
5. Microsoft’s post regarding discontinuation: <https://techcommunity.microsoft.com/t5/mixed-reality-blog/microsoft-s-azure-kinect-developer-kit-technology-transfers-to/ba-p/3899122> [↑](#footnote-ref-6)
6. Orbbec Femto Bolt: <https://www.orbbec.com/products/tof-camera/femto-bolt/> [↑](#footnote-ref-7)
7. Orbbec’s SDK. At the bottom of the page, wrappers can be found. <https://www.orbbec.com/developers/orbbec-sdk/> [↑](#footnote-ref-8)