Distance and Movement Measurement of an Object based on Stereo Images

Lennard Rose, 5118054, FHWS Moritz Zeitler, 5118094, FHWS

Abstract

This paper explains how to compute distance to, and movement of an object with the means of stereo camera vision. Included is the whole workflow consisting of all steps that have to be made to get such a recognition to work. These steps are the hardware setup, all calibration steps, object detection, object tracking and the measurements of the values themselves.

I Introduction

The human eye enables us to perceive our surroundings. Alongside color and shape, it enables us to determine the position in relation to us and to each other. This is possible because a human being has two different point-of-views, the left and right eye. In this project, the so called Stereo-Vision will be applied with the means of a modern computer. To do this, a stereo camera setup is used to take photos/videos, create a depth map and try to compute the distance and movement of an object. The following chapters will describe this in a step by step fashion, starting with the basic theory behind it and the implementation afterwards.

II Prerequisites

The initial idea was to run this Stereo-Vision on some kind of self driving vehicle, in our case a JetBot AI Kit[1]. This JetBot AI Kit is based on a NVIDIA Jetson Nano Developer Kit[2]. Mounted on this Bot is a the IMX219-83 Stereo Camera[3] of Waveshare. After assessment of the provided resources and some in depth testing, the decision was made to not use the bot for running our applications, since it would impede development tremendously. This means all data used in the project was recorded using the bot, but all computation and programming was done on other machines. This means that the resulting pipeline should be deployable to the bot but there are some severe driver problems that stop this from happening.

III Stereoscopy

To visualize the idea behind stereoscopy, the human eye can be used as an intuitive example. Focusing on an object and alternately closing the eyes, the object will appear to be jumping right to left and vice versa. The closer the object, the farther the object will jump. With this information from the different viewpoints of our eyes, the brain is capable of perceiving depth. This is the whole concept behind stereoscopy simplified. Without this, it would be hard to impossible to guess the position of neighboring objects with the same appearance or do something simple as catching a ball out of the air. When capturing a 3D object in an image, the objects gets projected from a 3D space to a 2D (planar) projection space. This is called the Planar Projection, the objects are losing all their depth information. The depth information can only be guessed by shadows our overlapping features. The question now is, how to get back the depth information from this 2D space, without the means of experience. With 2 cameras, this is possible by comparing the left and the right image, called triangulation. Finding an object on these two images can be done by using epipolar geometry and stereoscopy. This is basically to reduce the searchspace and not having to compare every pixel.

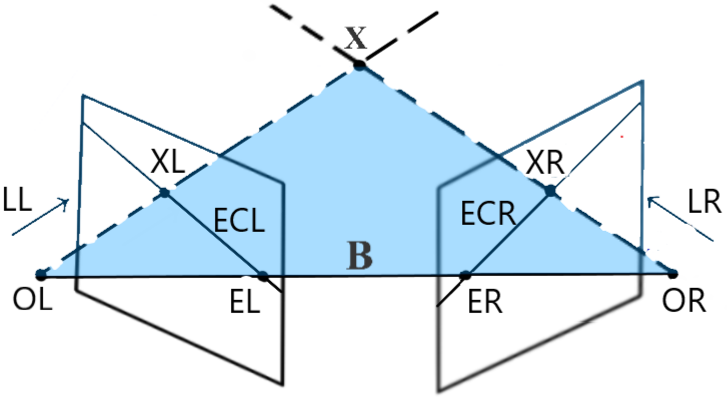


Fig. 1: Epipolar geometry

Following the above figure this concept works by projecting the line **LL** between the points **OL**(left camera), **XL**(projection of point **X** in the left frame) and **X**(being the target 3D point) as **ECR** on our right picture, crossing the point **XR**(projection of point **X** in the right frame). The same on the left picture, project the line **LR** between the points **OR**(right camera), **XR** and **X**, crossing the point **XL** as **ECL**. The centers of both cameras get also project to there significant other frame. With this, the searchspace got reduced of a corresponding point to compute our disparity from to a single line, **ECL** respectively **ECR** (this is called an epipolar constraint). **EL** and **ER** is the projection of one camera onto its significant other cameras pov. These points are called epipoles. Together with **LL** and **LR** this baseline forms the epipolar plane (blue in figure 5). All this information can be represented by the Fundamental matrix of the image, which can be calculated with the projected points and the projection matrices of the cameras. Further mathematical description can be found in the book *Multiple View Geometry in Computer Vision*[4], but won't be discussed any further at this point.

IV Workflow

This chapter contains a detailed theoretical and practical description of each individual step in the workflow.

IV.A Stereo Camera Calibration

Cameras are projecting a three dimensional object to a two dimensional image. This projection is never not only determined by the original object, but also by all parameters of the used camera. These you can separate in two groups, external and internal. External parameters are the camera orientation in different planes. Internal parameters are determined by the cameras internal characteristics.

$$\mathbf{K} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 2: Intrinsic Matrix

These internal parameters are represented as an upper triangular matrix K. This matrix has the values fx and fy as the x and y focal lengths. cx and cy as the x and y coordinates of the optical center in the image plane. Gamma is the distortion between the axes, this value is replaced by a ”0” in openCV´s calibration function.

$$\left[R \mid t\right] = \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{array}\right]$$

Fig. 3: Extrinsic Matrix

The external parameters are represented by an 3x3 rotation matrix, that indicates the rotation bias of the camera. Combined with an 3x1 translation vector, which describes the position of the world coordinates origin in the camera coordinate system.

$$\mathbf{P} = \overbrace{\mathbf{K}}^{\text{Intrinsic Matrix}} \times \overbrace{[\mathbf{R} \mid \mathbf{t}]}^{\text{Extrinsic Matrix}}$$

Fig. 4: Projection Matrix

Together these form the 3x4 Projection Matrix. This matrix and the real world coordinates of our point is used to calculate the image coordinates of the 2D projection. Calibration does nothing else than alter these matrices so our 2D projection matches the 3D object in the real world. This is usually done by giving the calibrating program/ function a set of 2D images with known 3D coordinates. This Calibration consists of two steps removing of Distortion as well as rectifying the images[5].

IV.A.1 Distortion

A camera lens always has some kind of curvature. This curvature is also visible in an image taken with this lens. This is called distortion. There are many different types of distortion which can be seen in the next figure.

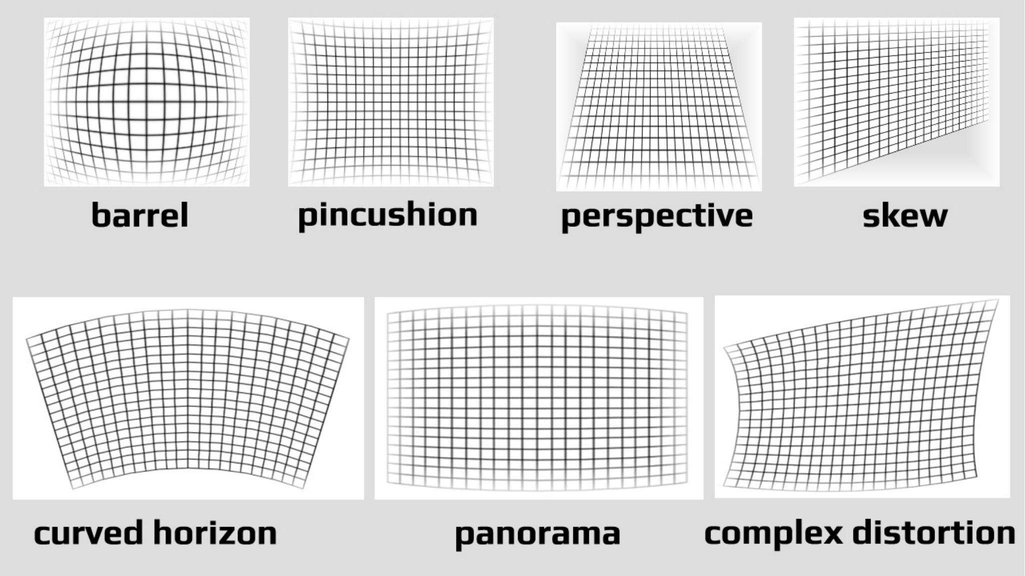


Fig. 5: Different types of distortion

To remove this distortion, pictures of a chessboard are taken. openCV, the package used to do all calibration steps, can easily detect key points using this chessboard pattern. This is based on the big difference in pixel values between white and black. In combination with the size of the squares as well as the amount of the squares, openCV is able to create a calibration matrix removing the distortion.

A very important part is to be very precise while taking the calibration pictures. To get a good result you need to take calibration photos from all parts and angles of the cameras perspective.

The python implementation looks like the following:

First of all, the input picture was converted to grayscale and passed to a detection function for chessboard corners.

```
ret, corners = cv2.findCirclesGrid(inputImage, (chessboardColumns, chessboardRows), None)
```

The inputs are the chessboard picture, columns of the chessboard, rows of the chessboard and an output array not further used. This function returns whether a chessboard got detected and if so the 2D points of the chessboard corners. If the detection was successful the returned points get even more refined using:

```
cv2.cornerSubPix(inputImage, corners, (11, 11), (-1, -1), criteria)
```

The ’cornerSubPix’ of openCV refines the corner locations [6] of the checkboard. All results of the previous steps are saved in a list, which caches them for future use. With these lists a single camera can be calibrated :

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(object_pts, img_pts, img.shape[::-1], None,
                                                                                        None)
```

Passing these points, as well as the shape of the image into the calibrate camera function, results in computation of the intrinsic and extrinsic parameters. These parameters are stored inside the *mtx* variable. Also the overall RMS re-projection error[4] in *ret* and the distortion coefficients are returned. *Rvecs*[7] and *tvecs* are rotation and translation vectors, which aren’t used in this system and won’t be described further. On a high level openCV executes the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points *imagePoints* and the projected (using the current estimates for camera parameters and the poses) object points *objectPoints*[8]. With these matrices a new optimal matrix based on these parameters can now be computed.

```
h, w = img.shape[:2]
new_mtx, roi = cv2.getOptimalNewCameraMatrix(mtx, dist, (w, h), 1, (w, h))
```

As a result, the function returns a new optimal intrinsic camera matrix. The function also returns a valid pixel region which outlines a region of all-good-pixels in the undistorted image. These matrices and results will be later used to undistort and rectify images coming from our camera for better results while computing the disparity and depth map.

IV.A.2 Rectification

With image rectification, the goal is to project the two different images taken by the camera onto a common image plane. This process ensure that the individual horizontal pixel-lines of both images are on the same level. This means the epipolarlines are calibrated to be exactly horizontal, which helps finding the correspondence of a point in both of the stereo pictures by reducing the searchspace to a horizontal line. For this step the previous calibration matrices for both of our cameras as well as pictures from both cameras are needed, containing the chessboard to compute the rectification. This results in the following matrices and vectors:

1) essential matrix
2) rotational matrix
3) translation vector

```
retS, new_mtxL, distL, new_mtxR, distR, Rot, Trns, Emat, Fmat = cv2.stereoCalibrate(obj_pts,
                                                                                img_ptsL,
                                                                                img_ptsR,
                                                                                mtxL, distL,
                                                                                mtxR, distR,
                                                                                imgR.shape[::-1],
                                                                                criteria_stereo,
                                                                                flags)
```

The function can also compute the full calibration for each of the cameras but isn’t recommended to do so due to the high dimensionality of the parameter space and noise in the input data[9]. Because of this the individual calibration is done beforehand and gets supplied as an input. These matrices allows computation of a coordinate representation of a 3D point in one of the cameras using the result matrices as well as one of the individual camera matrices. Similarly to *calibrateCamera*, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error. Now the goal is to rectify these cameras using stereoRectify:

```
rect_l, rect_r, proj_mat_l, proj_mat_r, Q, roiL, roiR = cv2.stereoRectify(new_mtxL,
                                                                                distL,
                                                                                new_mtxR,
                                                                                distR,
                                                                                imgL.shape[::-1],
                                                                                Rot, Trns,
                                                                                rectify_scale,
                                                                                (0, 0))
```

Once again all previous results are input for the function. The function results in two rectification transform matrices, which transfer points from unrectified camera’s coordinate system to points in the rectified camera’s coordinate system.

IV.B Disparity Map

Disparity is usually computed as a shift of an image feature when viewed in the right/left image. For example, a single point that appears at the x coordinate k (measured in pixels) in the left image may be present at the k coordinate k minus 3 in the right image. This would result in a disparity value of 3 in pixels. These disparity values are the base for the later computation of the depth map. This can be done using specific algorithms. In this case the Block Matching Algorithm is used.

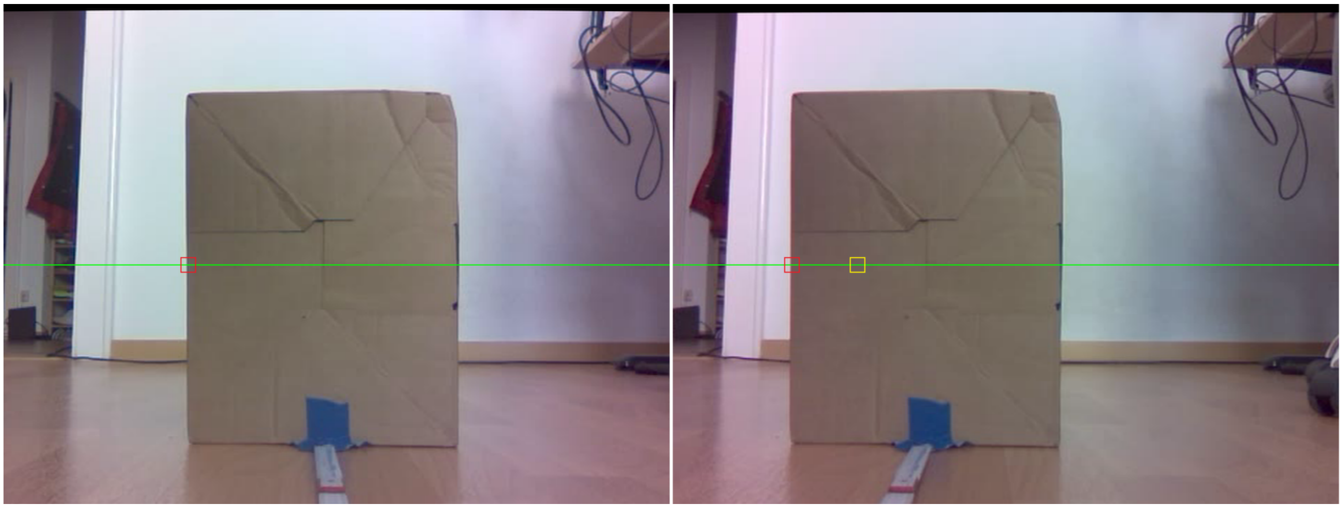
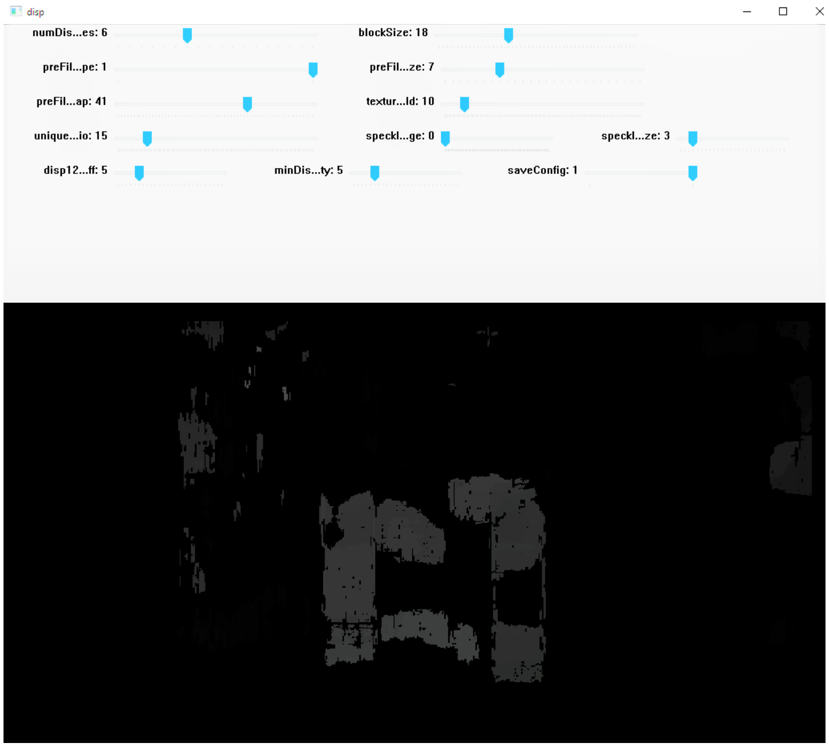


Fig. 6: Visualization of the Block Matching Algorithm working on input and output image

The above figure displays the method the algorithm implements. Input is one pixel, or in this case a small window(box) of pixels, which makes the algorithm more resilient. With the input, the algorithm searches for the most similar neighbor in the corresponding stereo picture. In this case the left image is the input, with the red square as the search window. The algorithm searches for its corresponding window on the right picture. The best match is visualized by the red window, the coordinates of the original box in yellow. The green line indicates the epipolar constraint. This can be done in with different metrics, for example absolute or normalized differences. The algorithm searches for the least differences to determine correspondence. However, the algorithm tends to find multiple pairs of corresponding points due to repeating textures or similar occurrences, in this case this would be most of the boxes brown surface, the wall or the floor. In Python the algorithm is implemented as a class which needs to be instantiated. To be more precise, the implementation used in this project is slightly more advanced[10] but follows the same principle.

```
stereo = cv2.StereoSGBM_create(minDisparity=min_disp,
                               numDisparities=num_disp,
                               blockSize=window_size,
                               uniquenessRatio=10,
                               speckleWindowSize=100,
                               speckleRange=32,
                               disp12MaxDiff=5,
                               P1=8*3*window_size**2,
                               P2=32*3*window_size**2)
```

Important are the different options supplied to the SGBM class. The block size parameter supplies the window size for the algorithm. The disparity amount, which has a minimum and a maximum, specifies at which range a disparity match is acceptable. Speckle and P1 and P2 are filtering algorithms to smooth the output[11], again further explanation will be skipped. To improve those parameters, an user interface was build to change the values on the fly.



numDis...es: 6
blockSize: 18
preFil...pe: 1
preFil...ze: 7
preFil...ap: 41
textur...ld: 10
unique...io: 15
speckl...ge: 0
speckl...ze: 3
disp12...ff: 5
minDis...ty: 5
saveConfig: 1

Fig. 7: UI for calibrating all parameters of the Block Matching Algorithm

Within the UI there are many different parameters to tune. Also there is the option to save the parameters. It isn't currently possible to use a button in openCV thus there is a slider which produces a *config.json*. The result can look something like Figure 8.

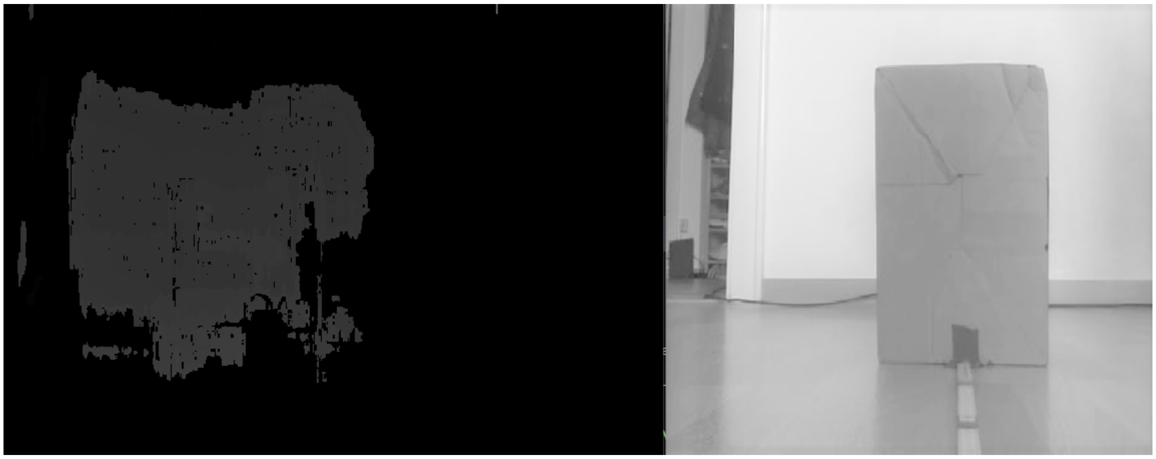


Fig. 8: Resulting Disparity Map after calibration without filtering on the left, grayscale left cameras image on the right

IV.C Depth Map

At this point there are two different options to compute a depth value from the previously created disparity map. The first one being the usage of the openCV function *reprojectImageTo3D*, which takes the disparity map as well as the Q matrix computed previously during the rectification process to compute a depth map. Alternatively computation of the distance with the inverse disparity-value is possible. The relation between depth and disparity is non-linear, but by using the inverse this relation becomes linear.

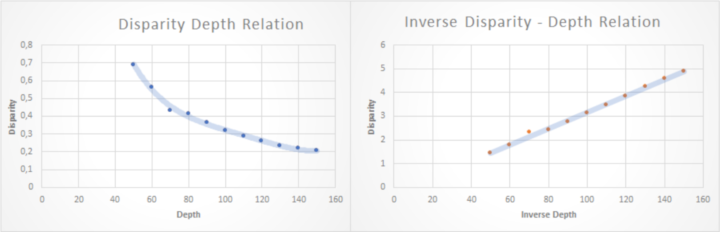


Fig. 9: Relation between Depth and Disparity(left) and Inverse Disparity(right)

Done by the following equation:

$$Distance = \frac{M}{Disparity} \tag{1}$$

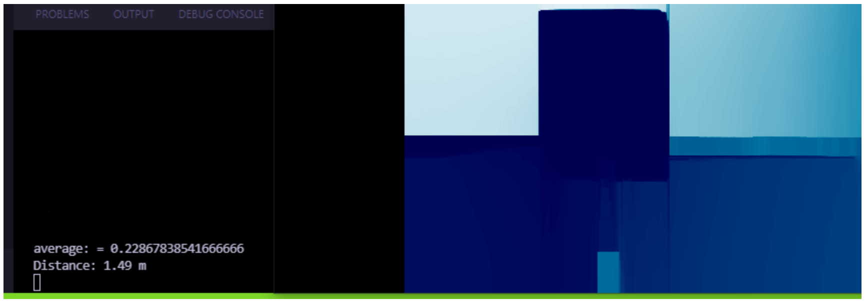
With **M** being the product of the focal length **f** and the baseline **B** of the camera.
In this practical scenario, the assumption was made that some of these values may have a small error. This might be because the disparity values are not as exact as expected, or because the camera has some internal calibration or construction issues. Similarly to the UI used to find optimal parameters for the Blockmatching Algorithm in figure 7, the **M** parameter gets sort of calibrated as well.

IV.C.1 Measurement

For testing purposes the initial idea was an event based system. On double-click, an openCV event gets triggered containing an x and y position of the mouse click. These values get enhanced to a window, all disparity values in this window get added up to compute an average. With this basic function computing the distance can be encapsulated. This results in following precision:

actual distance	computed distance
0.5	0.5
0.5	0.5
0.6	0.61
0.6	0.6
1.2	1.18
1.2	1.18
1.3	1.28
1.3	1.31
1.4	1.44
1.4	1.42

The mean squared error of these observations is 0.00034. The closer the value to 0 the better the result, meaning that the computed distance is extremely good. But this needs to be put into perspective, this test was performed under laboratory conditions.



```
average: = 0.22867838541666666
Distance: 1.49 m
```

Fig. 10: Depth map calculated with blue color filter and a computed distance based on a regression function

IV.D Object Detection

For object detection the first idea was to track large appearances on the disparity map sorted by their calculated depth, beginning with the nearest. Due to the disparity map being an inaccurate and error-prone projection of the real picture, this algorithm ended up detecting only unusable patterns:

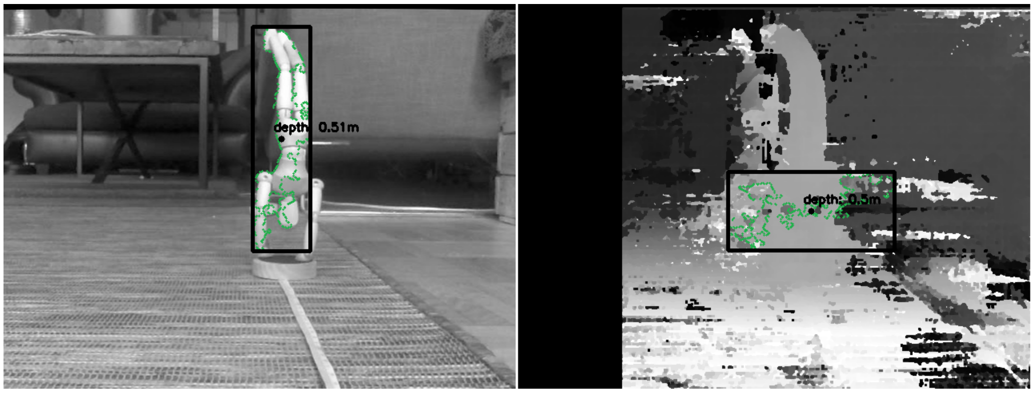


Fig. 11: Left: Detection on original picture Right: Detection on disparity picture

To solve this problem the original grayscale picture of one of the cameras was used to detect the objects. This was done by creating a mask that subtracts the unmoving areas of the picture, leaving only the targeted object. This was done by using the following openCV function:

```
back_sub = cv2.createBackgroundSubtractorMOG2(history=500, varThreshold=25, detectShadows=False)
```

The *createBackgroundSubtractorMOG2* function works by comparing a history of frames and declaring its unchanged pixels as background. The result is a mask only containing the potential objects that should be tracked. On this, some filtering is performed:

```
# Close dark gaps in foreground objects
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
# Remove noise
mask = cv2.medianBlur(mask, 5)
# Threshold the image to make it either black or white
_, mask = cv2.threshold(mask,127,255,cv2.THRESH_BINARY)
```

After preparing the images, applying openCVs findContours() function, the objects get outlined.

```
contours, hierarchy = cv2.findContours(mask,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)[-2:]
```

RETR_TREE computes a full hierarchy list for shapes inside other shapes. *CHAIN_APPROX_SIMPLE* removes all redundant points of the contourlines, leaving only the necessary minimum.

```
x,y,w,h = cv2.boundingRect(cnt)
cv2.rectangle(grayL,(x,y),(x+w,y+h),(0,255,0),3)
# Draw circle in the center of the bounding box
centerpoint_x = x + int(w/2)
centerpoint_y = y + int(h/2)
cv2.circle(grayL,(centerpoint_x, centerpoint_y),4,(0,255,0),-1)
```

The above code draws a bounding rectangle around the object and a point in its center. This points disparity value is used for the distance calculation. This might lead to error, if the object is for example ring-shaped, but taking the average of all the objects disparity-values will lead to big inaccuracies. A way to optimize this was not further pursued.

This rectangle and point were also used to prevent jumps caused by the inaccuracy of the contour-algorithm, as described in the next section.

IV.E Object Tracking and Direction

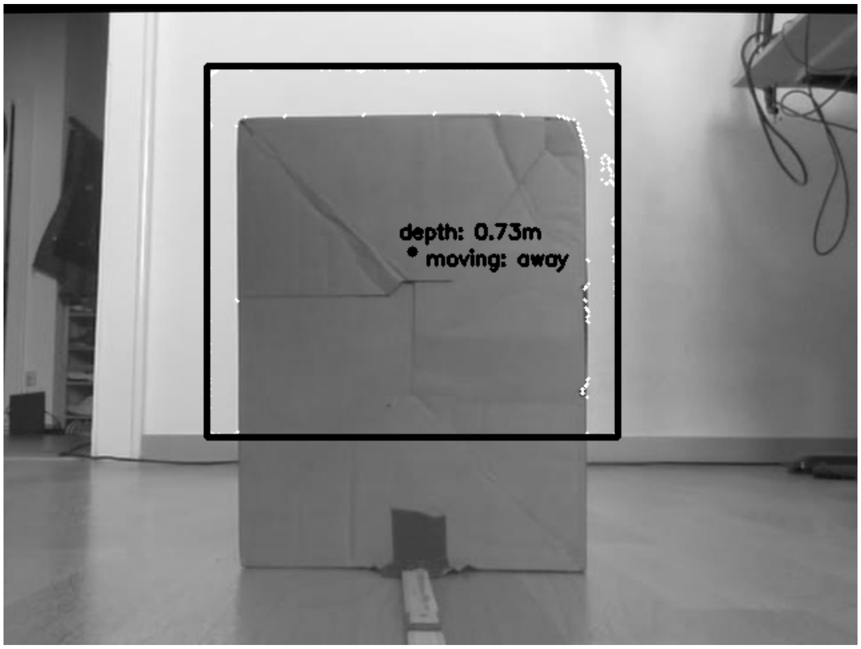


Fig. 12: Object with movement direction

The detected Objects were tracked by saving their coordinates, as well as their distance, to an array. Only Objects with their center-point within the rectangle-coordinates of the previous frame, as well as the changed depth within a certain range, were taken into consideration, otherwise skipped. The center-point coordinates and depth were then compared to the average of the last 3 points and its movement on the 3 axis determined as shown in the next table:

condition	direction
current center-point x smaller than average of the last center-points x	left
current center-point x bigger than average of the last center-points x	right
current center-point y smaller than average of the last center-points y	up
current center-point y bigger than average of the last center-points y	down
current center-point z smaller than average of the last center-points z	closer
current center-point z bigger than average of the last center-points z	away

IV.E.1 Measurement

With the test video being a box moving away from us, an accuracy of 70,5% was achieved with this simple implementation. Almost all wrong calculated directions were caused by incorrect disparity-values when the box wasn't moving, altering around the correct value. For example with the box standing 80cm away from the camera, the values tends to jump between 79 and 81cm, causing false directions.

V Problems

The results of computer-vision and especially stereo-vision are very prone to environmental factors[12] [13]. During development and research such problems were repeatedly encountered, dramatically affecting the result.

1) Shadows and object borders
2) Noisy Data
3) Lighting conditions on the object and on the camera
4) Reflecting, transparent, identical, textureless surfaces
5) Disadvantageous hardware
6) Blur because of motion or missing focus

VI Conclusion

The most important conclusion of this study is quite sobering. With the experience gained from working with the system, there are way too many factors that influence the result. Even more important those factors or variables are not always controllable.

This means from time to time, results from some functions are more or less arbitrary. Especially the calibration of the stereo cameras is quite a difficult task. Most of the time the result of openCV calibration, meaning removing of Distortion as well as rectifying images, isn't comprehensible. A big variable are the calibration-pictures, and their quality, as a consequence this means the camera used is quite important. In our case the camera is designed for this purpose, but this doesn't mean the camera doesn't have problems. This means the calibration is not as simple as most documentations suggest, at least in our experience. By this all following algorithms, for example computation of distance, are inferred by this, lets call it blur.

If those problems are left aside the result for distance as well as direction are quite satisfying. With a better setup that doesn't depend as much on calibration, for example usage of a Intel RealSense camera could enhance the result enormously. And almost as important, would ease up development since the camera module for JetBot in combination with the nVidia Jetson Nano development board does also provide some problems which won't be discussed at this point.

As a final conclusion, on a theoretical level, the initial problem doesn't sound complex. But once we factor in the practical part the difficulty level rises significantly. All in all this means the result of this study is quite satisfying once it's viewed from an objective standpoint.

From an educative perspective this topic was quite well chosen, because it had us digging deep into computer vision. Often, you can solve your tasks just by appending codelines found on Stackoverflow and googling error messages. In this project, particularly the calibration part, forced us to read all about the theory to understand what went wrong.

References

[1] Waveshare. (2020) Jetbot ai kit. [Online]. Available: https://www.waveshare.com/wiki/JetBot_AI_Kit

[2] NVIDIA. (2020) Nvidia jetson nano. [Online]. Available: https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-nano/

[3] Waveshare. (2021) Imx219-83 stereo camera. [Online]. Available: https://www.waveshare.com/wiki/IMX219-83_Stereo_Camera

[4] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2004.

[5] C. Loop and Z. Zhang, "Computing rectifying homographies for stereo vision," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 1, 1999, pp. 125–131 Vol. 1.

[6] W. Förstner and E. Gülch, "A fast operator for detection and precise location of distinct points, corners and centres of circular features."

[7] G. Gallego and A. J. Yezzi, "A compact formula for the derivative of a 3-d rotation in exponential coordinates," *CoRR*, vol. abs/1312.0788, 2013. [Online]. Available: http://arxiv.org/abs/1312.0788

[8] openCV. (2021) Camera calibration and 3d reconstruction - calibratecamera. [Online]. Available: https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga3207604e4b1a1758aa66acb6ed5aa65d

[9] ——. (2021) Camera calibration and 3d reconstruction - stereocalibrate. [Online]. Available: https://docs.opencv.org/master/d2/d85/classcv_1_1StereoSGBM.html

[10] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, 2008.

[11] openCV. (2021) Stereosgbm class reference. [Online]. Available: https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga91018d80e2a93ade37539f01e6f07de5

[12] M. Gosta and M. Grgic, "Accomplishments and challenges of computer stereo vision," 10 2010, pp. 57 – 64.

[13] M. A. Gennert, "A computational framework for understanding problems in stereo vision," Ph.D. dissertation, Massachusetts Institute of Technology, 1987.