Imaging system for autonomous marine vessels

Table of contents:

Summary and Achievements:

We had a major progress in objective 1 & 2. We created a different rigid setup incorporating 2 IR sensors – Workswell WIC 640 Thermal InfraRed Cameras, with 1 depth camera - Intel RealSense D465, same one used in the first year. We used 2 approaches for utilizing the additional sensors in a stereo-vision fashion and showed that the additional data can be fused with the real sense sensor, as much as left as its own view. This allows us to assume the new system is more robust to different sight conditions. We showed success to some level in object detection and tracking based on objects position and velocity in the spirit of objective 2, using machine & deep learning algorithms in indoor and outdoor scenes. Using this setup as a system unit, the system can be expanded to a multisensory panoramical system by creating a fitting rigid platform based on both platform experiences from the first and the second years. It can be fused with other capabilities such as GPU/IMU units and augmented reality visualization as suggested in objective 3, which we leave for possible future research.

We choose to locate the thermal cameras in a similar way to Real Sense left and right cameras are located as a first choice without any prior knowledge for imitation of the real sense setup, which is found to be a good choice as we will show in the next sections.

For these purposes we pre-recorded a few datasets of 400 examples each with different features and worked with it offline rather than in real time, ignoring in some cases the high computational cost of the algorithms. Accelerations can be done, and we leave both tasks of acceleration and real-time adjustments out of this work's scope, as it is a technical and not that hard task.

- For the record, we are the among the first to use deep learning on this scale of amount of thermal data, as the biggest available public dataset contains only about a 190 examples (thermal stereo pairs and disparities/depths) and used in such approach recently, which was published in a small paper in 2022.
  https://publications.waset.org/10012412/depth-estimation-in-dnn-using-stereo-thermal-image-pairs

The technological setup & developing environment and tools:

The cameras used are Real Sense, 2 WIC IR cameras on a mobile laptop recording stand with a power battery (left side) with 2-3 hours recording capacity with a 1 TB SSD hard drive. The sensors and the devices connected to a USB hub which is connected to the laptop.

Both thermal cameras are hardware synchronized by the connections you can see in the image (2 connections hanging below the thermal cameras). The recording is done in a fast C++ multi-threading fashion (each thread writes a frame simultaneously from each camera and independently) directly into mapped memory from the SSD (right side) and managed by the OS that way. The writing is done directly writing raw bytes of RGB and depth data without any preprocessing.

Real sense camera is recording with laser emitter enabled to get best depth estimations, a matter which is greatly

utilized by the deep learning approach for stereo-vision later.

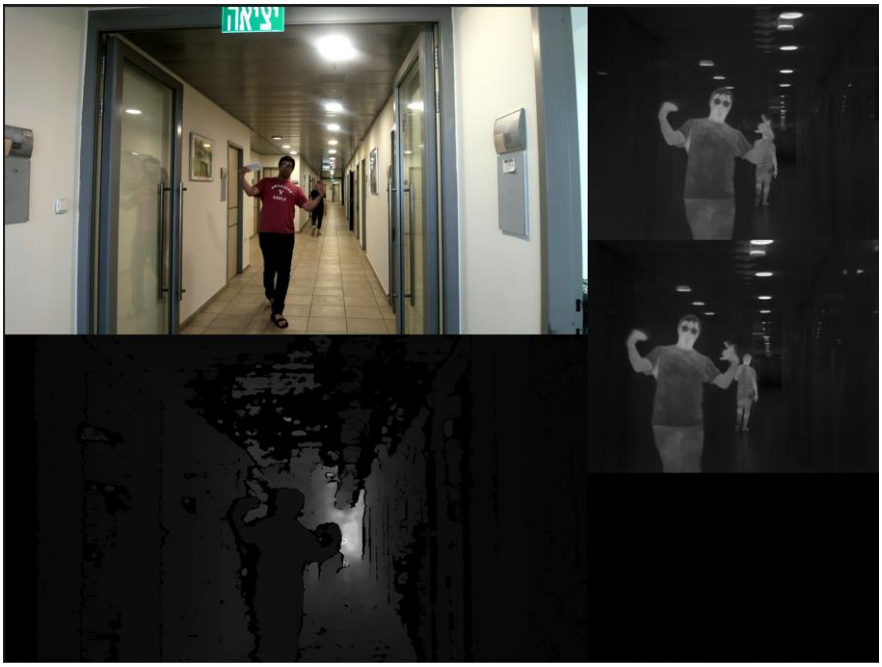
Real-sense camera and WIC cameras synchronized in a software fashion using frames arrival timestamps (as fast as the frame's thread gets the frame bytes).

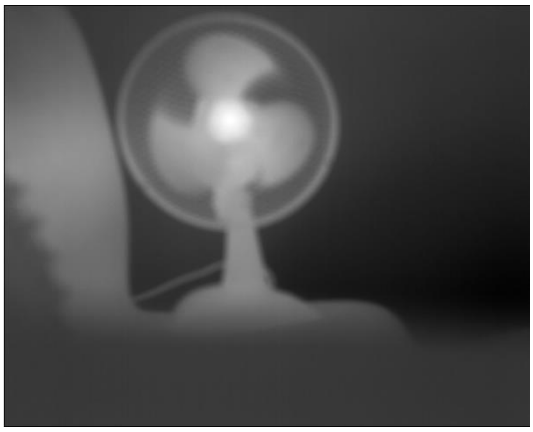
The synchronization, software and hardware is good enough to some velocity / frequency of the moving objects, but even the hardware synchronization show flaws dealing with high frequency fan spinning, see illustrations below.

For both used official SDKs supplied by the cameras manufacturers. Recording is done in indoor and outdoor scenes, in stationary and moving modes. For the computational acceleration needed for the deep learning approach GPU RTX 2080 was used.

Relating to the development platforms, as was mentioned before, for best recording performance C++ was used with CLion, for processing the recorded data mostly Python with PyCharm were used and for the stereo-vision, object-detection and tracking Matlab & Python were used both.

Sensors Synchronization test illustration:

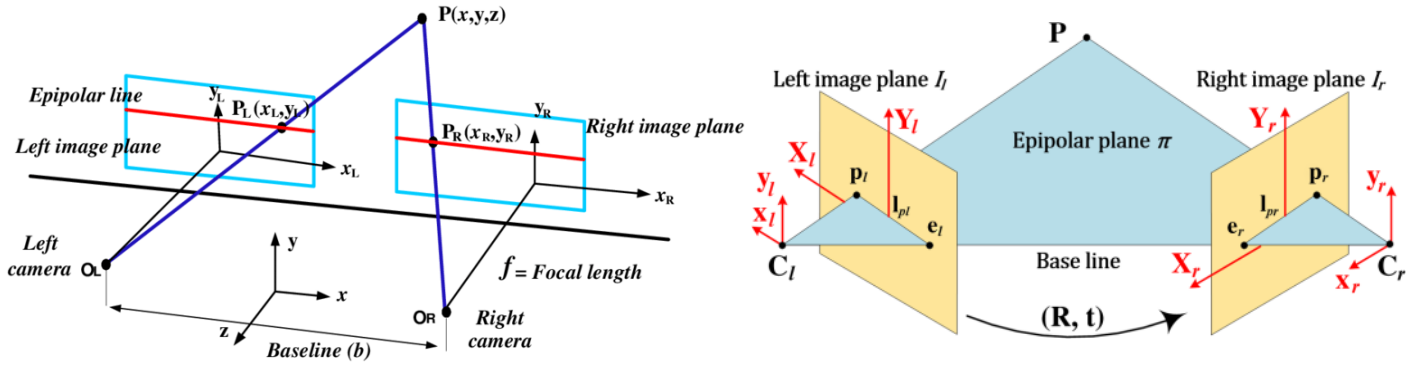




Thermal Stereo-Vision – Classical Computer Vision Approach

In this part, as was mentioned we incorporated into this project 2 thermal cameras FLIR TAU 2 longwave sensors and used left and right images from them to create a stereo vision system, which estimates the depth of each pixel in one of the cameras (in our work we choose the left as the axis camera arbitrary), and thus can be visualized for example as a point-cloud (3d data).

The classical approach:



The classical approach suggests a process of calibration between the cameras, for which we used a chessboard image 8 x 10 in good enough light conditions (otherwise the chessboard can not be seen at all). We recorded thousands of frames in different positions and angles of the chessboard and then filtered the best of them which imply minimal reprojection error, a measure which is mostly considered when creating a stereo-depth estimation out of two images.

It is interesting to mention that direct stereo-calibration got more accurate focal length estimation, then classical single camera calibration for each camera and only after that stereo calibration fixing the single estimations or using it as an initial guesses for the intrinsics estimation.

Then we used the calibration information (estimated intrinsics of the thermal camera and extrinsics from one camera to another) to perform stereo-vision classic operations such as rectification of the images, stereo-matching (a process where for each pixel at one image the corresponding pixel in other image is found, which is approximated to be the same one from the first, i.e a matching process, to estimate the disparity of each pixel and therefore object, i.e the

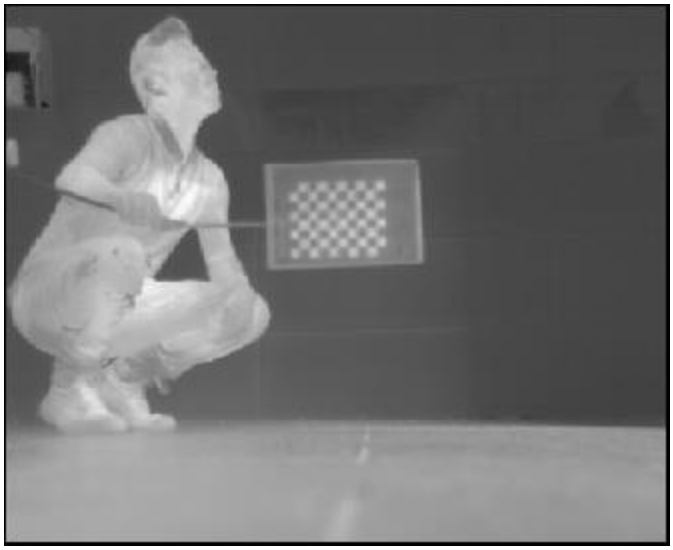
horizontal distance between the pixels, which is utilized to find the depth of the pixel for each one in the first image) and reprojection to an x,y,z coordinates, where z is the depth, i.e the distance from the camera.

In this approach the stereo-matching process using the classical stereo-matching algorithms such as SGBM showed good results in good light conditions (a lot of light and heat was in the scene) and less good results in a scenes where there was worse light.

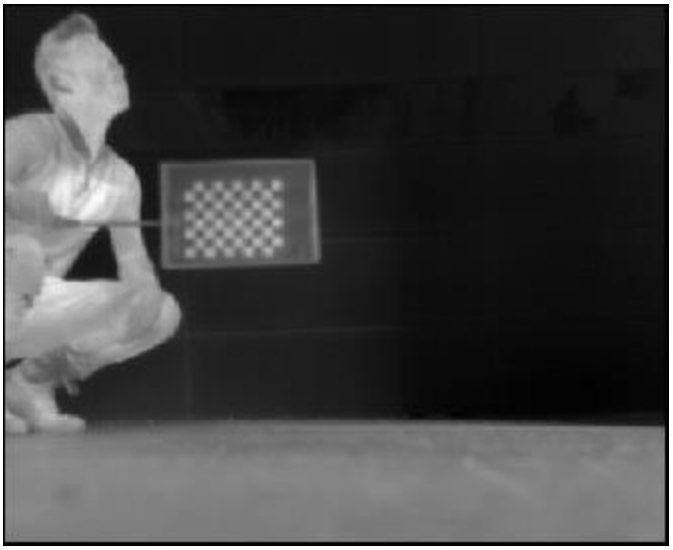
That why we incorporated a deep learning approach, i.e the use of a deep convolutional neural network which takes as input left and right images of the thermal cameras and predicts the disparity or depth. Better and easier results were achieved for direct depth estimations, result for which we present below.

Illustrations:

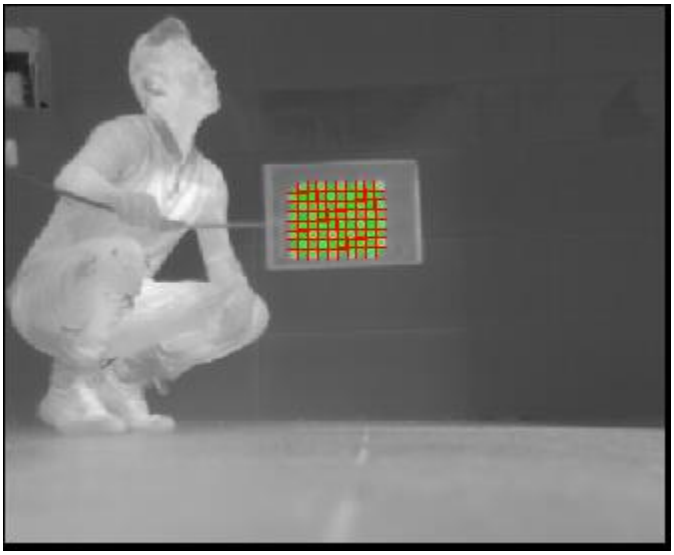
Left camera image example:

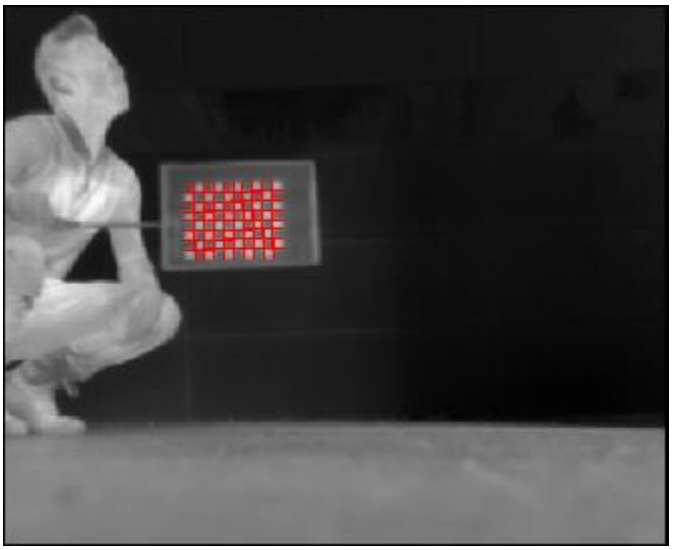


Right camera image example:

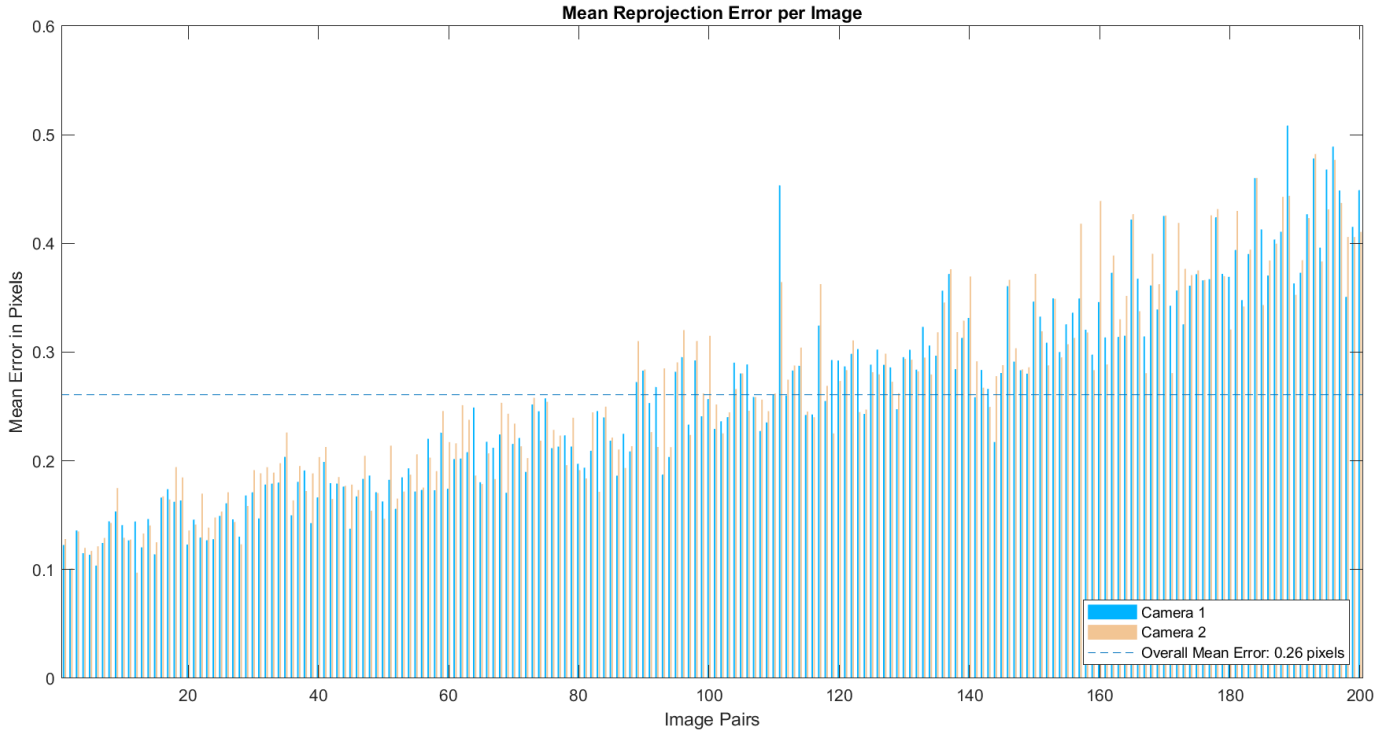


Detected and reprojected chessboard points:

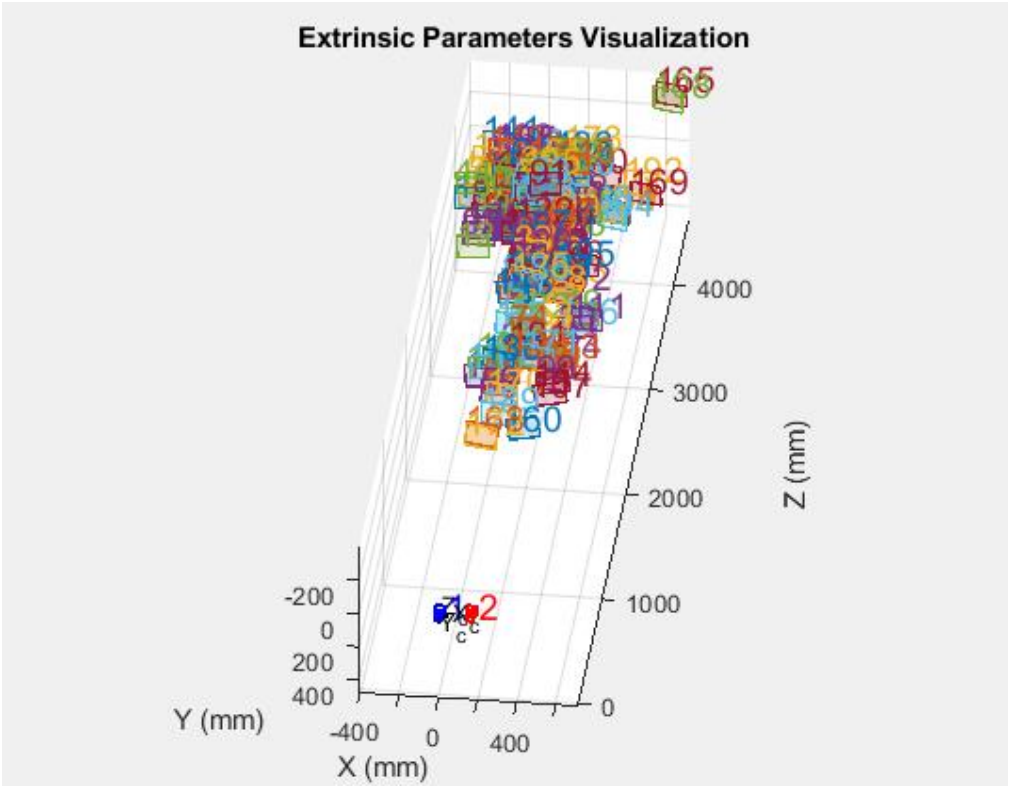




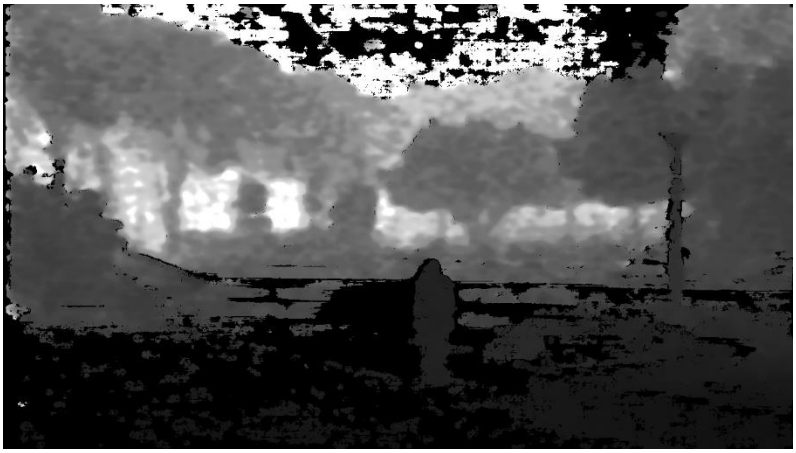
Reprojection errors graph:



Cameras image (extrinsics visualization) based on the calibration:

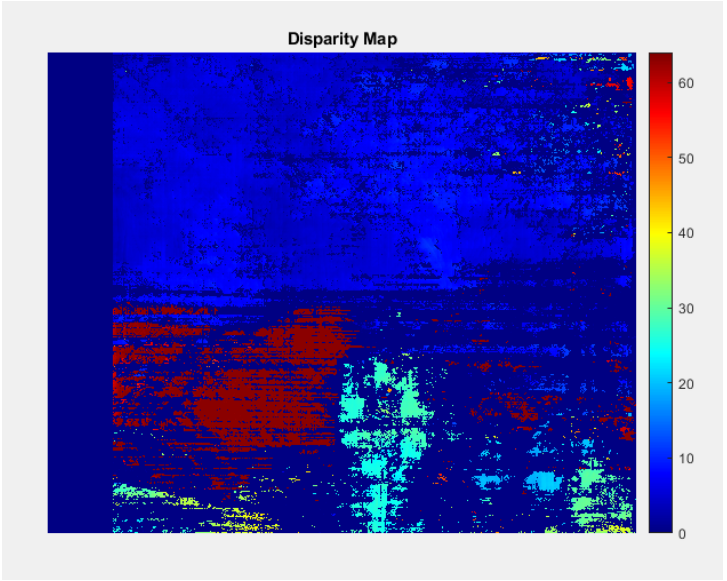


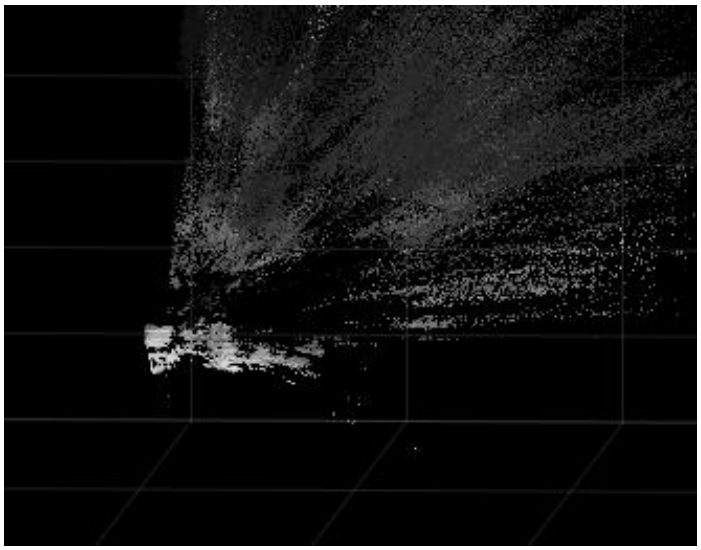
Comparing Real Sense 3D (about 900000 3D points depends on the scene) vs Ours:

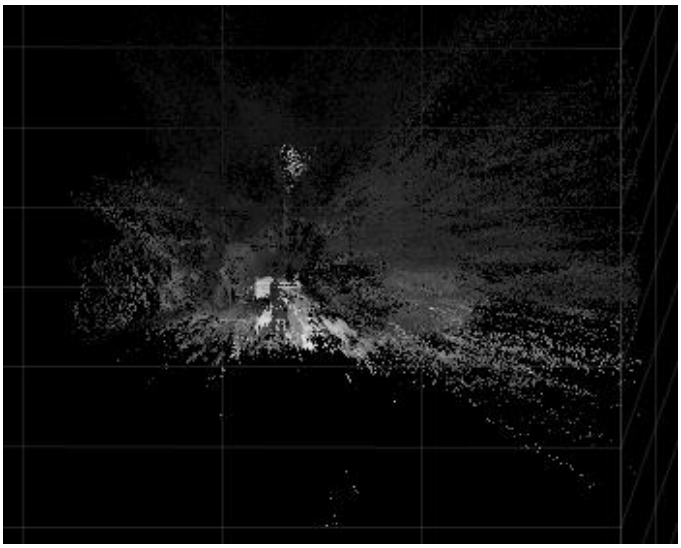


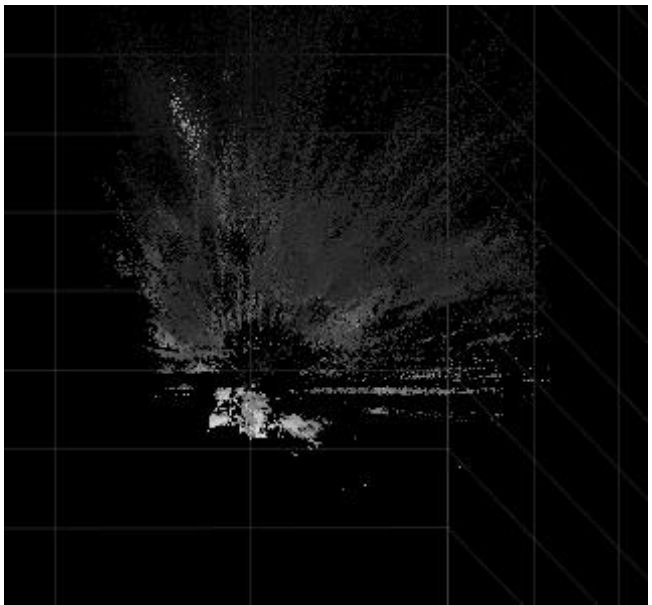
We've tried and show here 2 implementations of the matching algorithms, each with its own results:

SGM matlab method: 260000 3D points :



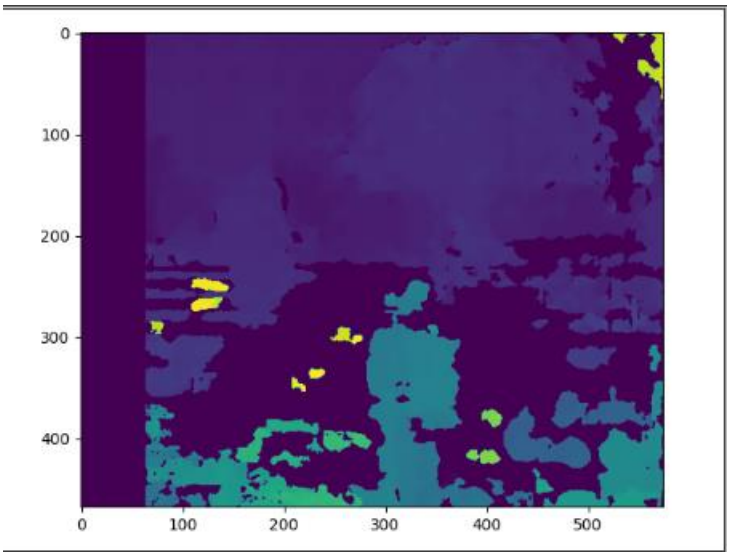


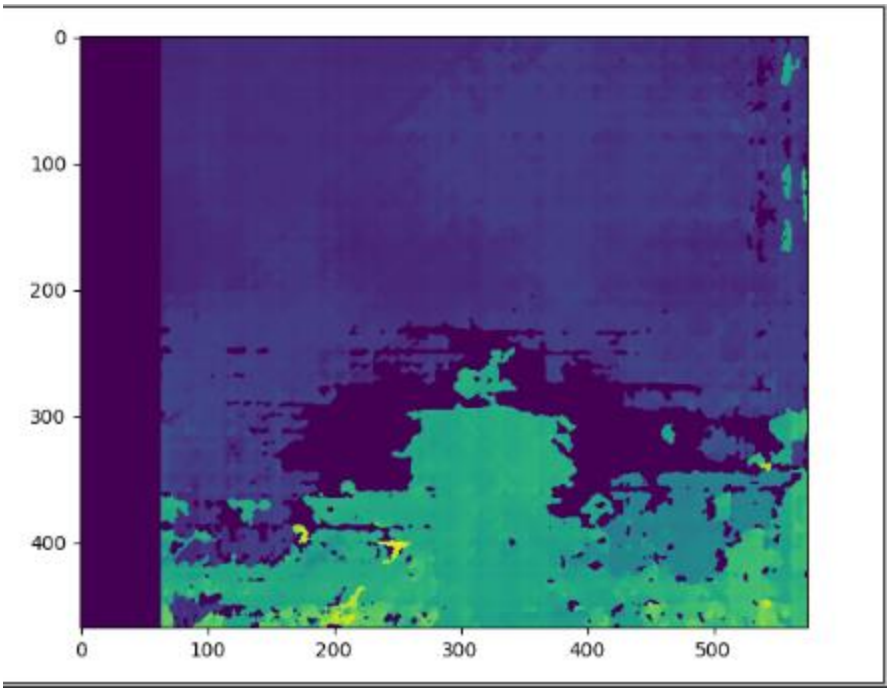


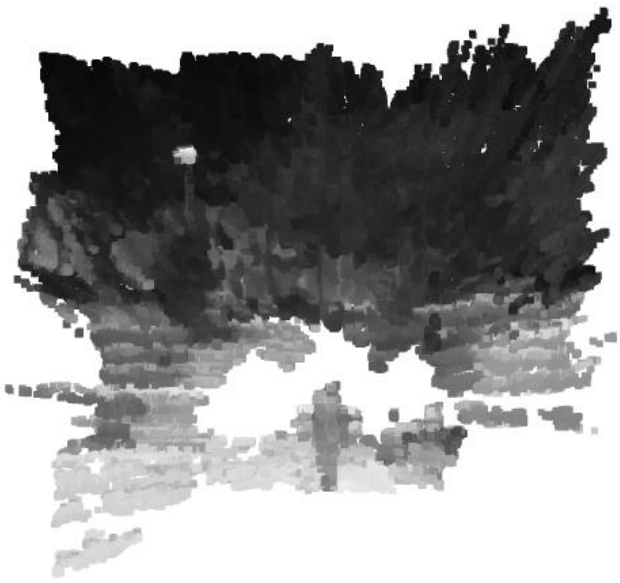


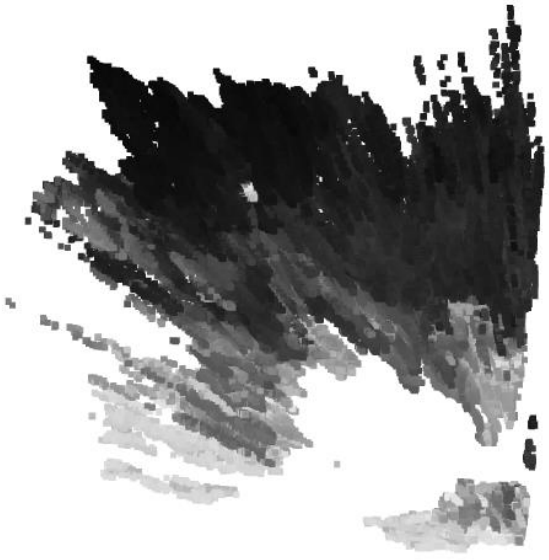
Different Block size parameters (with rectification of the images):

SGMB python openCV method: 150000-200000 3D points



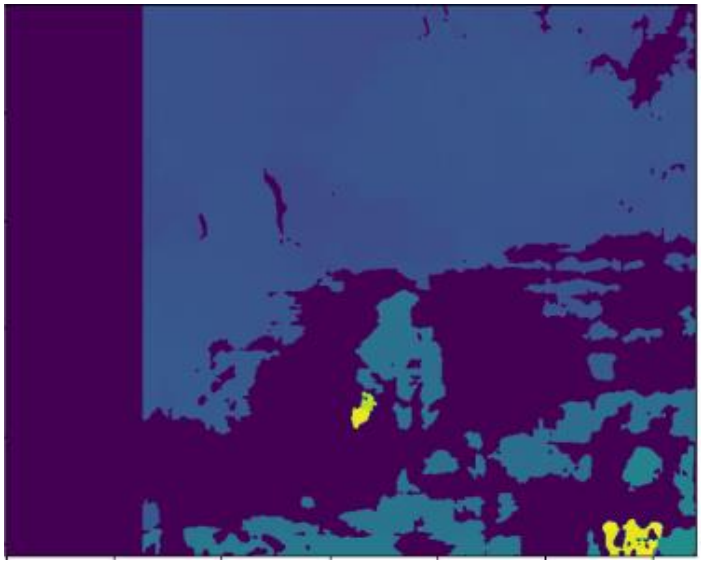

0
100
200
300
400
0
100
200
300
400
500





Without rectification of the images:

SGMB python openCV method: 200000 3D points

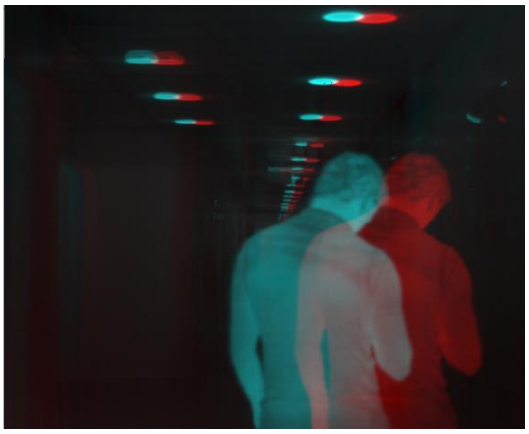


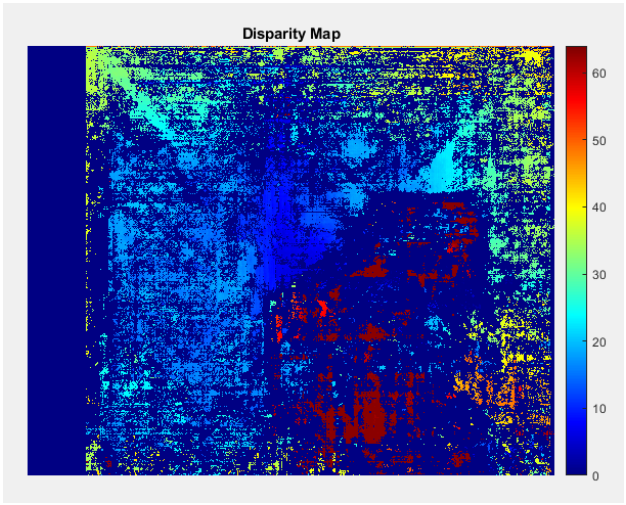
But in more harsh scene (in this example we use indoor scene: less light, less heated objects) good results can be difficult to achieve.

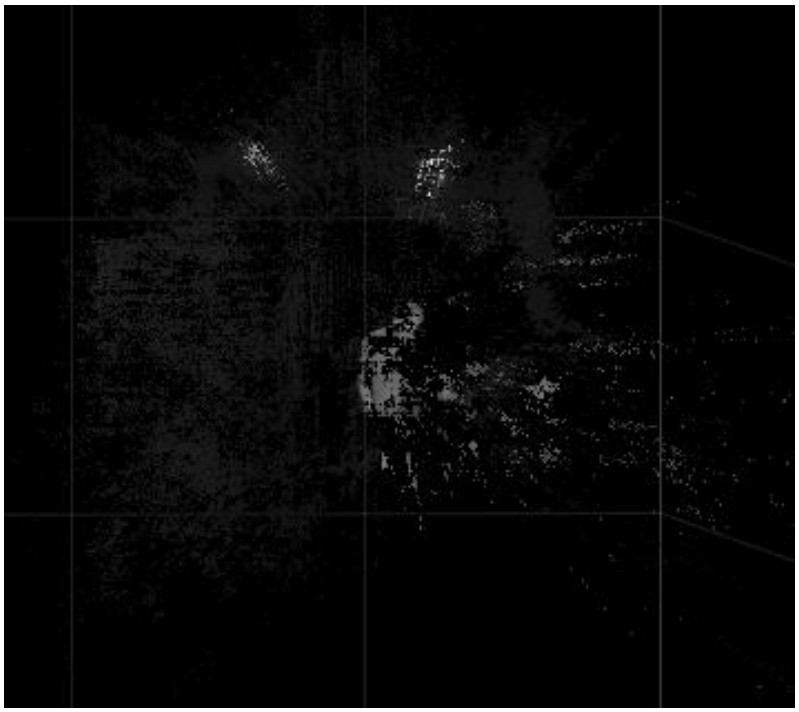
Illustration:

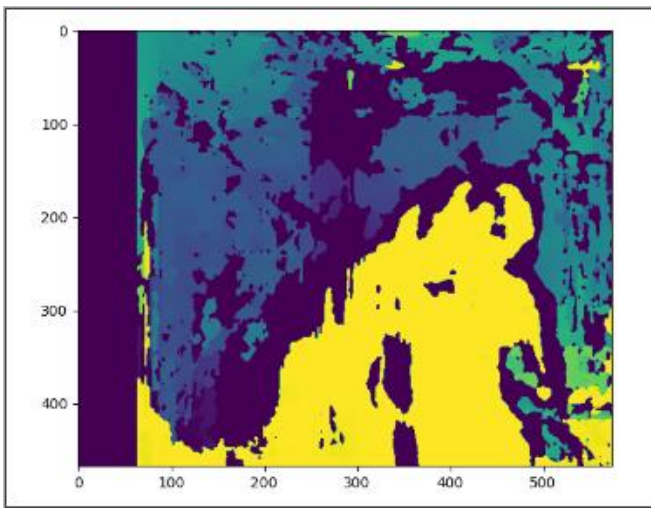







Disparity Map
60
50
40
30
20
10
0




0
100
200
300
400
0
100
200
300
400
500

Real-Sense and Thermal Stereo-Vision Fusion:

In addition to noise and such difficulties, there is also a matter of fusing the pointclouds for the next segmentation goal. One way to go is indeed to put them together by calibrating the left thermal camera with the real sense camera and then by using the intrinsics and extrisics to match the pointcouds as it is shown here with 2 Kinect cameras:

https://nishantrai18.github.io/resources/caris_presentation.pdf

Another one is to look at them separately without additional effort.

The issue of the first approach is that in the example this is done for 2 same cameras (same resolution, same intrinsics) with much better resolution and color cameras, which implies very good calibration results, much better then of thermal pair or thermal and color pair.

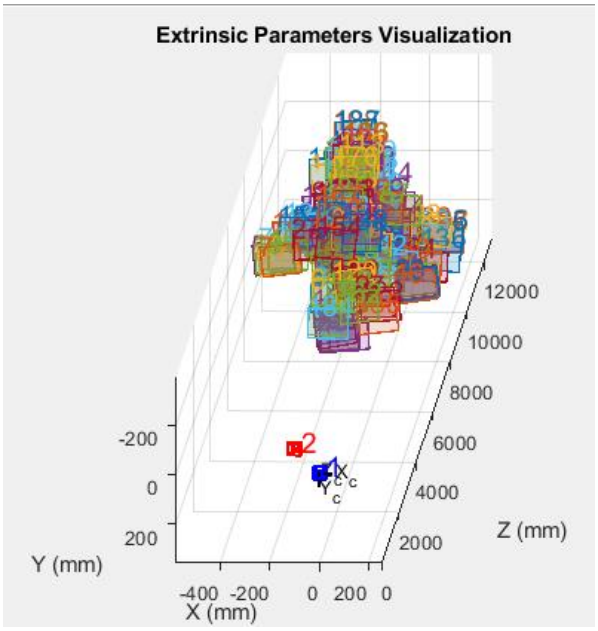
We believe it is possible as the results of the not-trivial calibration of color-thermal images (different color space, resolution and etc.) were promising (although less good then for a thermal pair of images), but with the time limits we had, we used the calibration results as an initial guess (as they weren't very accurate) and matched the pointclouds manually by some fixed objects in the scene that both cameras view, and then refining the matching even more with a moving object in space, refining the matching parameters (scale, rotation, translation) according to the position off the object. This gave not that bad results. Furthermore, we also tried the use of Iterative Closest Point algorithm to refine it even more. This algorithm considers a global matching and a refinement with local matching. Our manual matching, which can be done only once without too much effort can replace the global matching using IPC, which is computationally expensive, and some additional matching to the manual refinement can be done by the local IPC. Visualization of the last will not be illustrated below because it isn't able to be seen as it spread the points of the thermal pointcloud on the real sense pointcloud. We believe it can help the segmentation algorithms to cluster better as the density of the objects gets higher.

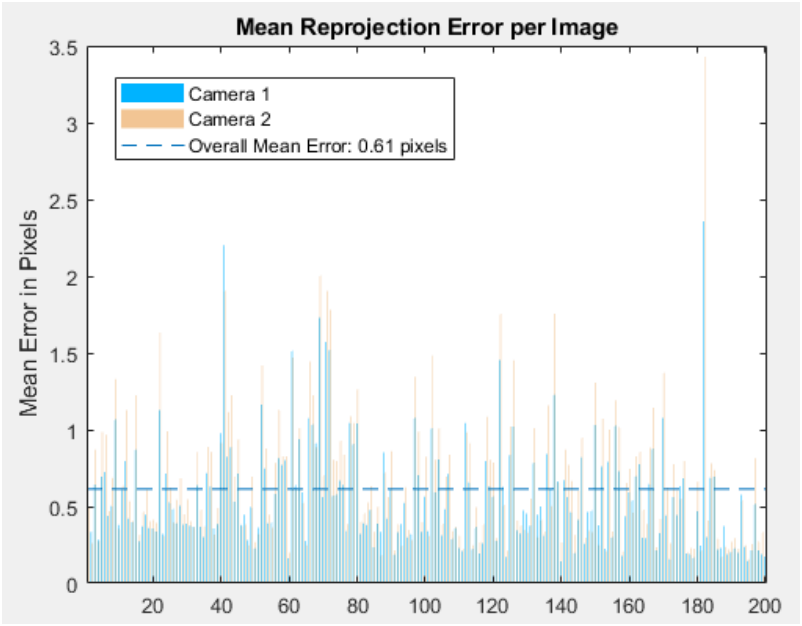
Although this process can give valid results of sensors fusion in such way, we saw it as a challenge to improve and automatize this long and challenging process which implies different noise and difficulties on different steps of it.

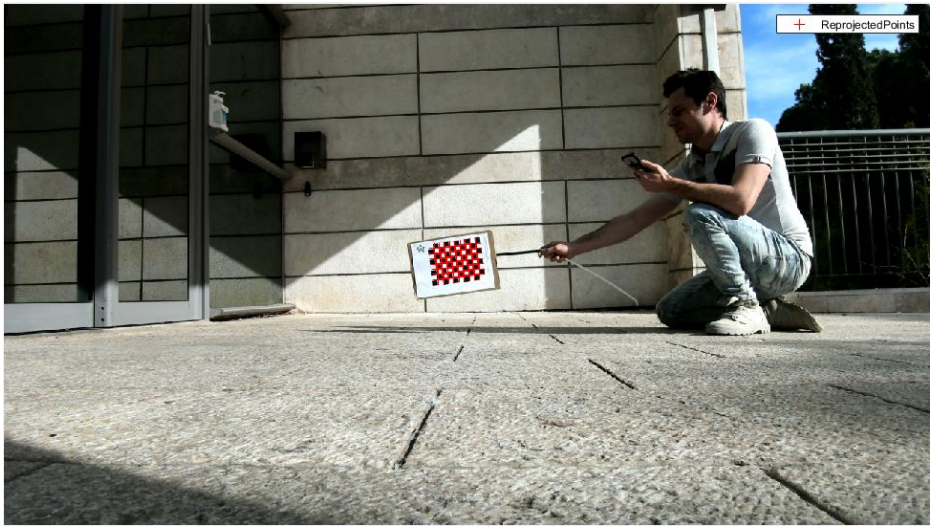
Also there is an effect of reflections on classical stereo-vision approach, which is caused by the fact that the matching algorithms can't differ a reflection and the actual object.

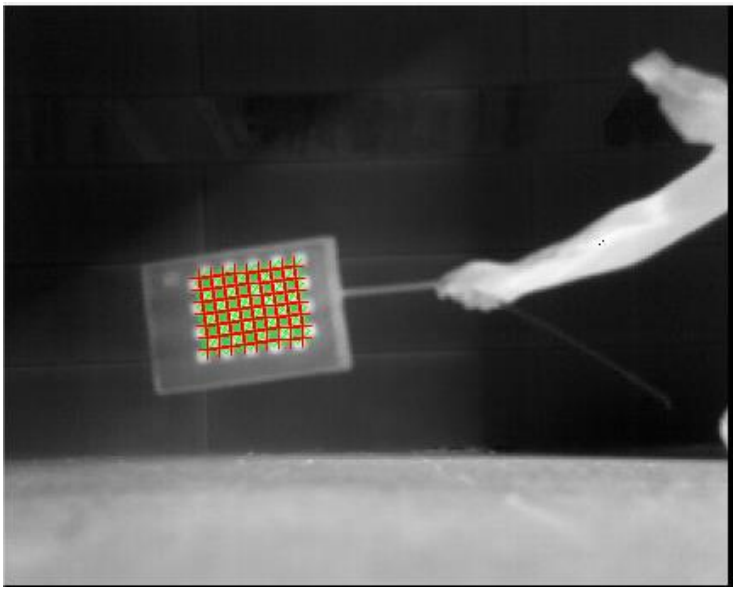
To deal with those, we dedicate the next section of incorporating a deep learning approach.

Illustrations of the mentioned above Realsense-Thermal cameras calibration and matching of the pointclouds with the manual adjustments:



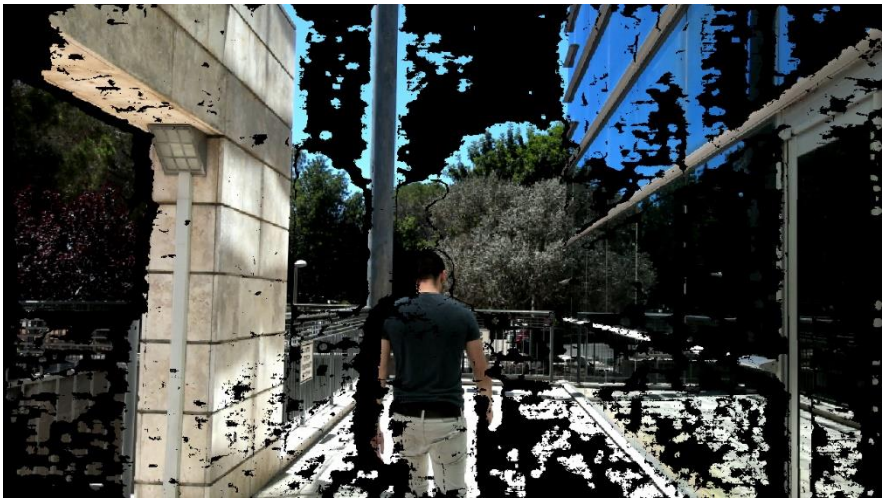


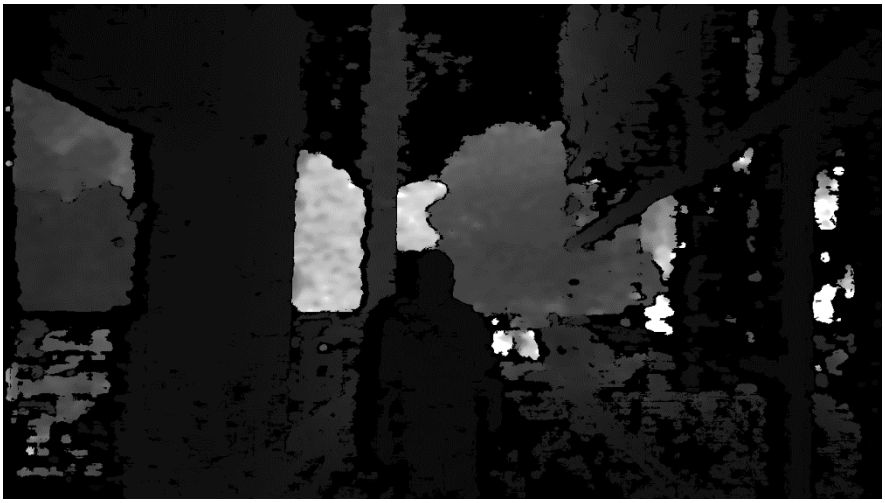




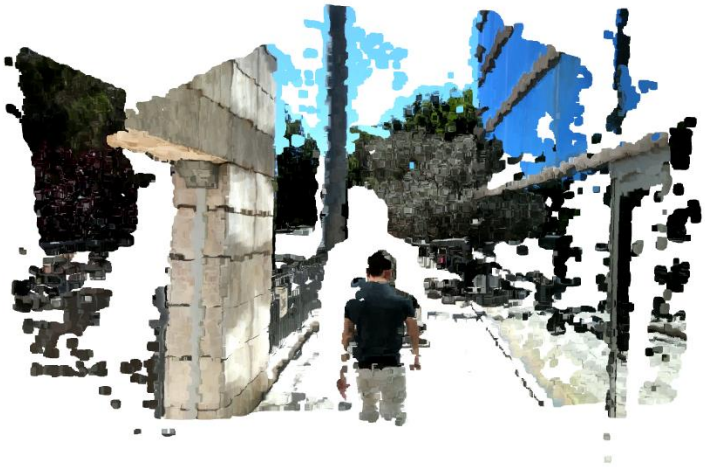
Pointclouds Matching results:

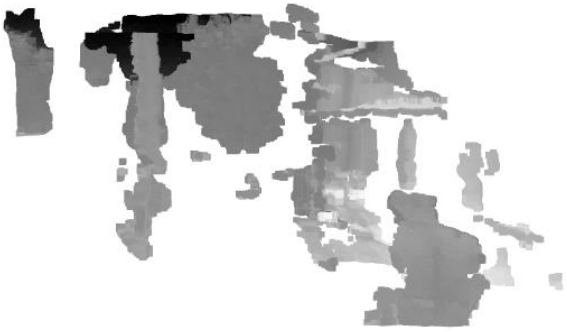
Real sense image & depth:



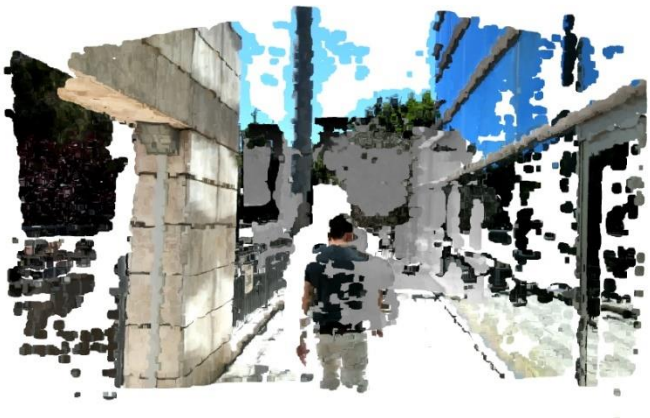


Rectified Thermal images:



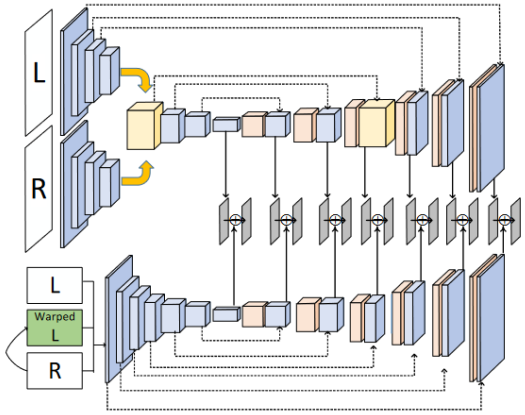


Manually adjusted and stitched:



Here for the sake of the proof of the concept we show only scaling adjustment, of course with rotation and translation transformation, which can be adjusted due to second frame with the same objects in different locations (for example the person in the scene moved forward), the results of this matching can be even better.

Thermal Stereo-Vision – Deep Learning Approach



FADNet: A Fast and Accurate Network for Disparity Estimation

We used the FADNet Network from a paper published in Mar 2020 for the task of supervised learning. We created the labels in a way is similar to the small public available dataset:

CATS: A Color And Thermal Stereo Dataset - VIMS Lab

Utilizing the fact that our thermal vision view is included in the real sense wide view, we projected the color and the depth of it to left and right thermal images. For the projection parameters estimation the previously calibration results were used and a manual refinement was done. As a check of a good projection results we were able to look at the color pixels projected on the thermal pixels and by putting them one on another, we saw that our projection matches even in the smallest details of small groups of pixels n the scene. Another confirmation of the good projection results also are seen beside the learning results, is a good y-axis disparity of only two constant values (which means we got the exactly the distance in pixels of one camera disposition from the other), which we will mentioned again below.

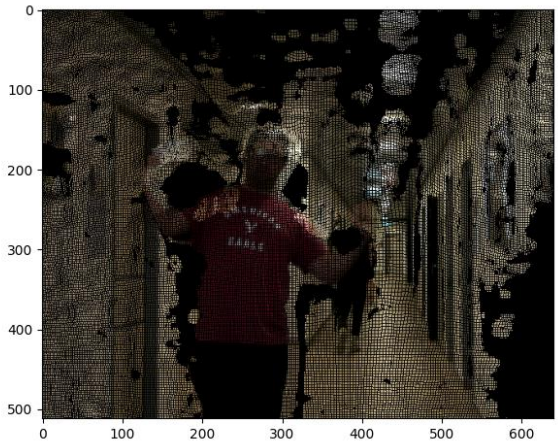
By the projection we were able not only get the depth for the left or right thermal pixels, but also knowing the projection locations to each left and right thermal images, we had the exact disparity values (the difference of the x positions on the left and right thermal images). In such way we got 2 birds at one shot.

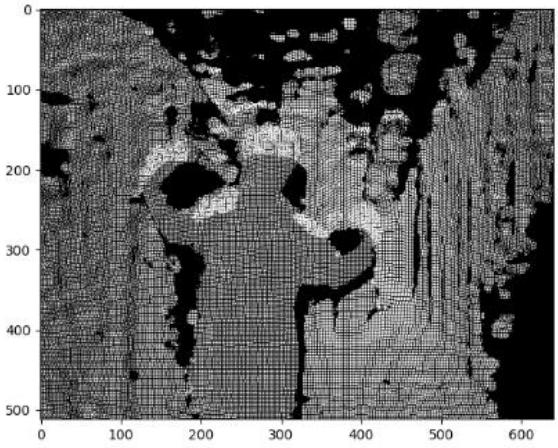
Here the example is for the left image, but same was done for the right image (projecting the real sense points to the right image and using the same projection for the depth points)

Left and right thermal images:

Original Real-sense image:

Projected Real-sense color and depth images on the left thermal image:

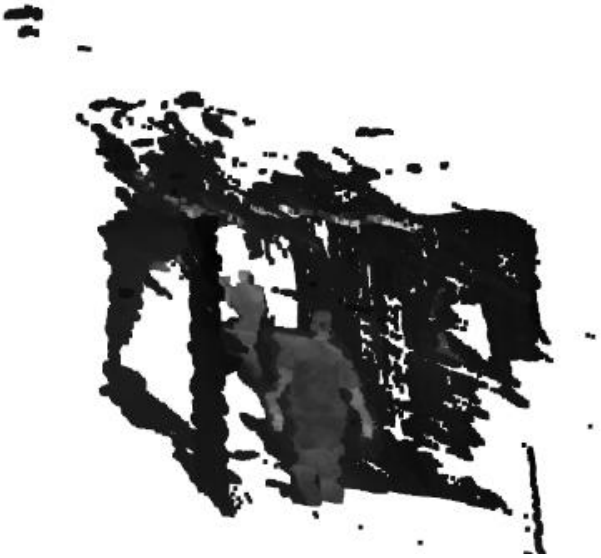




another example with 3d visualizations:


Projected Real-Sense points (color in this case) onto the left and right thermal images:


deduced Disparity and projected depth respectively:



In the pointclouds images above it can be seen how well the depth fits the thermal data. The pointclouds created by using the projected depth and the original thermal image as its color chanel.

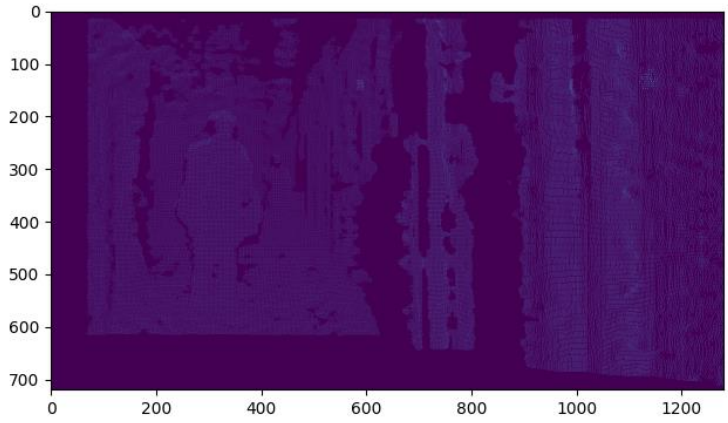
As was mentioned, another clue from the projected data that showed us that even though the projection was refined manually it is good is the fact that on the projected y axes coordinates we've got in all of them difference of 14-15 pixels between the left and the right images, which means that by subtracting 14 or 15 we've got a manual rectification (which we indeed did) with an error of only 1px in approximately in around a half of the pixels (the error was spread around the pixels chaotically without any pattern and can be though as an error of approximation as our manual rectification in this way can only rectify to integer pixels values and not for example half of pixel like a full calibration-rectification process).

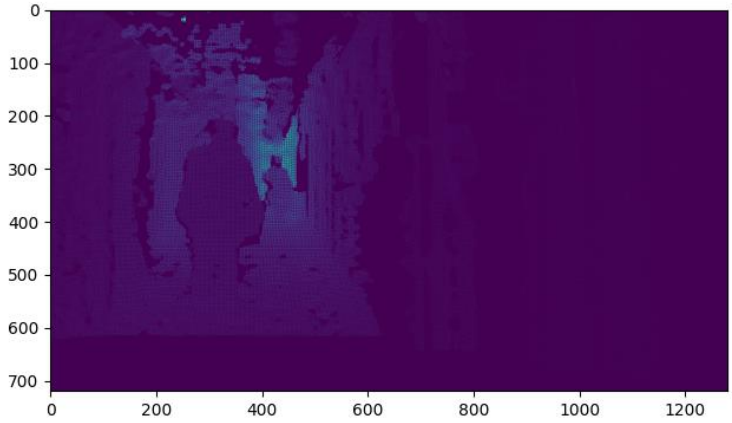
Traditionally the relation between the disparity and the depth is given by following formula:

$$disparity = \frac{Bf}{Z}$$

Where B is the baseline distance between the stereo-cameras and f is their focal length.

This opposite relation can be noticed in the images above and here:





In a more practical way we decided to find the numerator, i.e the multiplication of the baseline and the focal minimizing a mean-square error on our samples on the difference of the computed disparity and the depth using ***gradient descent***, for which we found 470 is that number.

For our surprise we found that for the real data we've got and the disparity was computed, this relation wasn't exact even for the best estimated value of 470, i.e there was an additional error of the disparity to depth transformation, to prediction error.

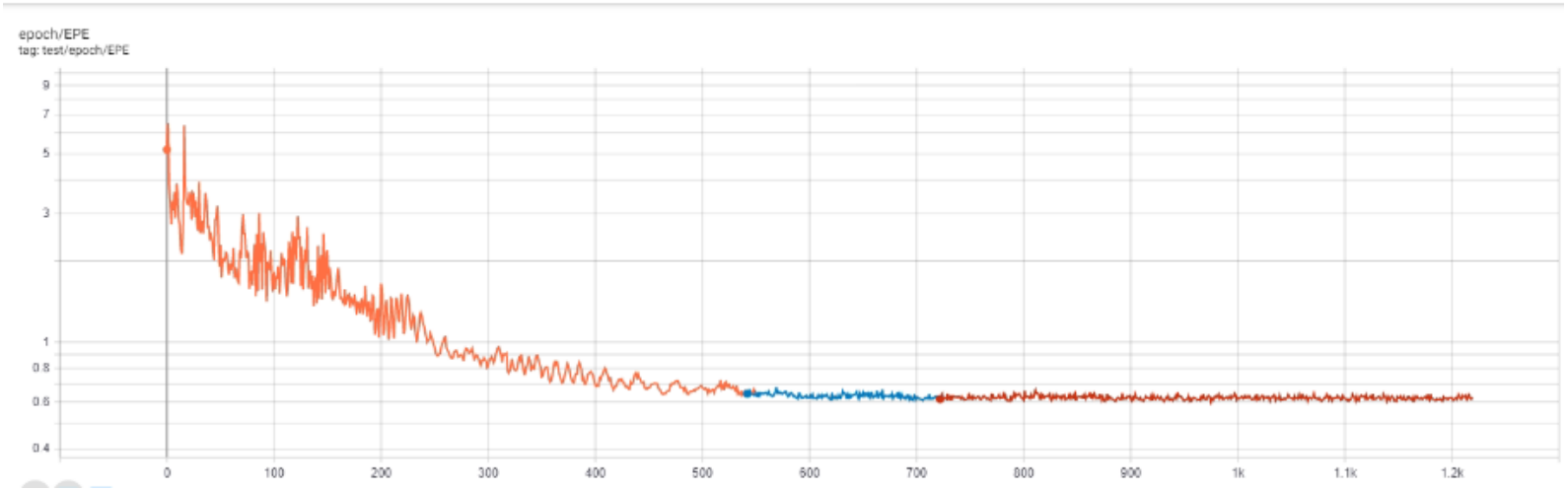
For that reason, although our network managed to learn also the disparity as we created it according to the projection, we preferred to work mostly with the results gotten from the process where we taught our network directly predict the depth from the stereo-pair of images

We believe more experiments should be done to estimate which is actually better and how can be both incorporate to achieve even better results then each one of them separately.

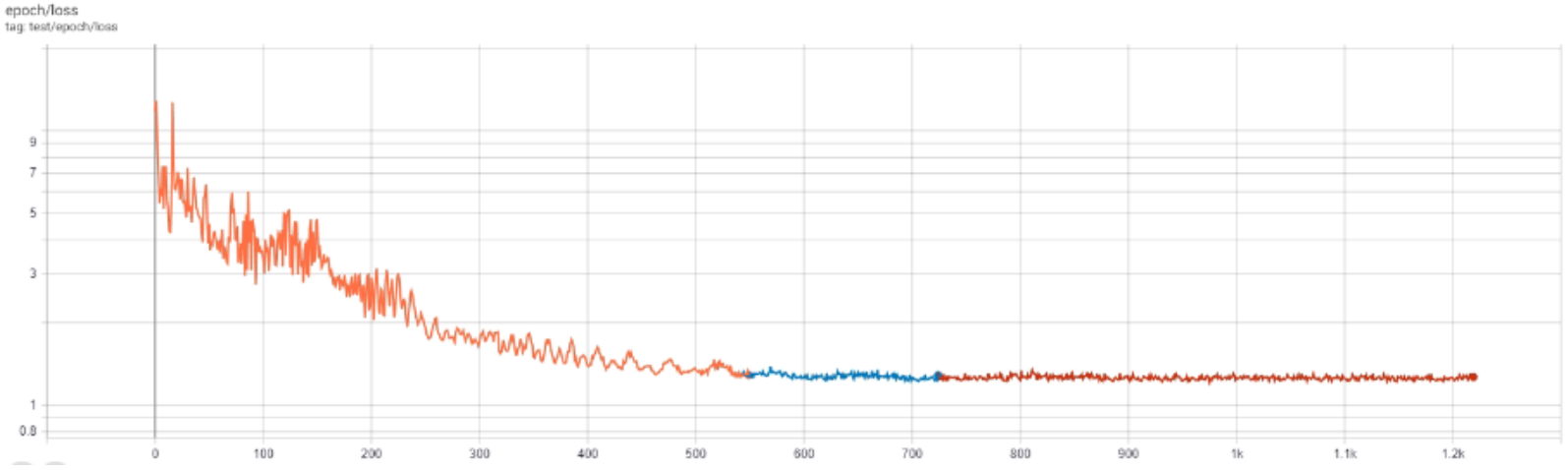
Illustration of the training process and results:

Training:

EPE (error-per-pixel) measurement – mean error of all pixels per image.



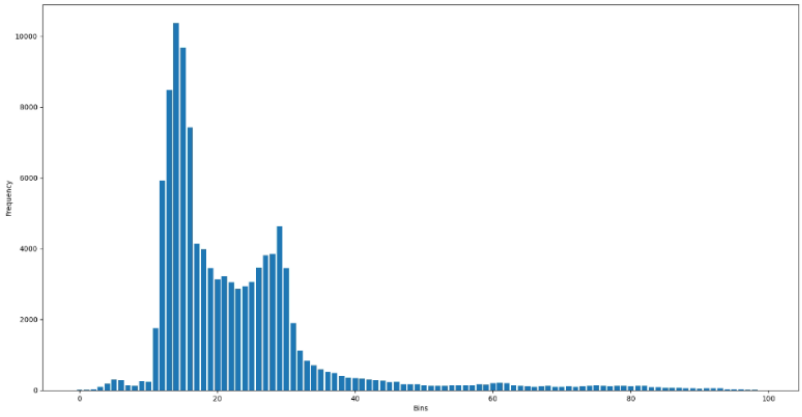
Total loss:

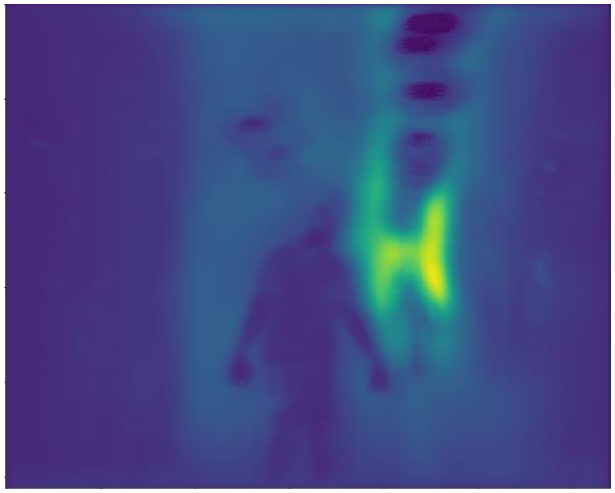


Results: *The network filled the holes (continuous smooth predictions)*

Output:

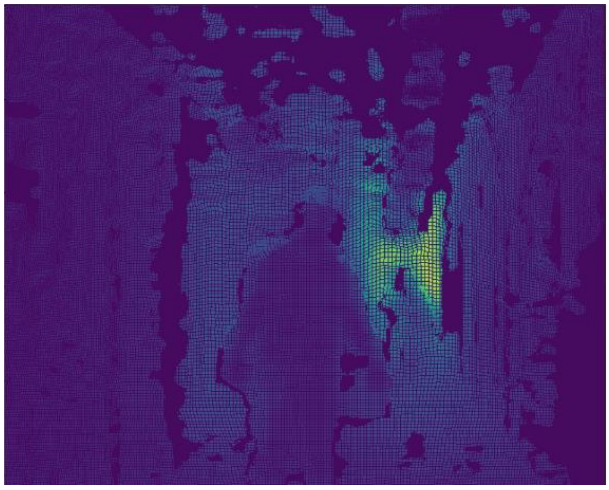
On the left below is the histogram of the predicted values



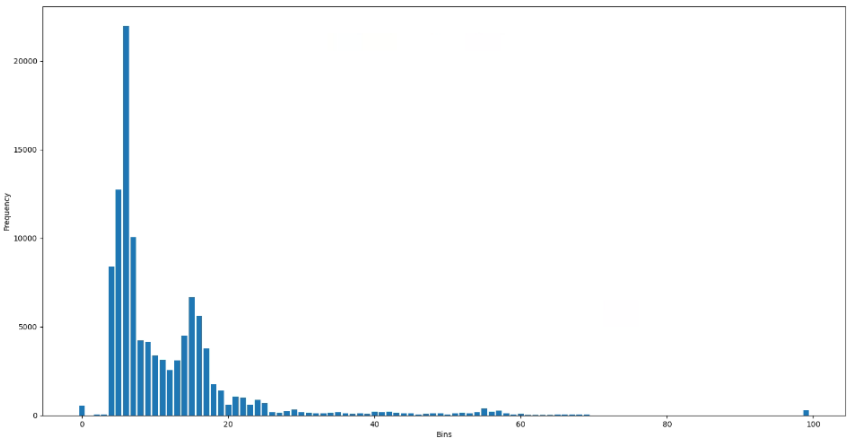


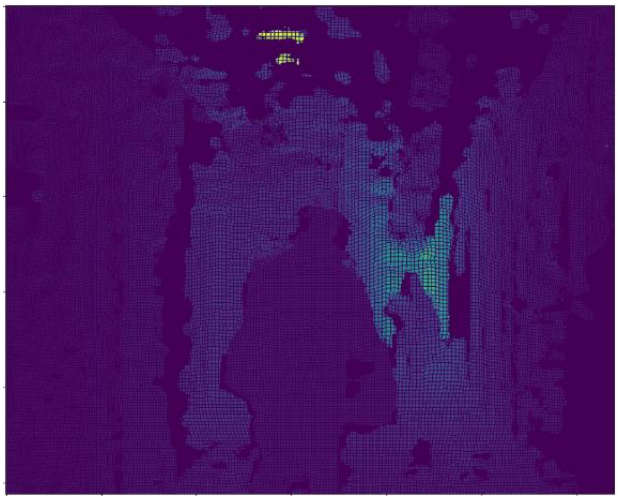
Same results as above, but with the mask

of the values were missing in the label (holes)

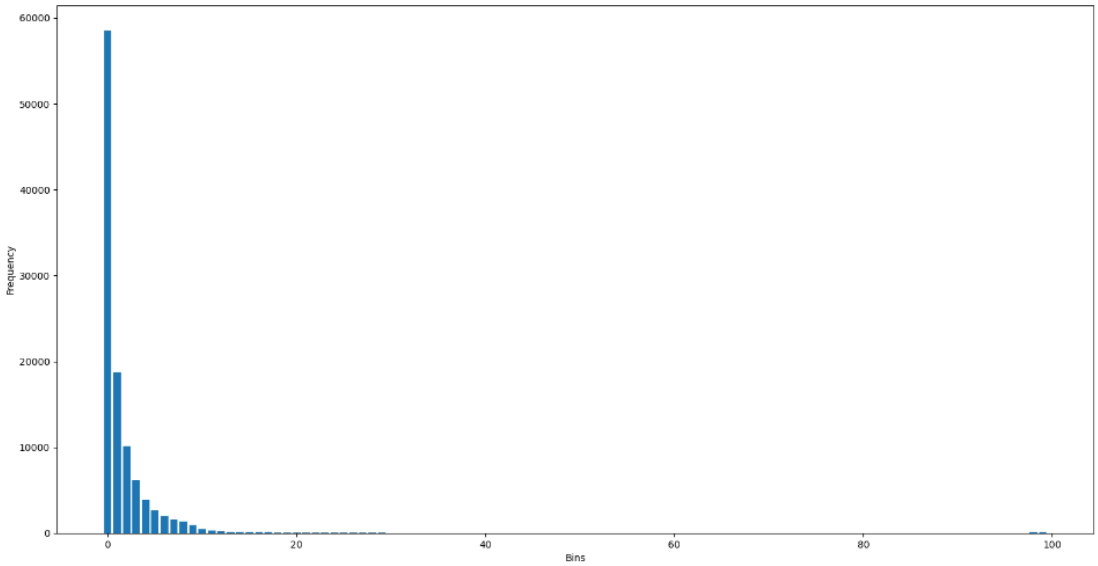


Label (Groundtruth):



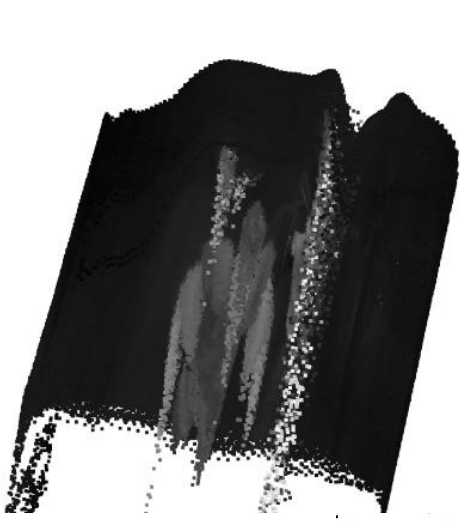


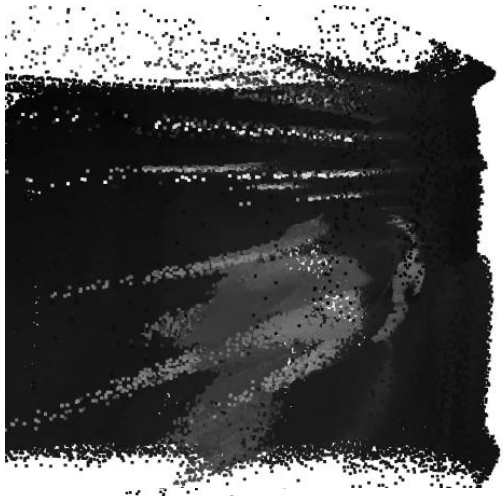
Difference (absolute value) between the predicted and the groundtruth)*:*

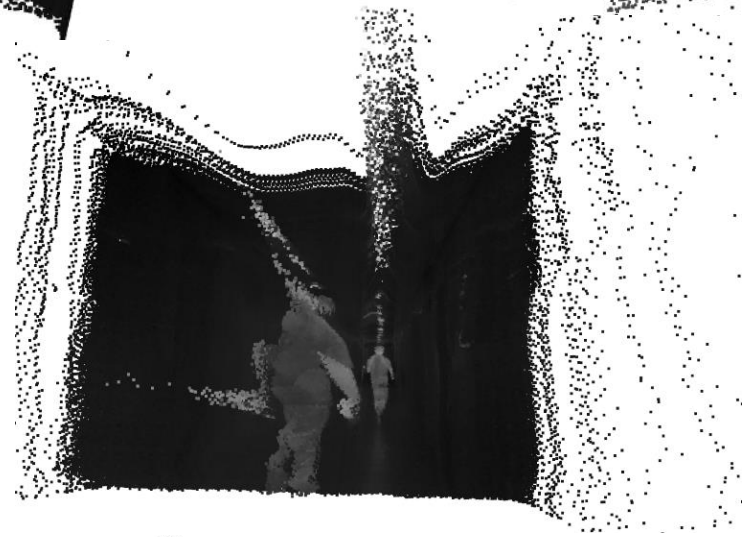


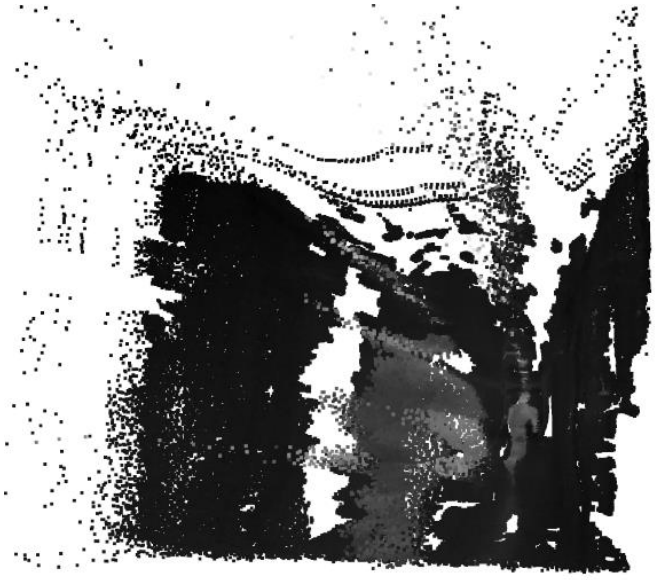
As expected the error is mostly around 0 because of the good results of prediction and also I gaussian (absolute value thus it is symmetrical more or less).

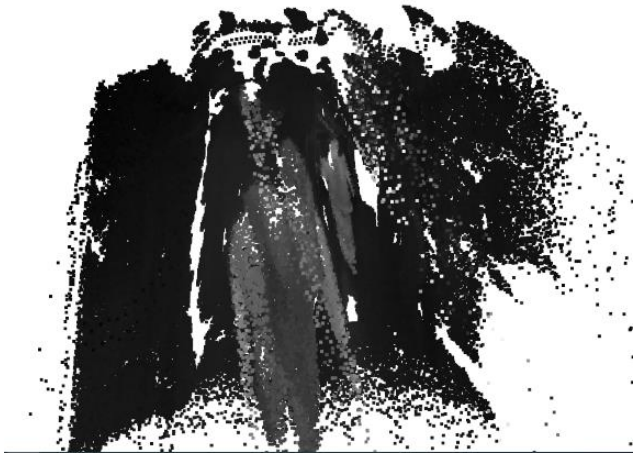
Predicted Pointcloud :





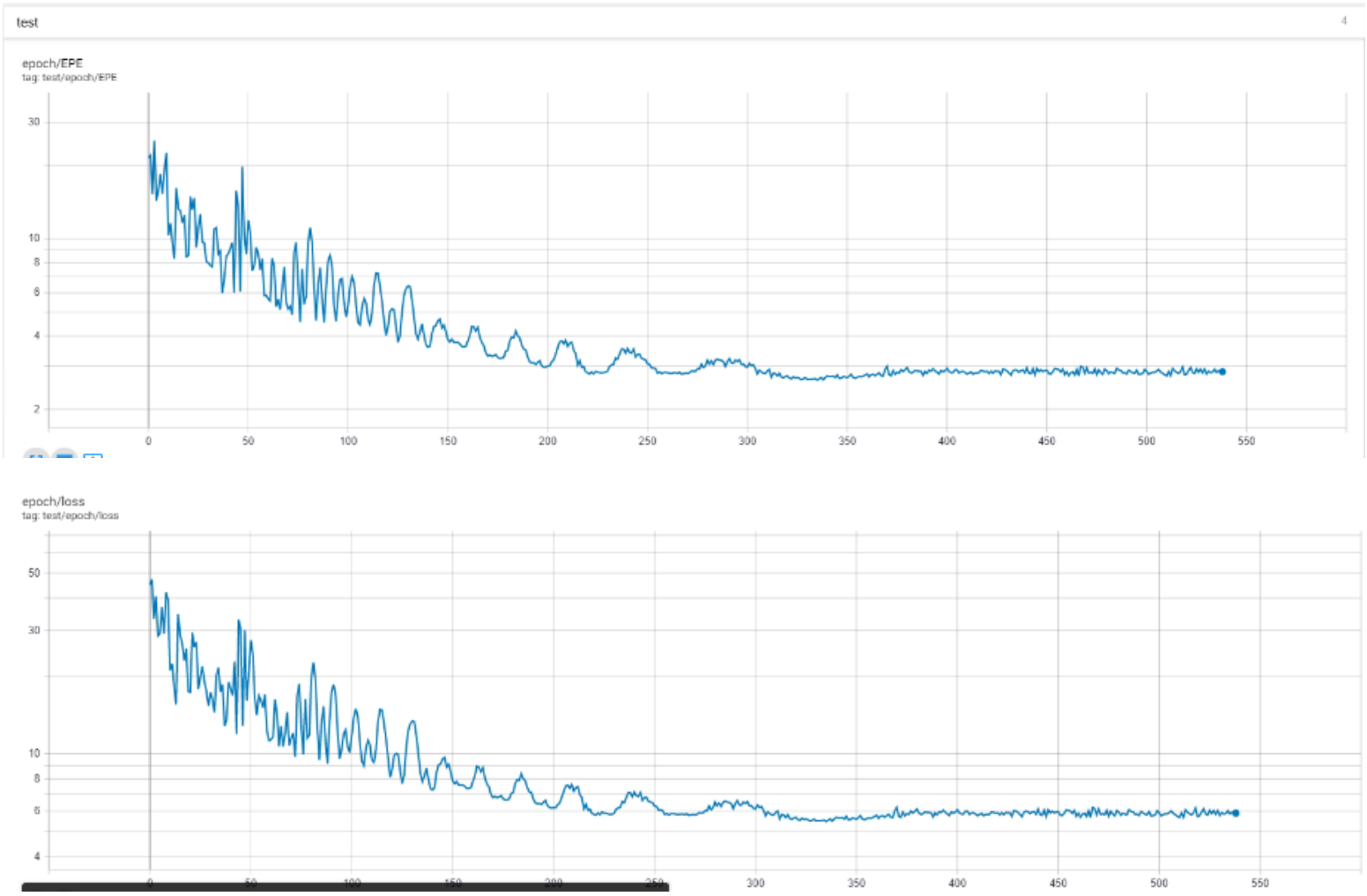






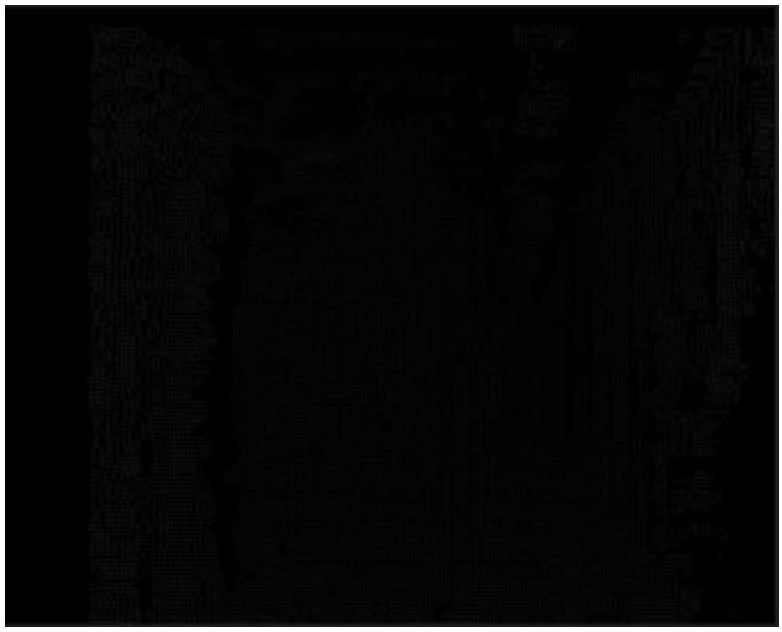
Although the histograms look similar from a structure (same pattern of distribution) point of view and the error seems to fit the low EPP of 0.61 (meters) was achieved, it looks like there is an effect of spreading to some level.

Disparity learning training :



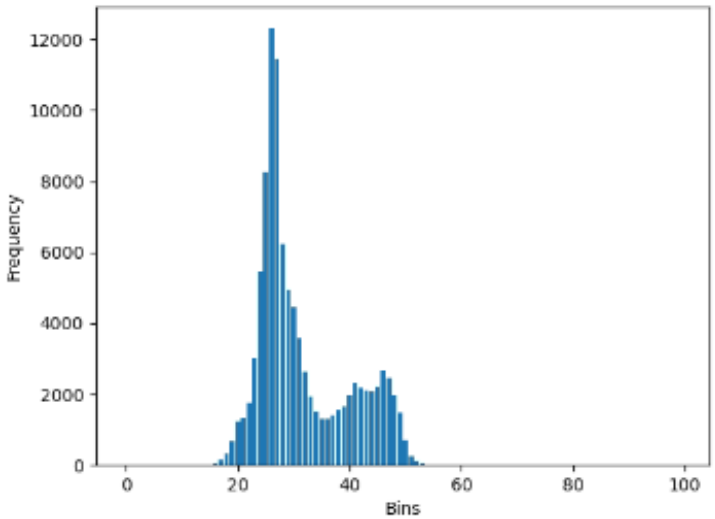
groundtruth:

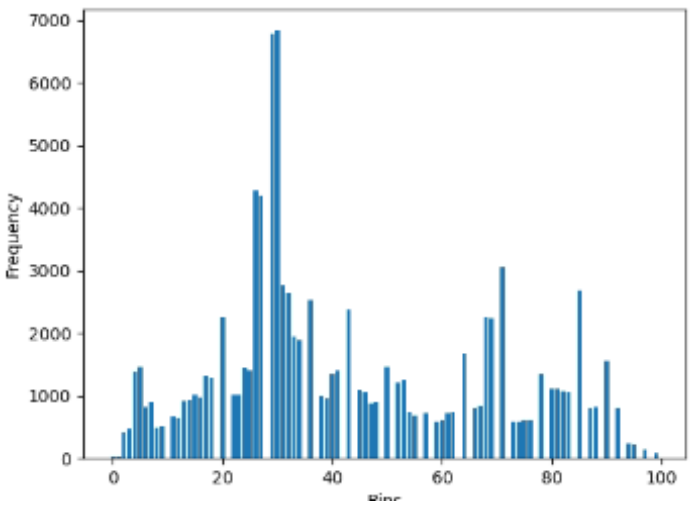
predicted:



Histogram of predicted:

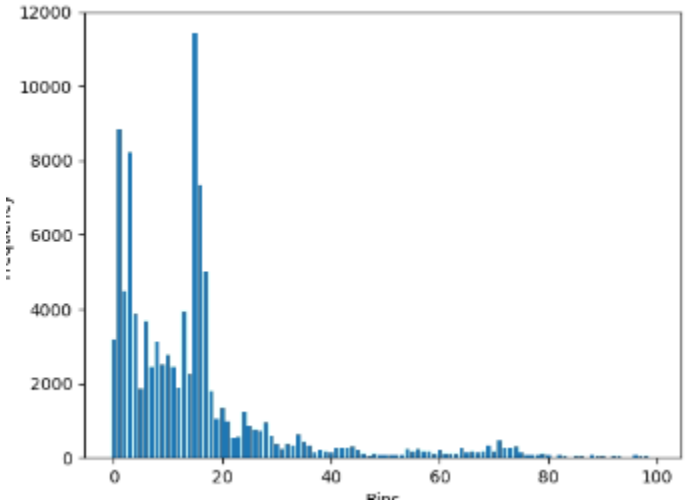
Histogram of groundtruth:

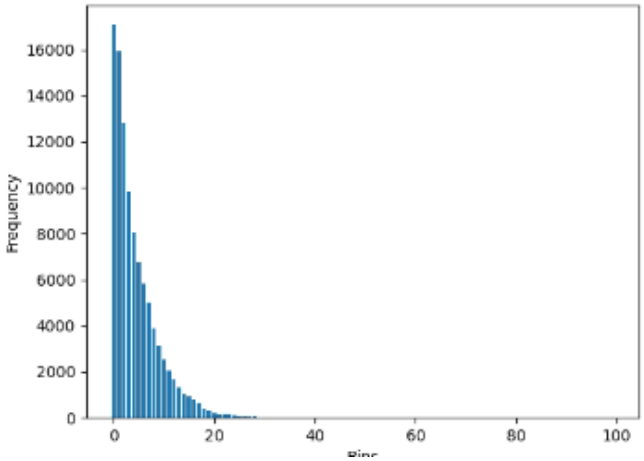




Errors:

Disparity to depth (labels) errors: *Predicted disparity to label disparity (prediction) errors*

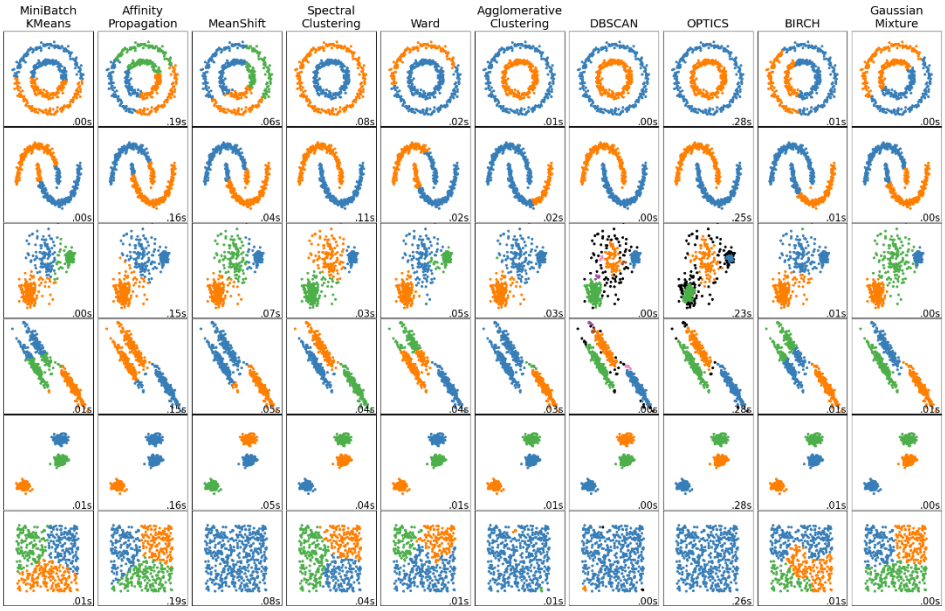


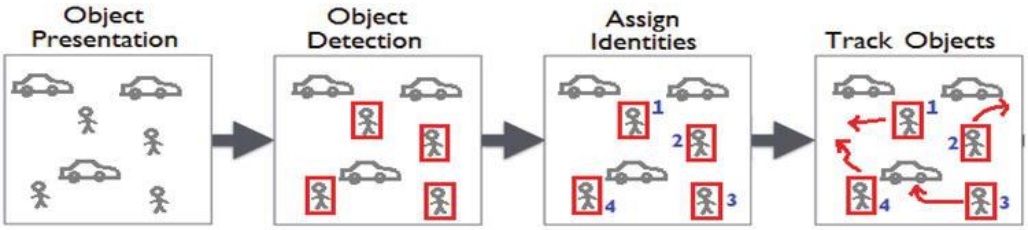


As a proof of concept we saw that for our setup such an approach can achieve results with pretty good estimation errors. The task of direct depth learning seem to be easier then learning the disparity and then transforming it to depth as there are more errors in this 2 steps then in a direct depth learning. Some spreading effect still have to be improved. More data, deeper network, longer training as much as additional regularization / loss constraints on the inner distribution (i.e Frechet inception distance) could be helpful for this task too. This approach has potential but more research should be done.

- An interesting notion is also the fact that the network seem to be good for hole filling, which can be used in the future in a way where we retrieve left and right Real Sense images.

Pointcloud Segmentation & Tracking – Horizontal view





In this part we use few clustering algorithms on different views of the data as object segmentation & detection method, then calculate the centroids and boxes for them, then create for each centroid a Kalman Filter for tracking it. The identity assignment of the objects/clusters will be done through solving a matching problem, where in each step we compare the predictions of the Kalman Filter and the current centroids of the new clustered and found clusters/objects. The cost is defined as the Euclidean distance in 2D, i.e in the image, rather then 3D. This is a choice we made on purpose due to the fact that it is easy to see the movement in the 2D plane. Any movement which can be identified int the 3D is also identified in 2D and in 2D even far objects which can in 3D move a big distance, but in 2D move a little bit, will be also covered by the cost we define for close-range objects movements. It is also easy for visualization without projecting 3D boxes and scene to 2D, although it can same easily be

shown in 3D. For these advantages, although we detect the objects incorporating the 3D data, which is excellent for the task of object segmentation/detection, we solve the matching problem for the centroids of the clusters in 2D, i.e., we know for each 3D point where it is on the image and after creating the cluster using its 3D representation, we find the 2D centroid of it on the image. It can be thought of as a projection on the 2D plane. Same is done in both following approaches, the clustering approach in the horizontal view and the foreground/background detection approach in the angled vertical view.

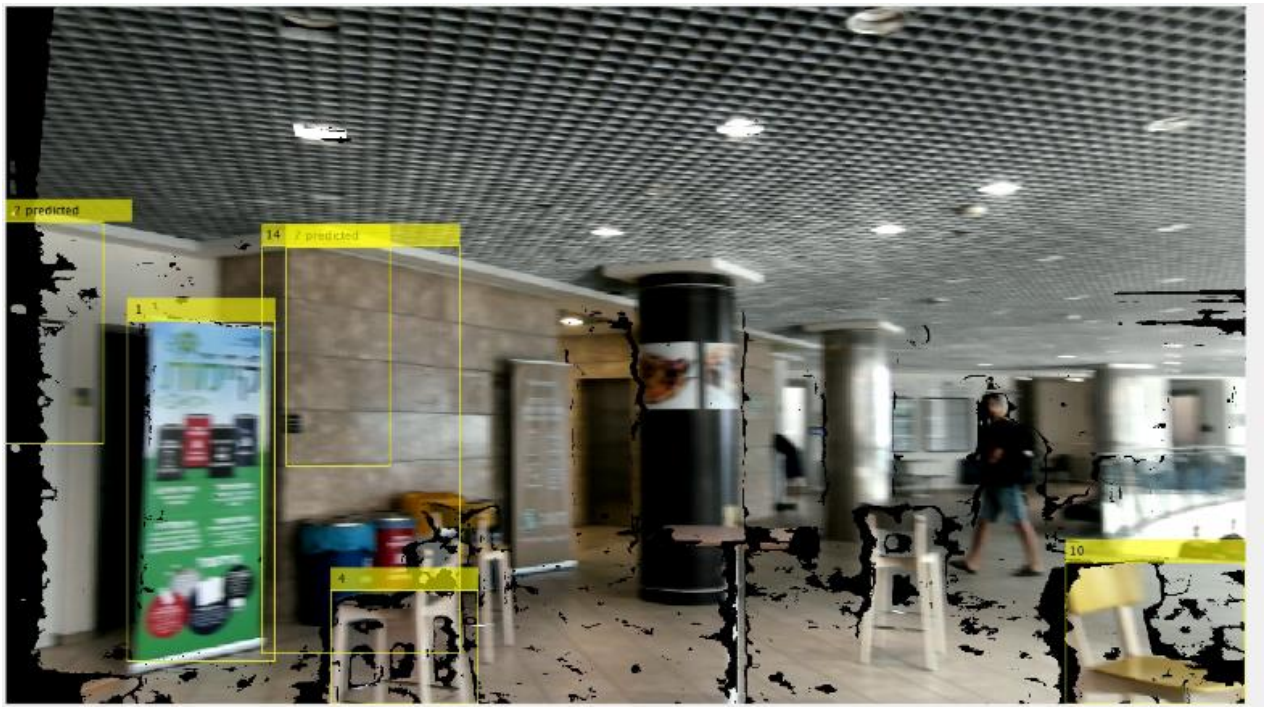
We show our results here on the Real Sense data: RGB images and depth data, but the methods are compatible also with the 3D thermal data the creation of which we achieved in the previous parts.

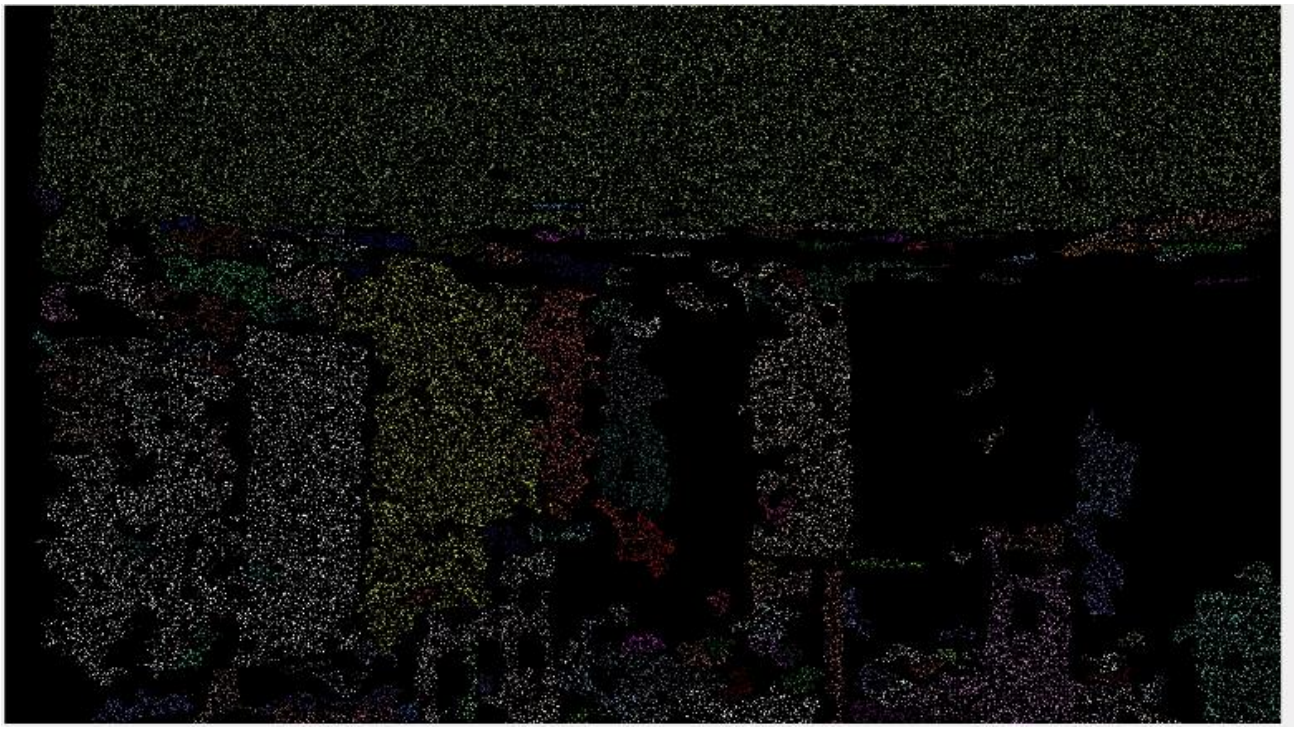
In this part we tried many clustering algorithms such as Hierarchical Clustering, K-Means, Gaussian Mixture Models and Spectral Clustering. Most of them are computationally expensive for 900K points except k-means, so we down-sampled the PointCloud to 0.1 of its original size with somewhat preservation of the structure and the PointCloud properties.

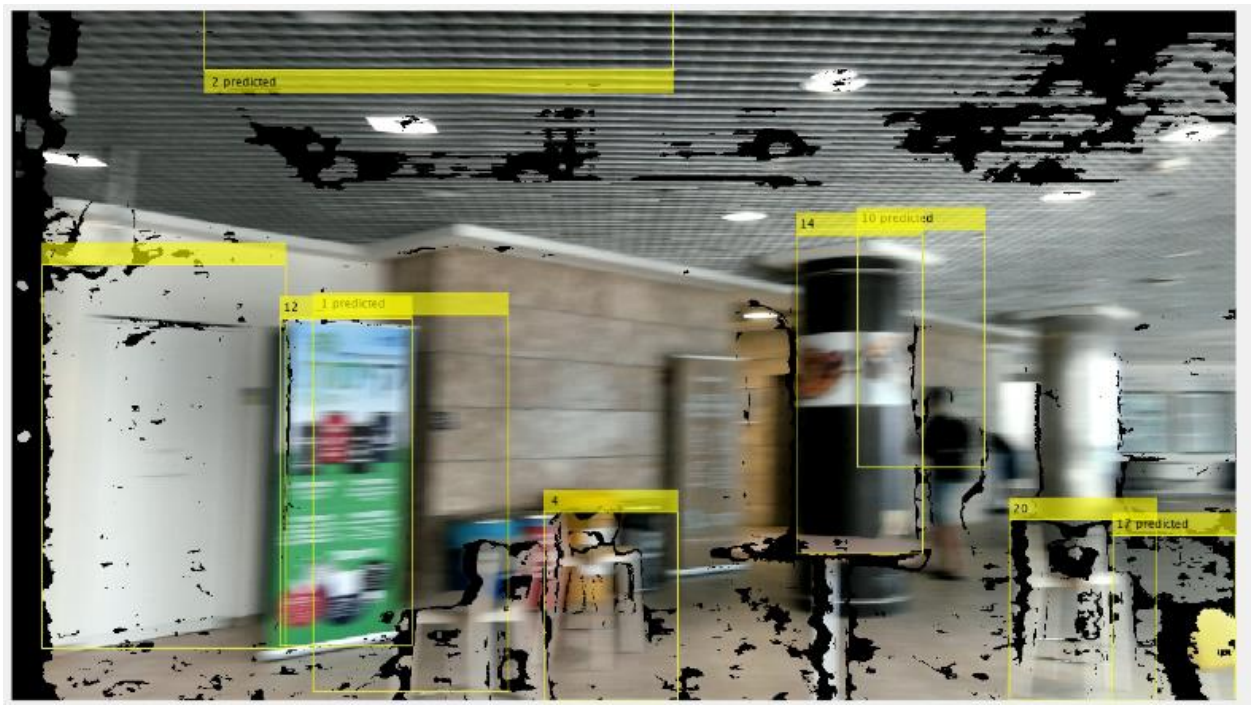
We present here only the results of the DBSCAN clustering method, as it showed interesting results for the object clustering/ detection and tracking.

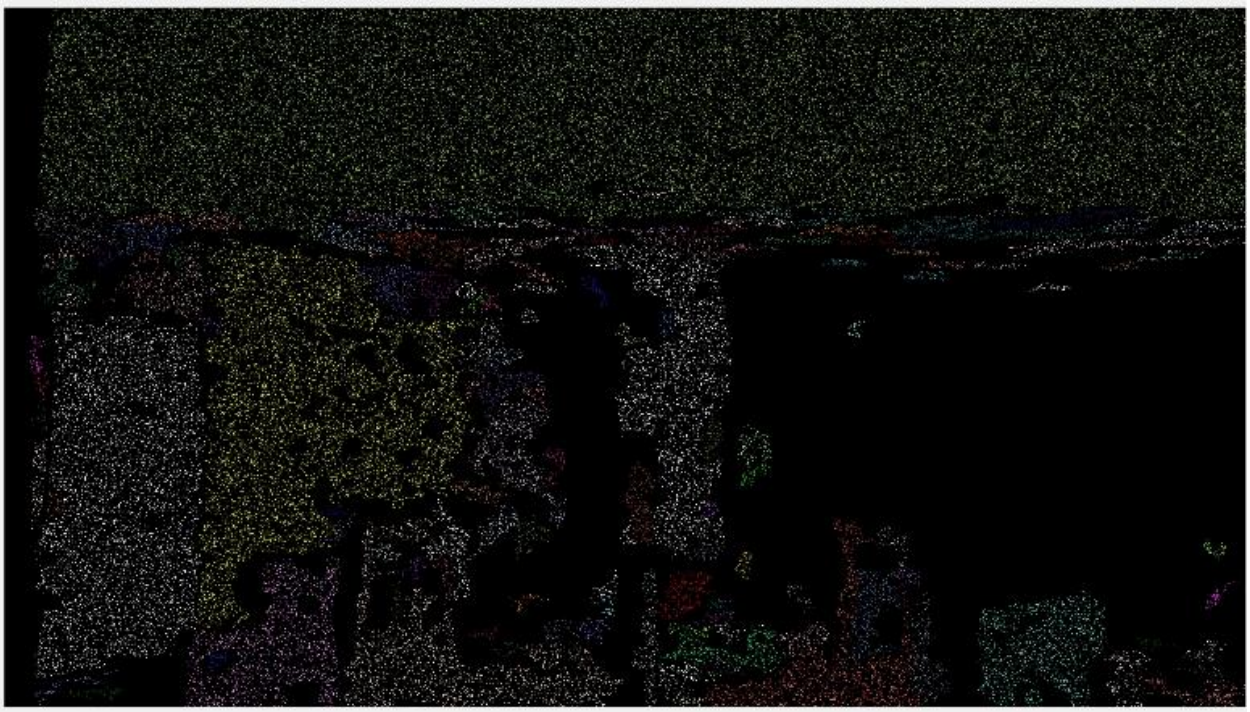
The next 9 pages of illustrations show a continuous flowing scene of the movement of the setup with moving objects around and the clustering of the down-sampled PointClouds and tracking in the approach was described above.

You can notice that as DBSCAN is a Density-Based Spatial Clustering method, it is very sensitive to the density of the points which is drastically changes within one scene for different distances, an effect which is created by the properties of the Real Sense Camera. Therefore, it can perform good for one kind of scenes and different for another, for which it should be adjusted with its epsilon hyper-parameter. As a result of this, the smallest values for the epsilon parameter give better results in clustering close-range objects, because the density of the points closer to the camera is bigger (the far points are clustered as noise) and bigger values are better for far-range points (the close-range clusters because of their density are united to big clusters, many times many objects in one cluster without good differentiation), which are also less dense and more spread. This characteristic shows that eventually more robust density based clustering algorithm is needed, which will be able to consider many density distributions in one scene and to cluster each one of them with the appropriate radius of search, which is he epsilon parameter. This effect can be seen especially on page 41.

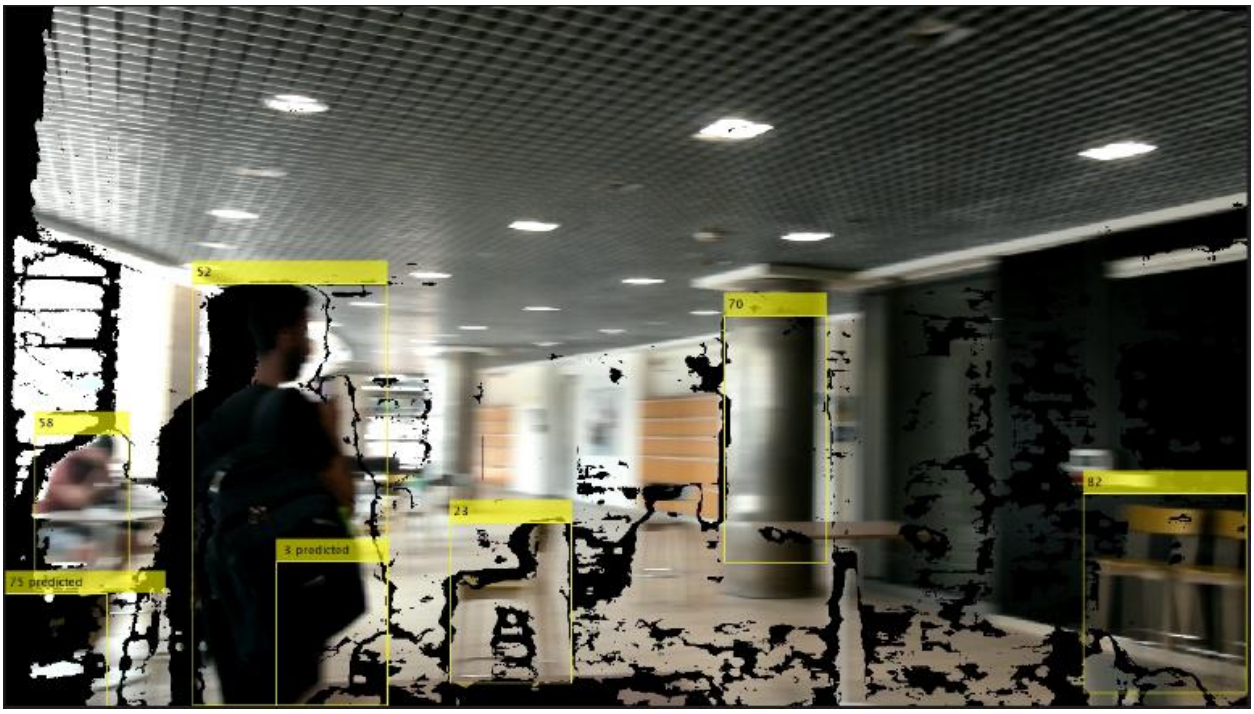

7 predicted
14 7 predicted
1
4
10

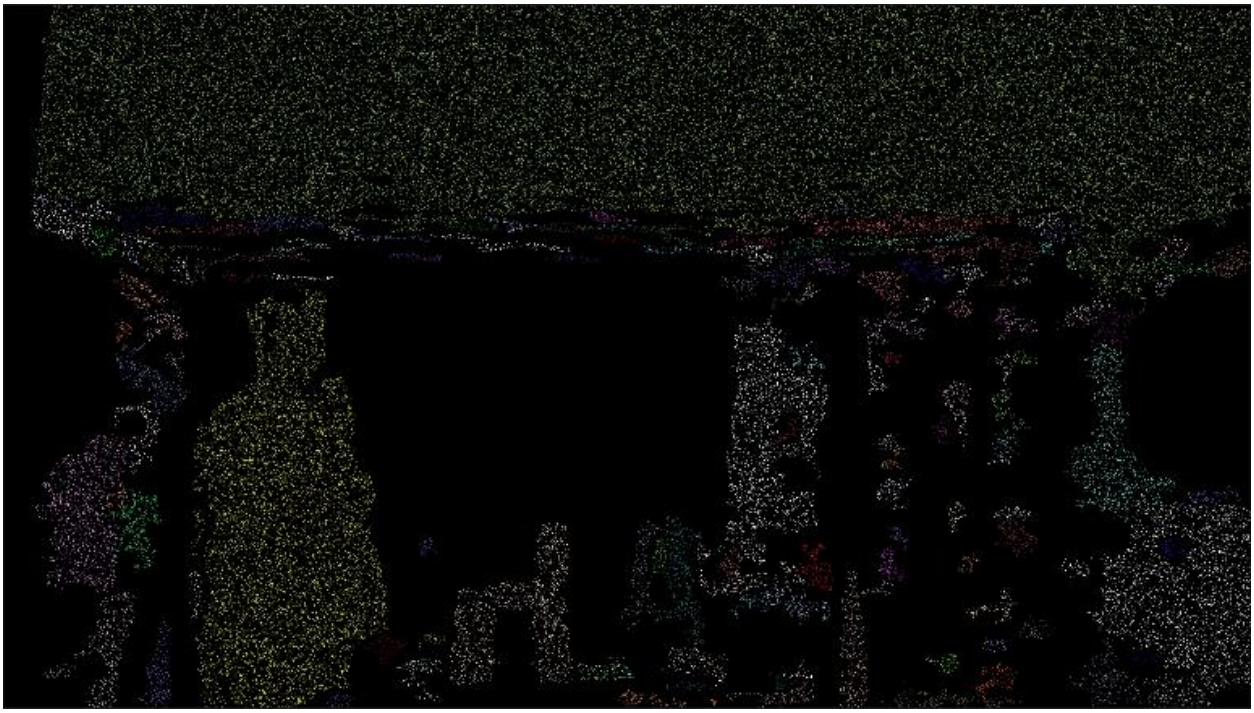



2 predicted
14
10 predicted
12
1 predicted
4
20
17 predicted

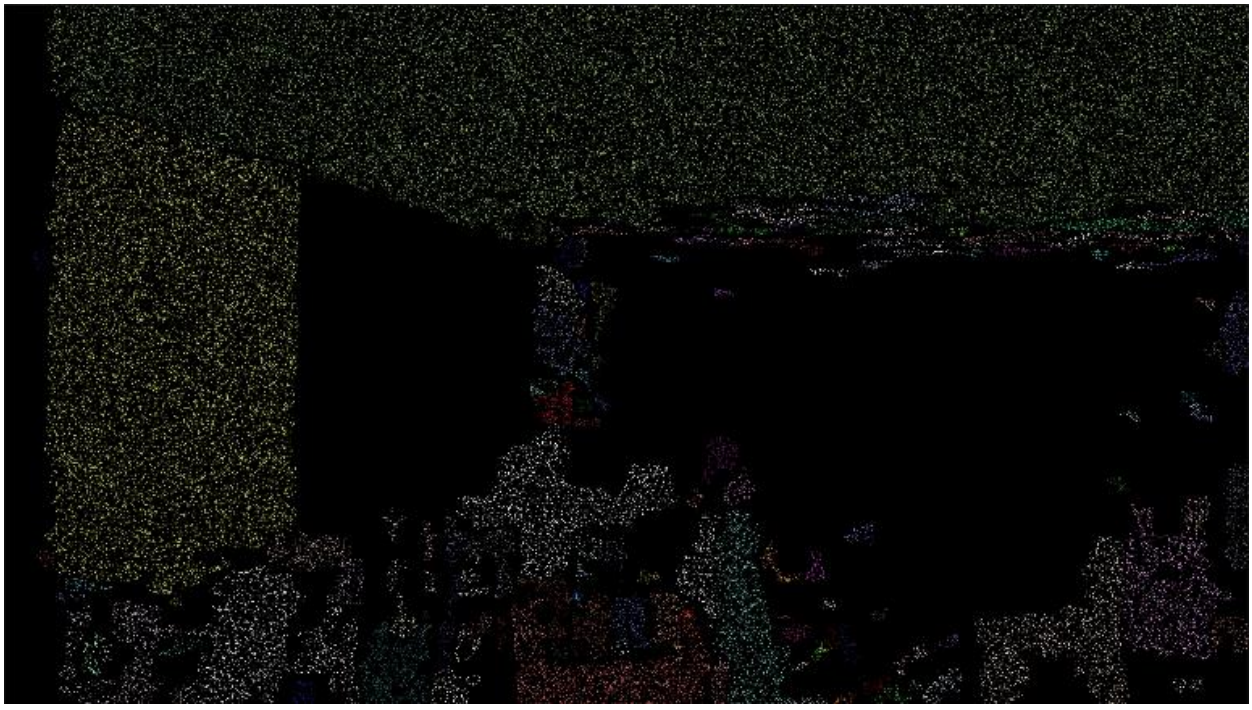


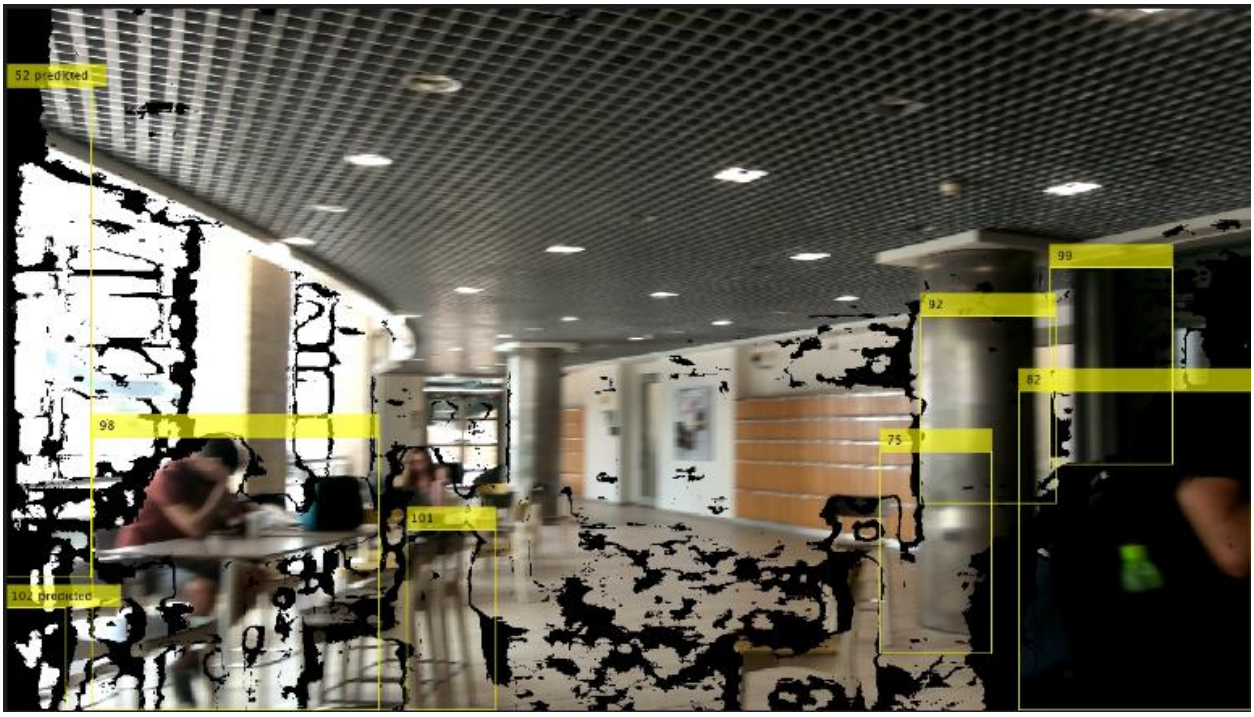

1
14
6
44
9
3 predicted
36
23 predicted


52
70
58
82
23
8 predicted
75 predicted

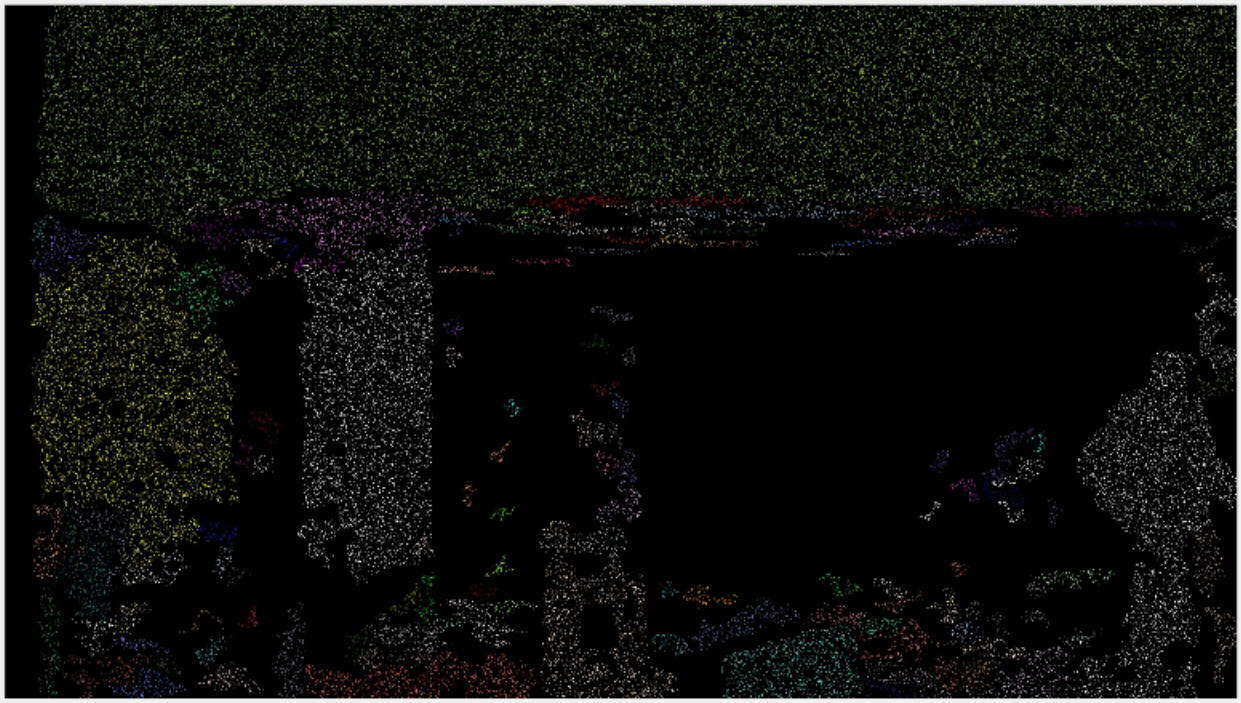



82 predicted
98
75
70 predicted
99

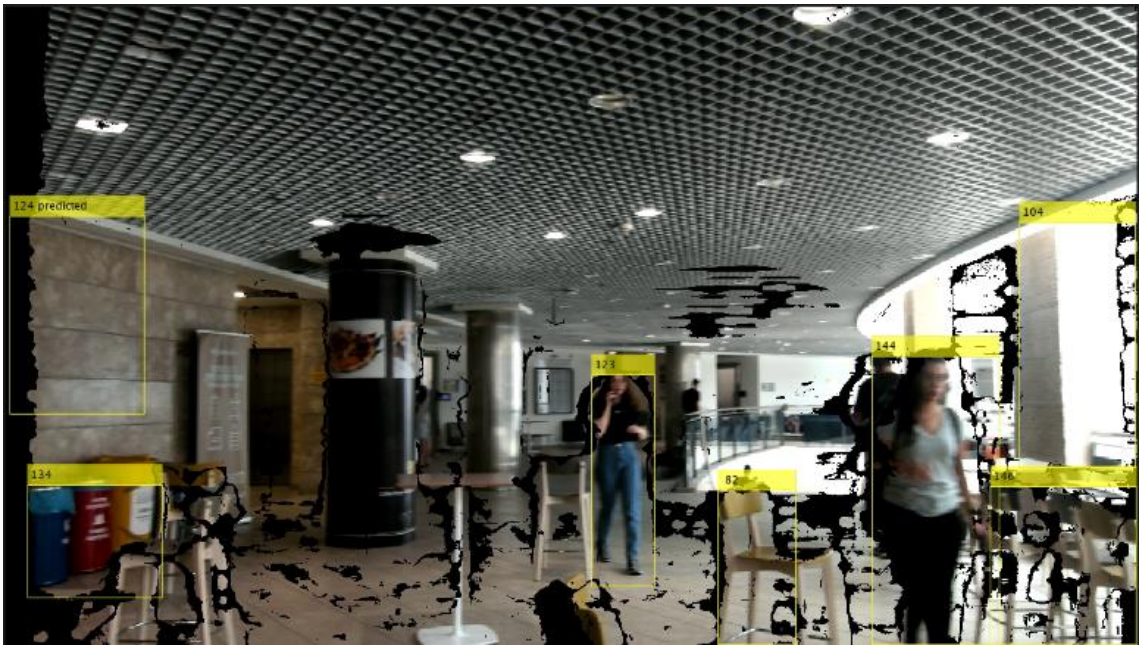


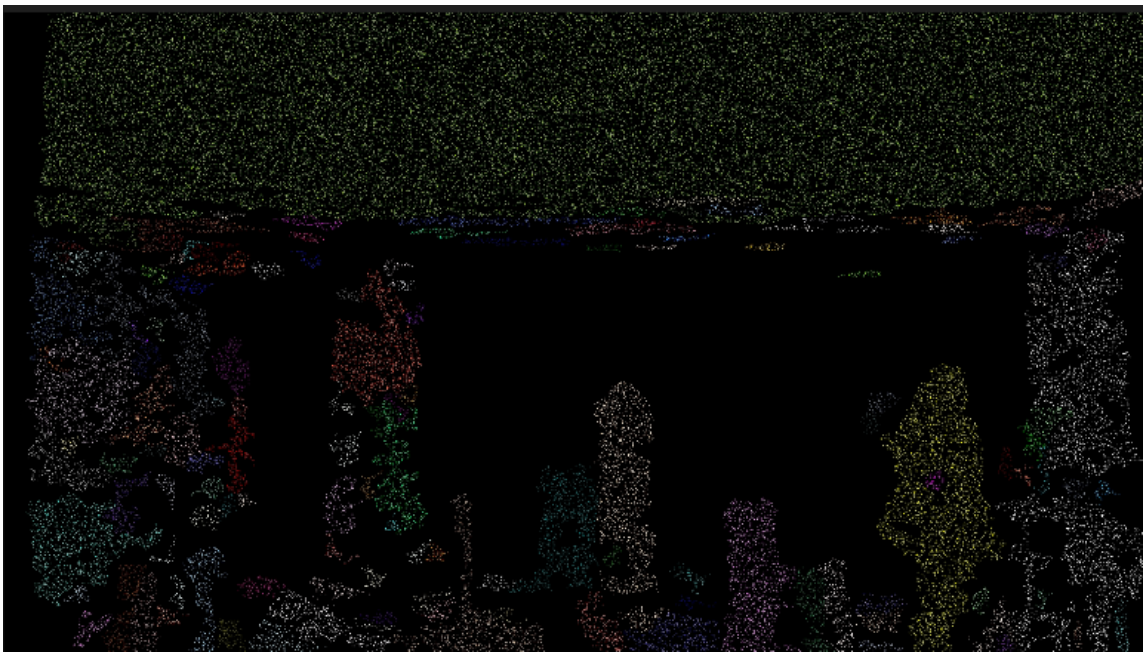

52 predicted
99
92
82
98
75
101
102 predicted


1
10 predicted
7 predicted
17 predicted
4
30
28 predicted
12




82 predicted
104
52


124 predicted
104
144
123
134
82



Pointcloud Segmentation & Tracking – Angled view

Foreground detector & tracking:

In this approach instead of clustering objects we take advantage of the angled view which simulating the recording setup being on a high angled position above the boat located on, for example.

In that approach we use Gaussian Mixtures Models to differentiate the background from the foreground, for which the angled view creates a very good conditions. Rather than it's a sea scene or a grounded scene, there will be always background on which objects are located from such angle. In this algorithm the number of gaussians for background detection is controllable, creating the option to control how complicate we suppose our background is. Everything is surrounded by background is a foreground blob which is potentially an object to track.

The background/foreground detector creates a binary mask after fitting the gaussians to the foreground as can be seen in the following pages. Then for each blob assigned a Kalman Filter as was one for the clusters before and the matching problem is solved for identity assignment as described in the general flow.

Here we use the depth and the RGB color in the following way:

On both we use the background/foreground detector separately and create 2 binary masks. We can see below that they can be noisy, especially the depth data. Then we do a logical-and operation and in that way we filter the noisy blob as much as emphasize the objects which are found both on color and depth data, i.e if for example on the color image we will see a reflection on the water of an airplane above us and the reflection could be seen as an actual object on the water surface (same with shadow of a high placed object like a bird and etc), the depth mask will clear. In this approach we don't actually use the x,y 3D coordinates of the data, but the RGB and depth data in a complementary and auxiliary way.

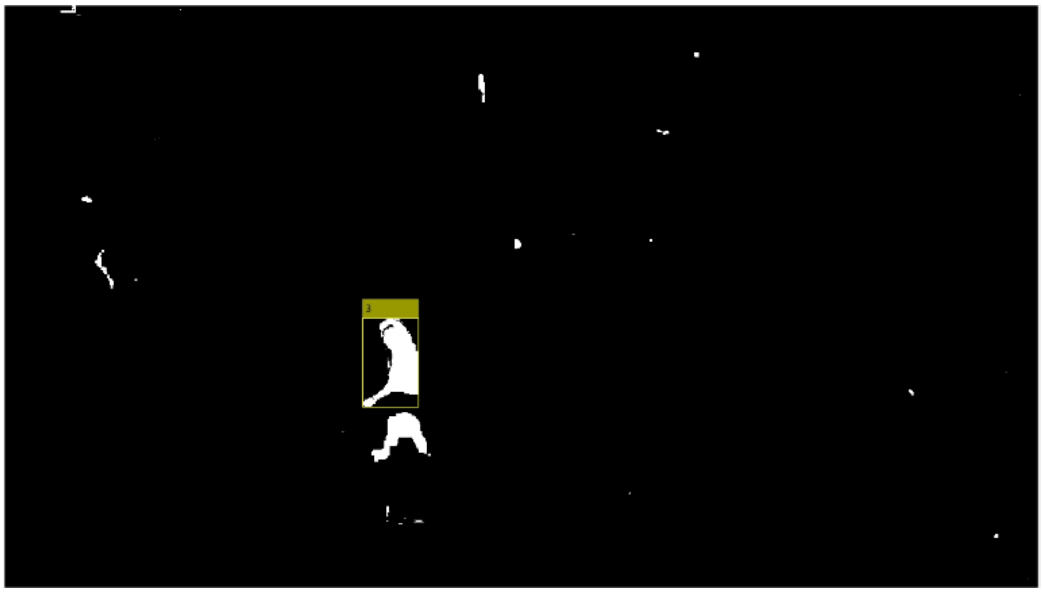
In this way some noisy blobs still survive (False Positives) but the major if not all True Positives are detected and tracked (which you can see also on the number labels they get through tracking).

Technical Note: for the depth data, the close range object to be foreground in view from above, we take the maximum value and reverse all the values by subtracting from it all the depth values (if we think of the depth values range o to 2^16 as a gray scale range (div 256) then the far objects has values closer to white and closer objects are have lowers values i.e closer to black and the foreground detector performs better when it's the opposite for experimental reasons).

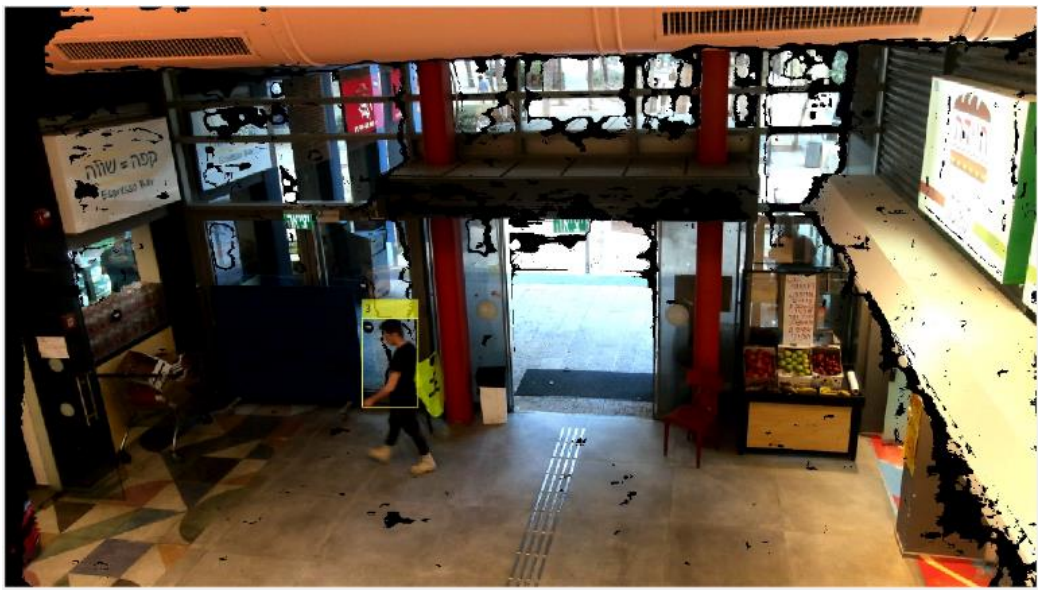
Note: in this approach to compute which objects are too close and should be alarmed about, we have to do some simple geometric computations (we know the angle we choose to locate the recording setup and the height, thus we an compute the actual horizontal distance up to some depth noise error).

Foreground/background detection on color data:

Mask:

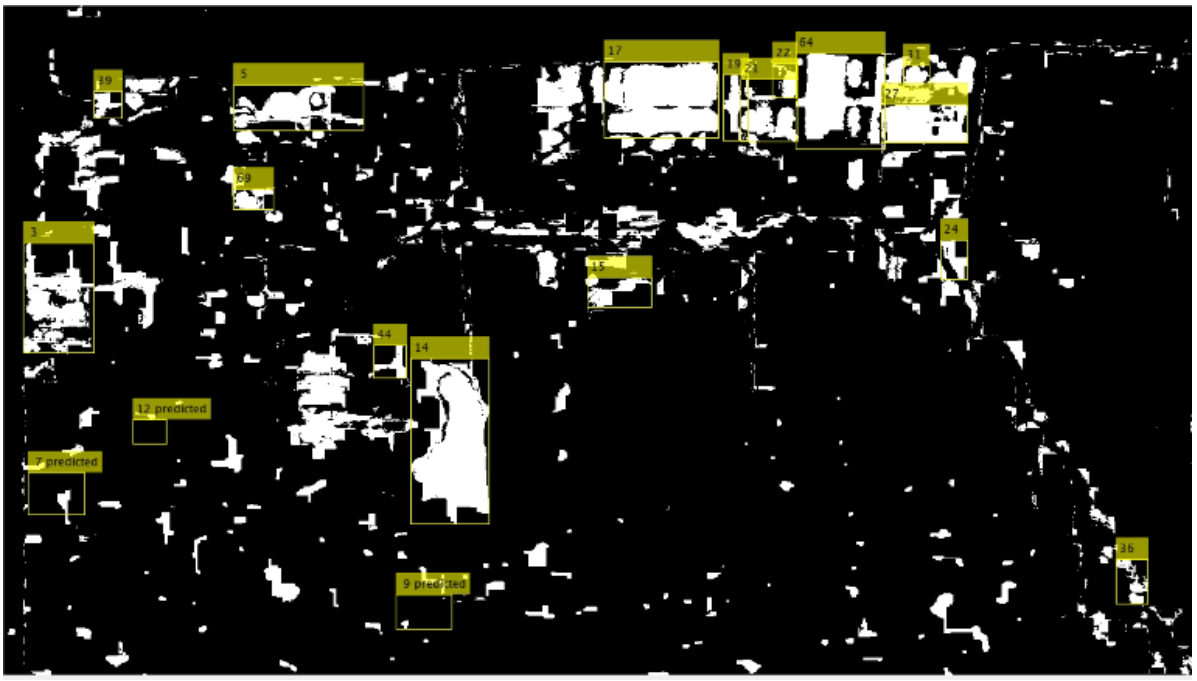


Tracking:



Foreground/background detection on depth data (illustrated on the color image):

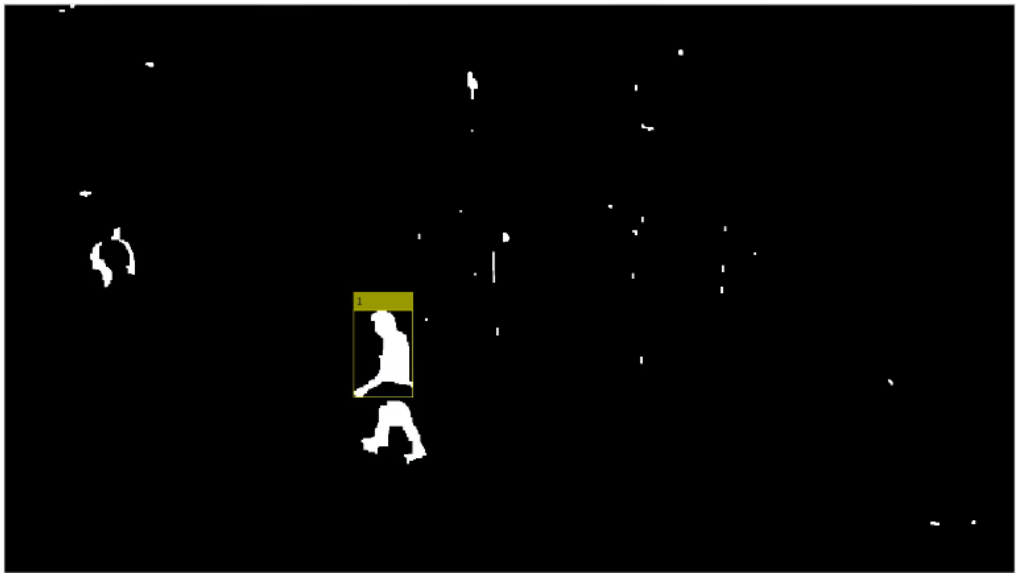
Mask:



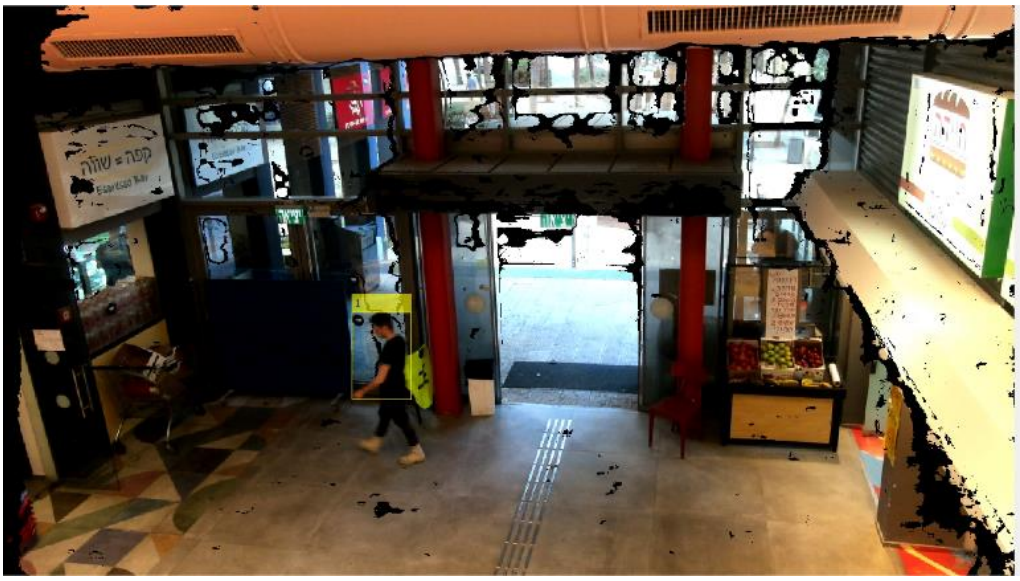
Tracking:

Intersection of the masks:

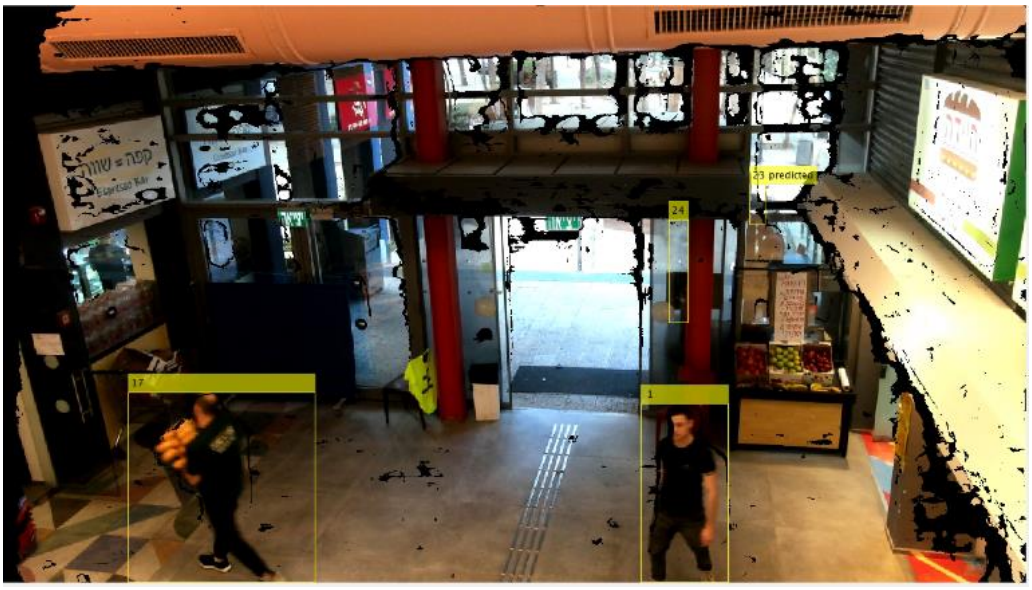
Mask:

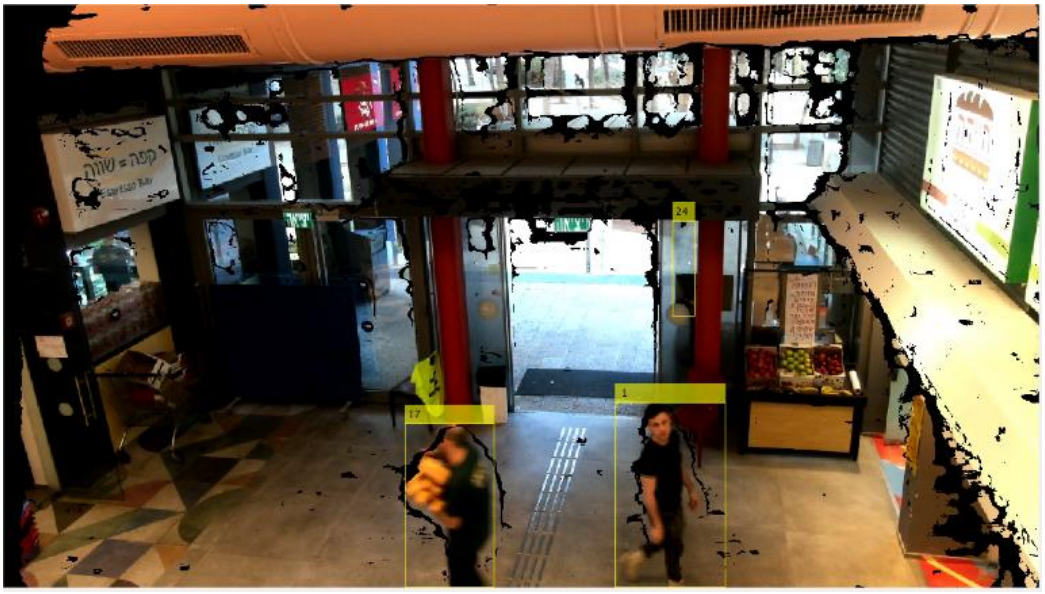


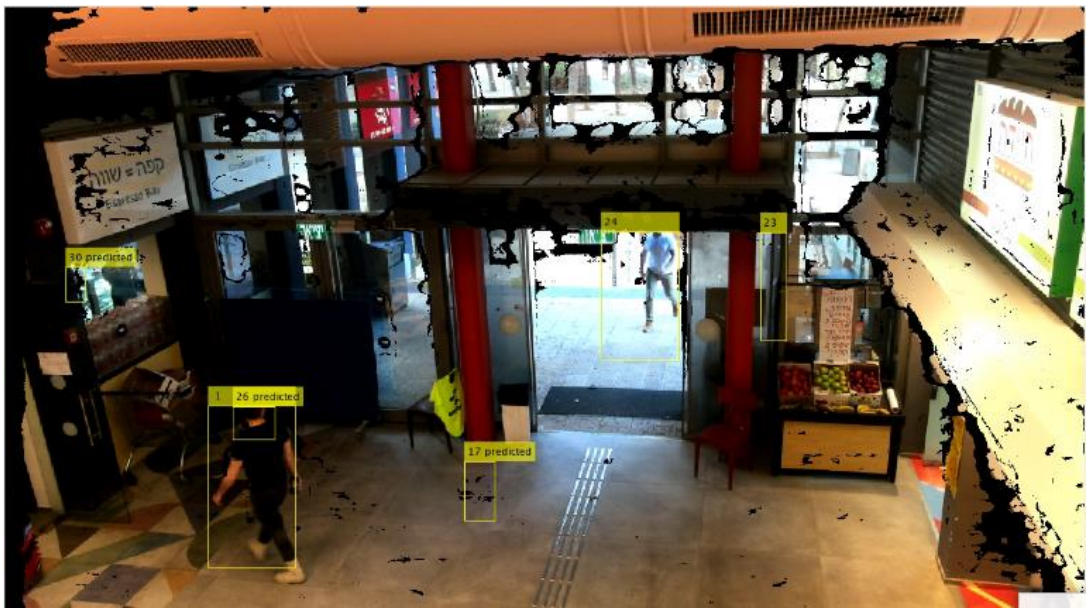
Tracking:

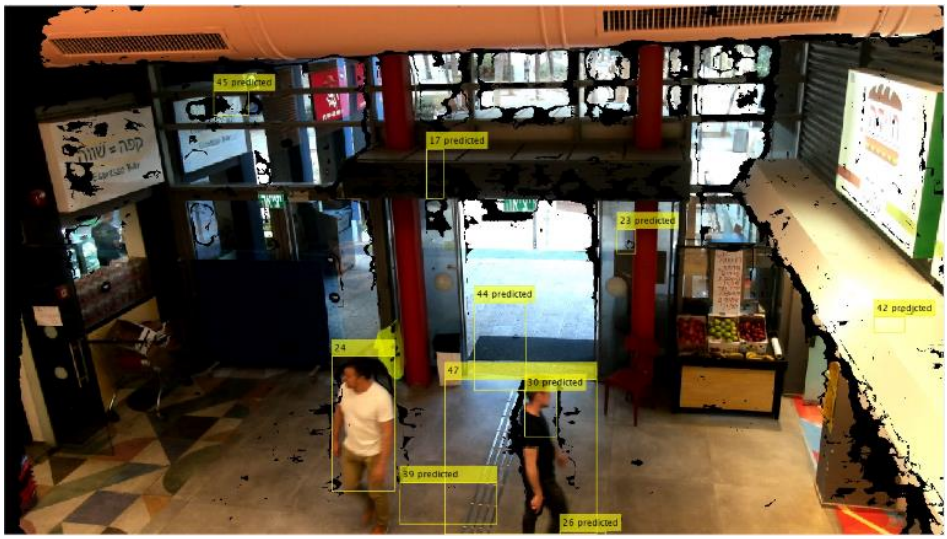


Here are many more examples of that scene, tracking the objects with the intersected color & depth mask (only images are shown as you saw the masks above):

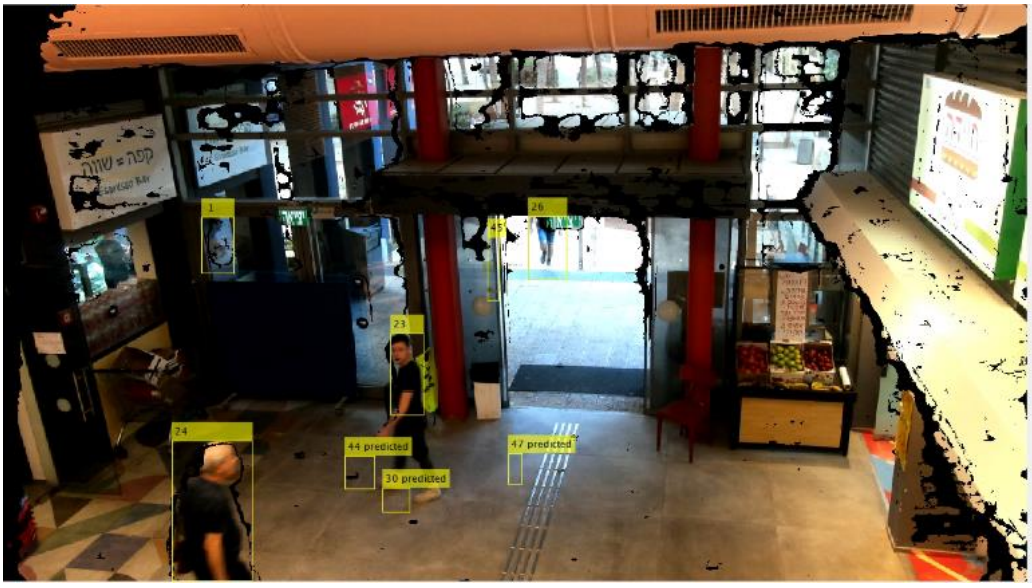

23 predicted
24
17
1

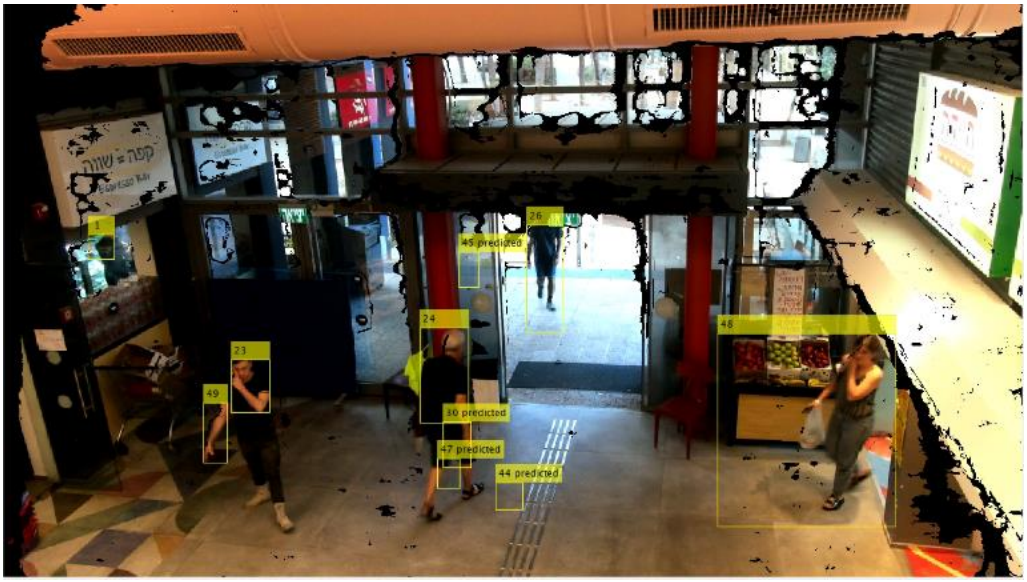

24
17
1


30 predicted
24
23
1
26 predicted
17 predicted


45 predicted
17 predicted
23 predicted
44 predicted
42 predicted
24
47
30 predicted
39 predicted
26 predicted


24
23
17
1
30 predicted
26 predicted


1
26
23
74
44 predicted
47 predicted
30 predicted



Note:

In both methods we showed only the detection and tracking of the objects, but of coarse it is very simple to show that for some threshold on the objects distance (in the first method directly on the average distance of the blob, which can be averaged on the cluster points or the box we have for it, and in the second same as in the first but with the geometric formula was mentioned), we can change the color of the too close objects to red, as a notification for potential accident. Also using the data of the Kalman Filter we can easily estimate time to collision by accumulating distance estimations (i.e velocity vector) and linearly predicting it, modeling time to collision with a constant velocity and direction, in the spirit of objective 2.