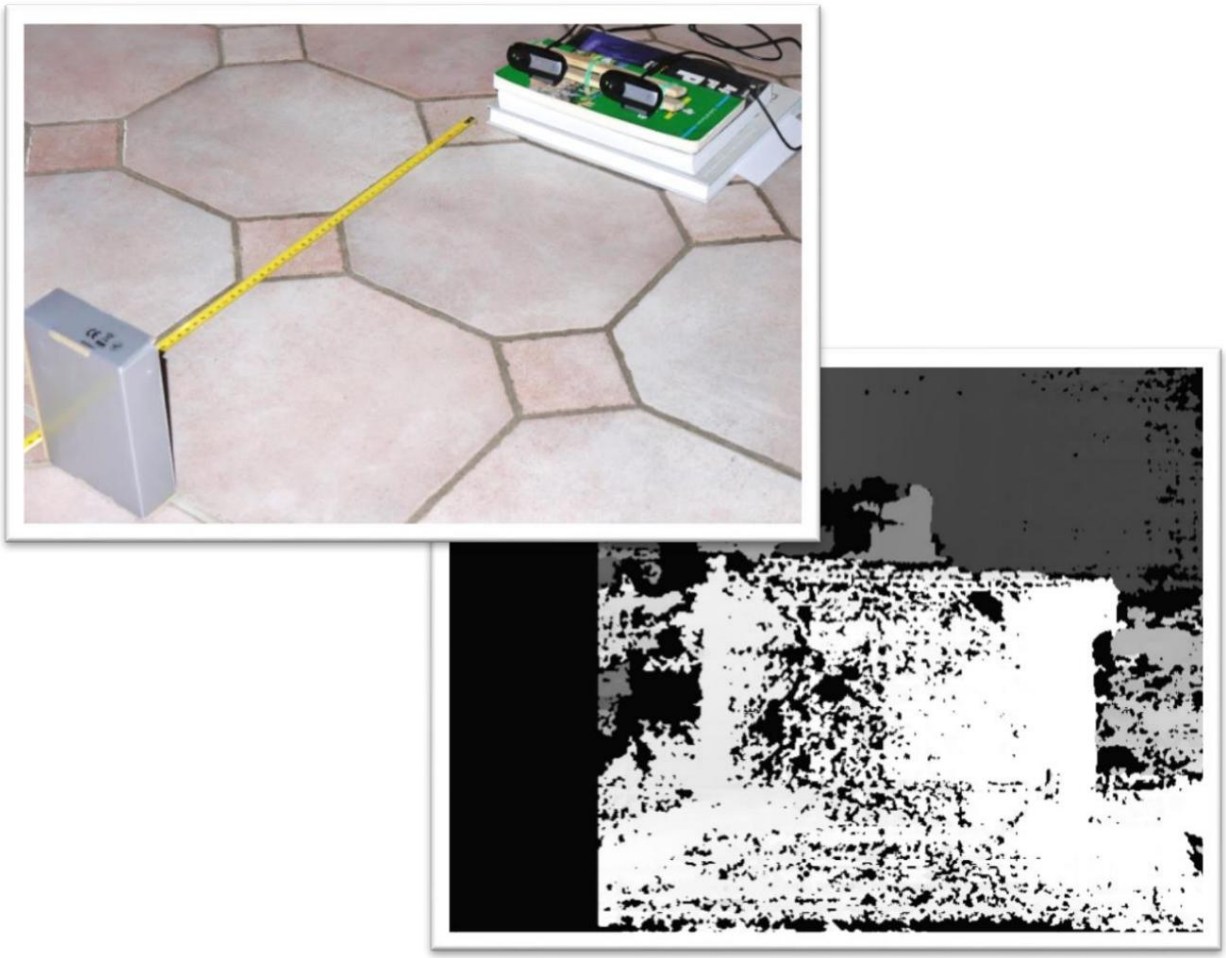
Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Stereo-Vision



Author:

Uhrweiller Frédéric and Vujasinovic Stephane
Registration number 58566 EIT Matriculation number: 59092 Mechatronics

Email: uhfr1011@hs-karlsruhe.de Email: vust1011@hs-karlsruhe.de

Supervisor: Prof. Dr.-Ing. Ferdinand Olawsky

Project coordinator MMT: Prof. Dr. Peter Weber

Project Code: 17SS_OL_Clean Room

Submission deadline: September 30, 2017

Table of Contents

List of Figures

code directory

1 Introduction

In order to understand stereo vision, one must first explain how a simple camera works, how it is basically constructed and which parameters have to be controlled.

A. What is stereo vision?

Stereoscopy is a technique that takes 2 images of the same scene and then constructs a disparity map of the scene. From this disparity map it is possible to measure the distance to an object and create a 3D map from the scene.

B. Stereo vision in the clean room robot

The aim of the large project is to develop a clean room robot that can measure particles in the entire clean room. In order to carry out the measurement without any problems, the robot must orientate itself without collisions. For this orientation we have decided to only use cameras and to measure the distance to an object we need at least two cameras (stereo vision). The two cameras have a distance of 110 mm between each other.

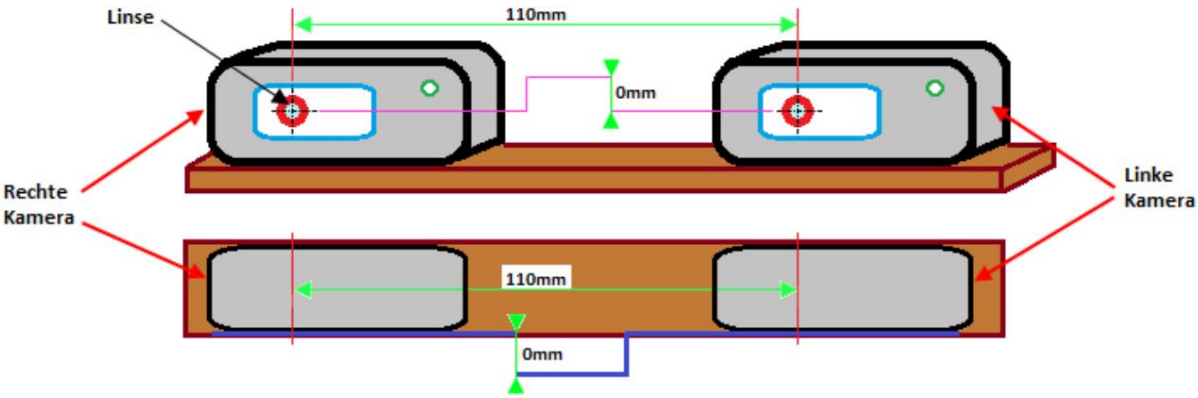


Figure 1: Stereo camera scheme

The larger this distance, the better it is possible to evaluate the distance to the scene for objects that are far away.

A Python program using the OpenCV library was implemented in this project to calibrate the cameras and to measure the distance to the objects in the scene. (See Appendix)

Figure 2: OpenCV and Python logo

The stereo camera consists of two Logitech Webcam C170.

Figure 3: Photo from the Logitech Webcam C170 (source: www.logitech.fr)

Video views are optimal with images of 640x480 pixels. Focal length : 2.3mm.
The homemade stereo camera:

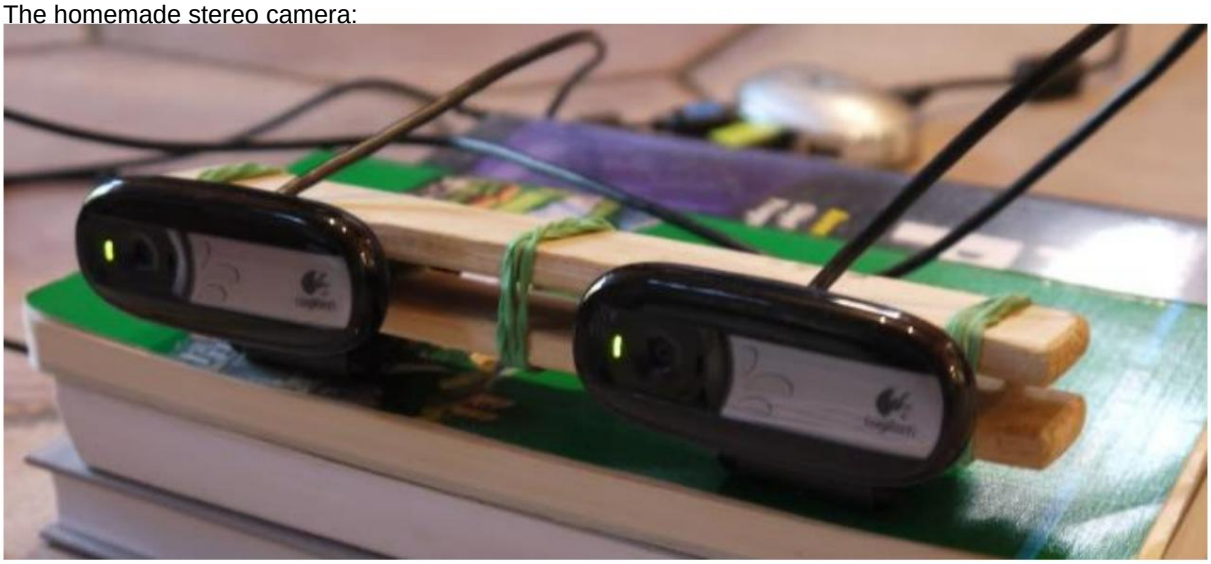


Figure 4: Photo of our homemade stereo camera

2. Camera model

Cameras capture the light rays of our environment. In principle, a camera works like our eye, the reflected light rays from our environment come to our eye and are collected on our retina.

The "pinhole camera" is the simplest model. It's a good simplified model to understand how a camera works. In the model, all light rays are stopped by a surface. Only the rays that penetrate through the hole are captured and on
of a surface in the camera configured in reverse. The picture below explains this principle.

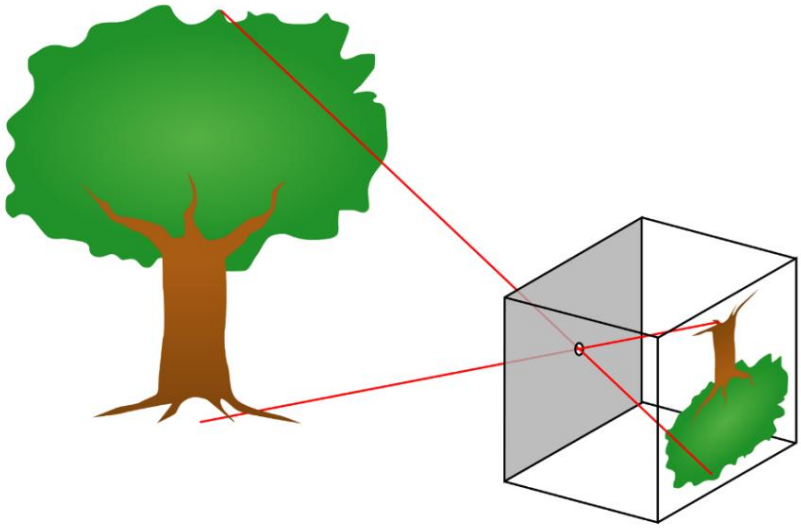


Figure 5: Functionality of a camera

Source: https://funsizephysics.com/use-light-turn-world-upside/

This principle is very simple, but it's not a good way to capture enough light in a fast exposure. Therefore, lenses are used to collect more rays of light in one place. The problem is that this lens introduces distortion.

There are two different types of distortions:

- the radial distortion
- the tangential distortion

The radial distortion comes from the shape of the lens itself and the tangential distortion comes from the geometry of the camera. The images can then be corrected using mathematical methods.

The calibration process allows creating a model of the geometry of the camera and a model of the distortion of the lens. These models form the intrinsic parameters of a camera.

A. Focal length

The relative size of the image projected onto the surface in the camera depends on the focal length. In the pinhole model, focal length is the distance between the pinhole and the area where the image is projected.

The Thales theorem then gives: -x = f * (X / Z)

With: • x: image of the object (minus sign comes from inverting the image)

- X: Size of the object
- Z: Distance from the hole to the object
- f: focal length, distance from the hole to the image

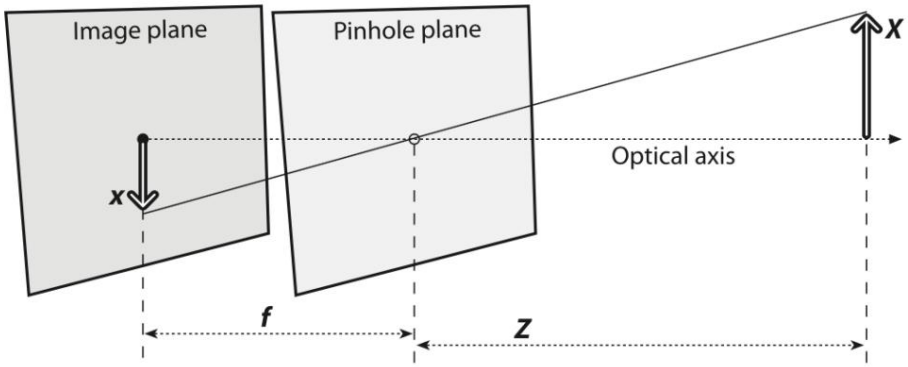


Figure 6: Projected object

Source: Learning OpenCV 3 - O'Reilly

Since the lens is not perfectly centered, two parameters are introduced, Cx and Cy, for the horizontal and vertical displacements of the lens, respectively. The focal lengths on the X and Y axes are also different because the image area is rectangular. This gives the following formula for the position of the object on the surface.

$$x_{\text{screen}} = f_x\left(\frac{X}{Z}\right) + c_x, \qquad y_{\text{screen}} = f_y\left(\frac{Y}{Z}\right) + c_y$$

The projected points of the real world on the screen can be modeled in the following way. M is the intrinsic matrix here.

$$q = MQ, \quad \text{where} \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

B. Distortion of the lens

Theoretically it is possible to build a lens that does not cause distortion with a parabolic lens. In practice, however, it is much easier to make a spherical lens than a parabolic lens. As previously discussed, there are two types of distortion. The radial distortion that comes from the shape of the lens and the tangential distortion that comes from the assembly process of the camera.

There is no radial distortion at the optic center and it increases progressively as one approaches from the edges. In practice, this distortion remains small, it is sufficient to do a Taylor expansion up to the third term. The following formula results.

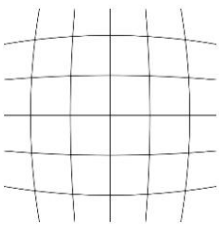


Figure 7: Radials Distortion (source: wikipedia)

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Frederic Uhrweiler
Stephane Vujasinovic

x and y are the coordinates of the original point on the image plane and this is used to calculate the position of the corrected point.

There is also a tangential distortion because the lens is not built perfectly parallel with the image plane. To correct this, two additional parameters are introduced, p1 and p2.

$$x_{\text{corrected}} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$
$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

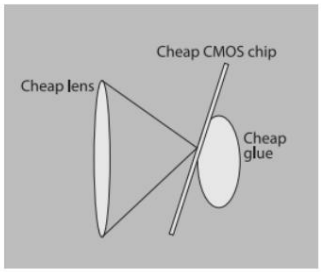


Figure 8. Tangential Distortion (Source: Learning OpenCV 3 - O'Reilly)

C. Calibration with OpenCV

The OpenCV library allows us to calculate the intrinsic parameters using specific functions, this process is called calibration. This is made possible with different views of a chess board.

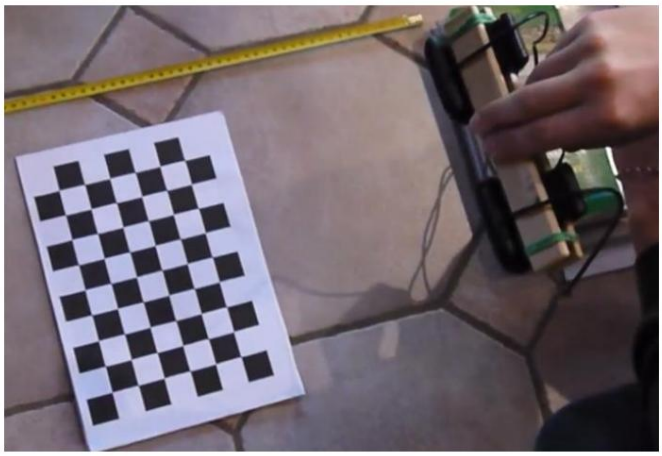


Figure 9: Photo while pictures are taken for calibration

The program for taking the pictures for later calibration is called **“Take_images_for_calibration.py”**
When the corners of the chess board are detected on both cameras, two windows will open with the detected image for each camera. The images are then either saved or deleted by the user. Good pictures can be seen in which the corners are very clearly visible. These images will later be used for the calibration in the main **program “Main_Stereo_Vision_Prog.py” .** OpenCV recommends having at least 10 images for each camera to get a good calibration. We got good results with 50 images for each camera.

To calibrate the cameras, the Python code finds the corners of the chessboard on each image for each camera using the OpenCV function: *cv2.findChesssboardCorners*

The position of the corners for each image are then stored in one image vector and the object points for the 3d scene are stored in another vector. Then use these Imgpoints and Objpoints in the *cv2.calibeCamera()* function at the output
the camera matrix, the distortion coefficients, the rotation and translation vectors
returns.

The *cv2.getOptimalNewCameraMatrix()* function allows us to get precise camera matrices that we will later use in the *cv2.stereoRectify()* function.

After calibrating with OpenCV we get the following matrix M for our camera:

Matrix M without rectification (right camera):

885,439	0	301,366
0	885,849	233,812
0	0	1

Matrix Mrekt Rectified (Right Camera):

871,463	0	303,497
0	869,592	233,909
0	0	1

Matrix M without rectification (left camera):

748,533	0	345,068
0	749,062	228,481
0	0	1

Matrix Mrekt Rectified (Left Camera):

730,520	0	349,507
0	725,714	227,805
0	0	1

3. Stereo imaging

A. Explanation

The stereo vision allows to see the depth in an image, to make measurements in the image and to do 3D localizations. Among other things, points that match between the two cameras must be found. From this one can then derive the distance between the camera and the point. The geometry of the system is used to simplify the calculation.

Stereo imaging involves four steps:

1. Removal of radial and tangential distortion by mathematical calculations. This gives undistorted images.

2. Rectifying the angle and spacing of the images. This stage allows both images to be coplanar on the Y axis, thus facilitating the search for correspondences and it is only necessary to search on a single axis (namely the X axis).

3. Finding the same feature in the right and left images. This gives a disparity map showing the differences between the images on the x-axis.
4. The final step is triangulation. The disparity map is transformed into distances by triangulation.

Step 1: Removal of the distortion
Step 2: Rectify
Step 3: Find the same feature in both images
Step 4: Triangulation

B. Triangulation

The final step, triangulation, assumes that both projection images are coplanar and that the left image's horizontal pixel row is aligned with the corresponding left image's horizontal pixel row.

With the previous hypotheses, one can now construct the following picture.

The point P lies in the environment and is mapped to pl and pr in the left and right images, with the corresponding coordinates xl and xr. This allows us to introduce a new quantity, the disparity: d = xl - xr. It can be seen that the further away the point P is, the smaller the quantity d becomes. So the disparity is inversely proportional to the distance.

The distance can be calculated with the following formula: Z=f*T/(xl-xr)

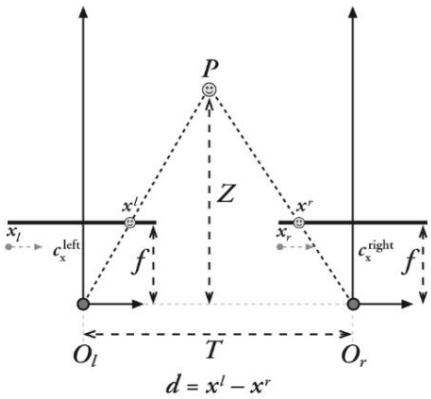


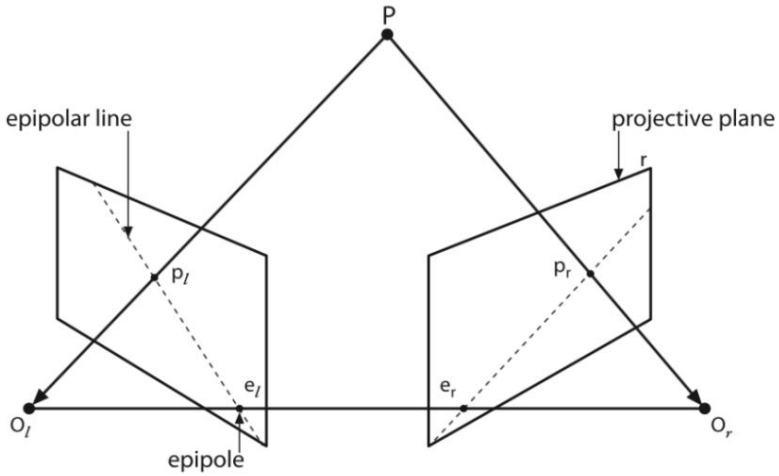
Figure 10: Triangulation (Source: Learning OpenCV 3 - O'Reilly)

It can be seen that there is a non-linear relationship between disparity and distance. When the disparity is close to 0, small disparity differences result in large distance differences. This is reversed when the disparity is large. Small differences in disparity then do not lead to large differences in distance. From this one can conclude that the stereo vision has a high depth resolution, only for objects that are close to the camera.

However, this method only works if the configuration of the stereo camera is ideal. In reality, however, this is not the case. Therefore, the left and right images are mathematically aligned parallel. Of course, the cameras must be positioned physically parallel, at least approximately.

Before explaining the method of mathematically aligning the images, we must first understand Epipolar Geometry.

C. Epipolar Geometry



The image above shows us the model of an imperfect stereo camera that consists of two pinhole camera models. The crossing of the line of the projection centers (Ol, Or) with the projection planes creates the epipolar points el and er. The lines (pl, el) and (pr, er) are called epipolar lines. The image of all possible points of a point on one projection plane is the epipolar line that lies on the other image plane and goes through the epipolar point and the point sought. This allows the point search to be limited to a single dimension rather than an entire plane.

So the following points can be summarized:

- Each 3D point in a camera's view is included in the Epipolar Plan
- A feature in a plane must be on the corresponding epipolar lines of the another level (epipolar condition)
- A two-dimensional search of a corresponding feature is converted to a one-dimensional search if one knows the epipolar geometry. • The order of the points is preserved, ie that two points A and B are found in the same order on the epipolar line of one plane as on the other plane.

D. Essential and Fundamental Matrices

To understand how the epipolar lines are calculated, one must first explain the essential and fundamental matrices (corresponding to matrices E and F).
The essential matrix E contains the information on how the two cameras are physically arranged with one another. It describes the localization of the second camera relative to the first camera using translation and rotation parameters. These parameters cannot be read directly in the matrix, as this is used for configuration. In the *Stereo Calibration* section we explain how to calculate R and T (rotation matrix and translation vector).
The matrix F contains the information of the essential matrix E, for the physical arrangement of the cameras and the information about the intrinsic parameters of the cameras.

The relation between the projected point on the left picture pl and the one on the right
Image pr is defined like this:

pr TEpl=0

One might think that this formula fully describes the relationship between the left and right points. However, one must note that the 3x3 matrix E is of rank 2. This means that this formula is the equation of a straight line.

Thus, in order to fully define the relationship between the points, one must consider the intrinsic parameters.

We recall that q=Mp, with the intrinsic matrix M.
Substituting in the previous equation gives: *qr*
^{T}*(Ml* $^{-1}$*) TEMl 1ql=0*

One replaces:
F=(Ml $^{-1}$*) TEML* $^{-1}$

And so gets:
qrTFql =0

E. Rotation matrix and translation vector

Now that we have explained the essential matrix E and the fundamental matrix F, we need to see how the rotation matrix and the translation vector are calculated.

We define the following notations:

- *Pl* and *Pr* define the positions of the point in the coordinate system of the left and right cameras
- *Rl* and *Tl* (or *Rr* and *Tr)* define the rotation and translation of the camera to the point in the environment for the left (or right) camera.
- R and T are the rotation and the translation, the coordinate system of the right brings the camera in the coordinate system of the left camera.

The result is:
Pl=RlP+Tl and *Pr=RrP+Tr*

You also have:
Pl=RT(Pr-T)

With these three equations we finally get:

R=RrRlT
T=Tr-RTl

F. Stereo rectification

So far we have dealt with the topic "stereo calibration". It was about the description of the geometric arrangement of both cameras. The task of rectification is to project the two images so that they are in the exact same plane

and align the pixel rows precisely so that the epipolar lines become horizontal in order to be able to find the correspondence of a point in the two images more randomly.
As a result of the process of aligning both images, we get 8 terms, 4 for each camera:

- a distortion vector
- a rotation matrix *Rrect to* be applied to the image
- a rectified camera matrix *Mrect*
- a non-rectified camera matrix *M*

OpenCV allows us to calculate these terms using two algorithms: the Hartley algorithm and the Bouguet algorithm.

1. Hartley Algorithm

The Hartley algorithm searches for the same points in the two images. He tries to minimize the disparities and to find homographs that put epipoles in infinity. With this method one does not need the intrinsic parameters
to be calculated for each camera.

An advantage of this method is that calibration is possible just by observing points in the scene. A big disadvantage is that you don't have any scaling of the image, you only have information about the relative distance. You cannot measure exactly how far an object is from the cameras.

2. Bouguet Algorithm

The Bouguet algorithm uses the calculated rotation matrix and translation vector to rotate both projected planes by half a turn so that they are in the same plane. This makes the main rays parallel and the planes coplanar, but not yet lined up in rows. This will then be done later.

In the project we used the Bouguet algorithm.

4. Functionality of programs for stereo imaging

As already mentioned, the program is coded in Python and the OpenCV library is used. We chose the Python language and the OpenCV library because we already had experience with it and because there is a lot of documentation about it. Another argument for this decision is that we only wanted to work with "open source" libraries.

Two Python programs were developed for this project.

The first **“Take_images_for_calibration.py”** is for taking good images that will later be used in the calibration of the two cameras (distortion calibration and stereo calibration).

The second program and thus the main program **"Main_Stereo_Vision_Prog.py"** is used for stereo imaging. In this program we calibrate the cameras with the

Images taken create a disparity map and thanks to a straight line equation found experimentally we can measure the distance for each pixel. A WLS filter is used at the end to better detect the edges of objects.

The Python programs can be found in the appendix.

A. Packages used

The following packages were imported into the program: • The
version of OpenCV.3.2.0 with opencv_contrib (contains the stereo functions) as
Called “cv2” in Python, contains:
o the image processing library
o the functions for stereo vision

• Numpy.1.12. o
Used for matrix operations (images consist of matrices)
• Workbook by openpyxl
o Package to be able to write data in an Excel file
• "normalize" of the library sklearn 0.18.1
o sklearn enables machine learning but in this project one uses
only the WLS filter

```
# Package importation
import numpy as np
import cv2
from openpyxl import Workbook # Used for writing data into an Excel file
from sklearn.preprocessing import normalize
```

Code 1: Package Import

B. Video loop

To work with the cameras you have to activate them first. The function
cv2.VideoCapture() activates both cameras by entering the number of ports of each camera (Two objects are thus created in the program that can use methods from *cv2.VideoCapture()* class).

```
# Call the two cameras
CamR= cv2.VideoCapture(0)
CamL= cv2.VideoCapture(2)
```

Code 2: Activation of the cameras with OpenCV

To get an image from the cameras, the method *cv2.VideoCapture().read()*
used, as output you get an image of the scene that the camera is looking at at the moment this function is called. To then get a video you have to call this method again and again in an infinite loop. To be more efficient it is recommended to convert the BGR images to Gray images, this is done with the *cv2.cvtColor()* function

executed.

```
while True:
    retR, frameR= CamR.read()
    retL, frameL= CamL.read()
    grayR= cv2.cvtColor(frameR,cv2.COLOR_BGR2GRAY)
    grayL= cv2.cvtColor(frameL,cv2.COLOR_BGR2GRAY)
```

Code 3: image processing

To view the video on the PC, the *cv2.imshow()* function is used, it allows opening a window where the video can be shown.

```
cv2.imshow('VideoR',grayR)
cv2.imshow('VideoL',grayL)
```

Code 4: Show images

A break is used to get out of the infinite loop. This becomes active whenever the user presses the spacebar. The detection that a key has been pressed is performed thanks to the *cv2.waitKey()* function .
At the end, the two cameras used must be deactivated using the method *cv2.VideoCapture().release()* and the opened windows are with the function *cv2.destroyAllWindows()* closed.

```
    # End the Programme
    if cv2.waitKey(1) & 0xFF == ord(' '):
        break

# Release the Cameras
CamR.release()
CamL.release()
cv2.destroyAllWindows()
```

Code 5: Typical exit for a program

C. How the **“Take_images_for_calibration.py”** program works

When this program is started, both cameras become active and two windows open so the user can see where the chessboard is positioned in the images.

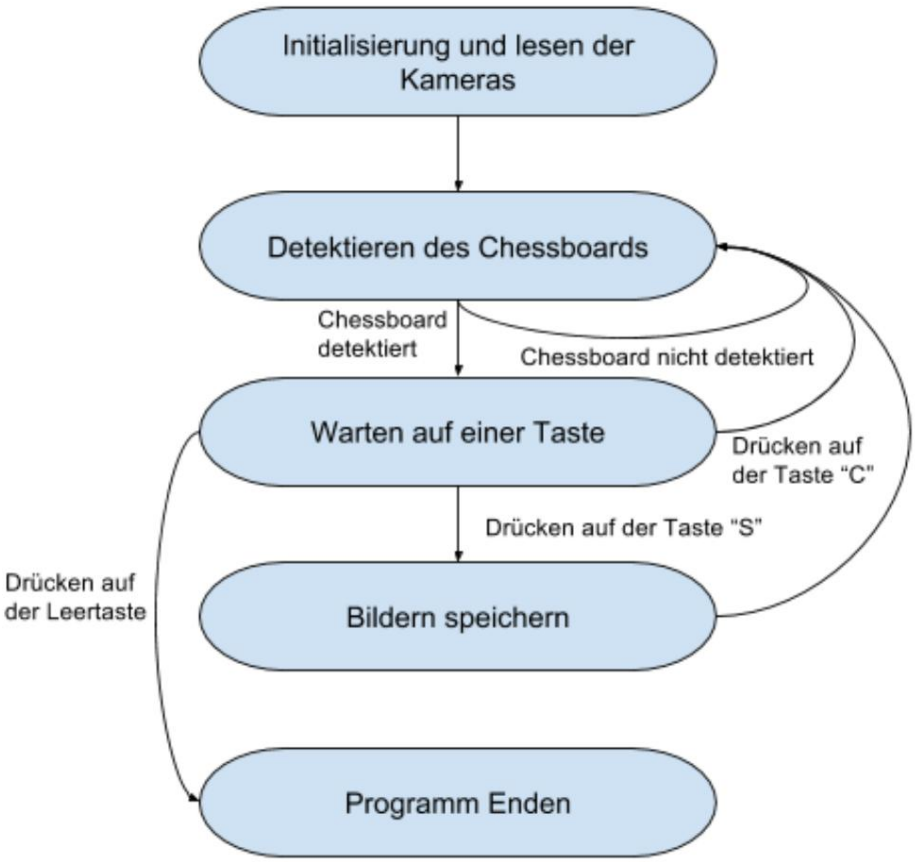


Diagram 1: ***How "Take_images_for_calibration.py" works***

1. Vectors for calibration

The *cv2.findChessboardCorners()* function will search for a defined number of chessboard corners and the following vectors will be generated:

- **imgpointsR:** contains the coordinates of the corners in the right image (in image space)
- **imgpointsL:** contains the coordinates of the corners in the left image (in image space)
- **objpoints:** contains the coordinates of the corners in object space

The precision of the coordinates of the found corners is increased using the *cv2.cornerSubPix()* function .

2. Capture the images for calibration

Once the program has recognized the position of the chess board corners on both images, two new windows will open where you can evaluate the captured images. If the images are not blurry and look good, press the “s” (Save) key to save the images. If the opposite is the case you can click the “c” (Cancel) key

Press so that the images are not saved. With the *cv2.drawChessboardCorners()* function , the program superimposes a chessboard pattern on the images.

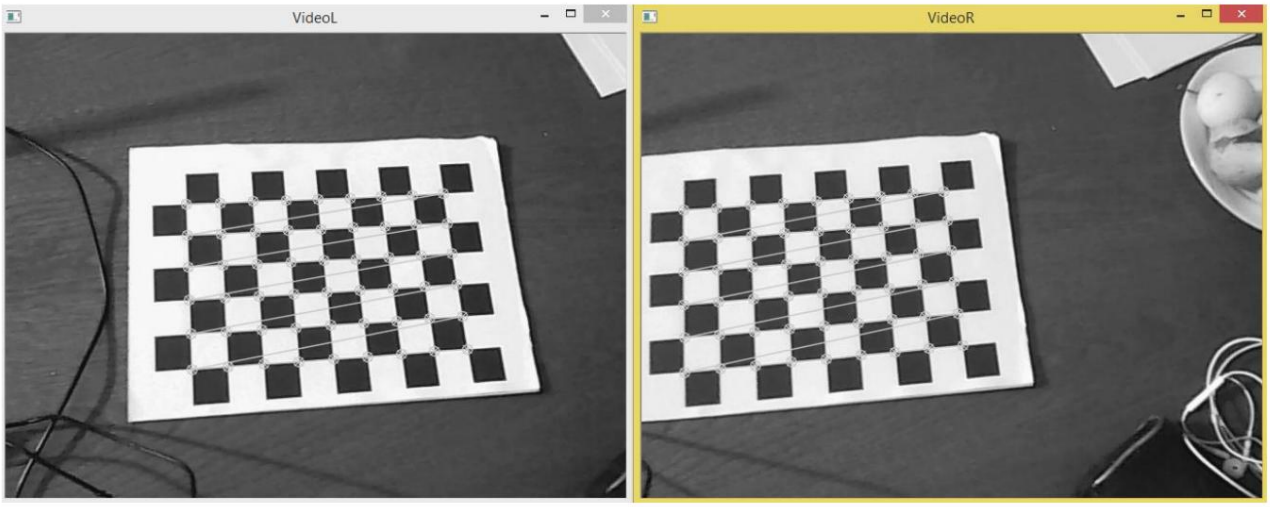


Figure 11: Detection of the corners of a chess board

D. Operation of the program **“Main_Stereo_Vision_Prog.py”**

In the initialization, the cameras are first calibrated individually to remove the distortion. Then stereo calibration is performed (eliminate rotation, align epipolar lines). In an infinite loop, the images are processed and a disparity map is generated.

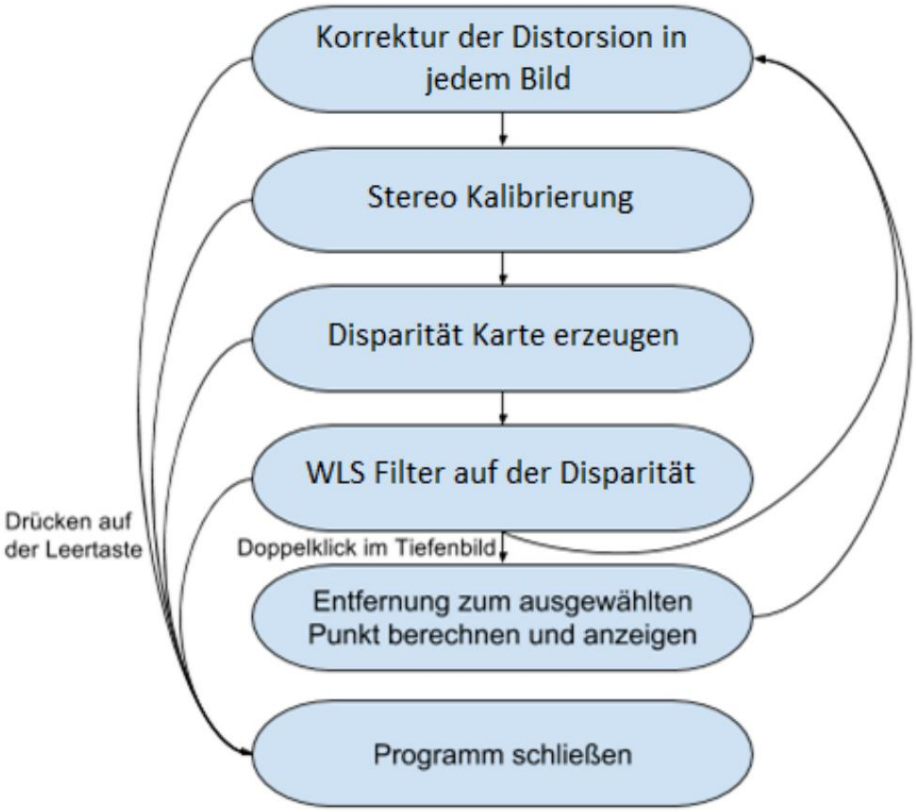


Diagram 2: How "Main_Stereo_Vision_Prog.py" works

1. Calibration of the distortion

For the correction of the distortion of the cameras, the images taken with the program “Take_images_for_calibration.py” are used.
This calibration is based on “Take_images_for_calibration.py” by storing the position of the checkerboard corners in **imgpoints** and **objpoints .** The *cv2.calibrateCamera()* function is used to calibrate new camera matrices (The Camera Matrix
describes the projection of a point in the 3D world in a 2D image), distortion
Get coefficients, rotation and translation vectors for each camera, which will later be used to remove the distortion from each camera. For the best
Getting camera matrices for each camera will be the function *cv2.getOptimalNewCameraMatrix()* used (precision increase).

```
#   Right Side
retR, mtxR, distR, rvecsR, tvecsR = cv2.calibrateCamera(objpoints,
                                                        imgpointsR,
                                                        ChessImaR.shape[::-1],None,None)

hR,wR= ChessImaR.shape[:2]
OmtxR, roiR= cv2.getOptimalNewCameraMatrix(mtxR,distR,
                                           (wR,hR),1,(wR,hR))
```

Code 6: Calibration of the distortion in Python

Frederic Uhrweiler
Stephane Vujasinovic

2. Calibration of the stereo camera

The *cv2.StereoCalibrate()* function is used for stereo calibration , it calculates the transformation between both cameras (one camera serves as a reference for the other).

The *cv2.stereoRectify()* function allows to bring the epipolar lines of the two cameras on the same plane. This transformation simplifies the work for the function that creates the disparity map, because then the block match only has to be found in one dimension. With this function you also get the Essential Matrix and the Fundamental Matrix which are needed in the next function.

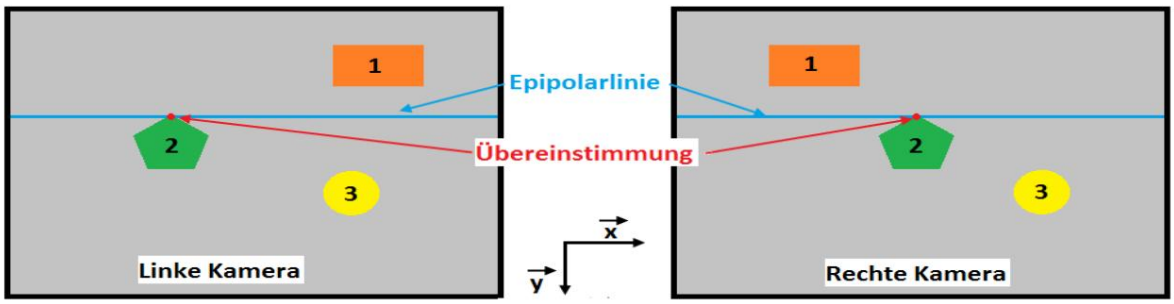


Figure 12: Principle of the epipolar lines

The function *cv2.initUndistortRectifyMap()* results in an image that has no distortion. These images are then used in the calculation of the disparity map.

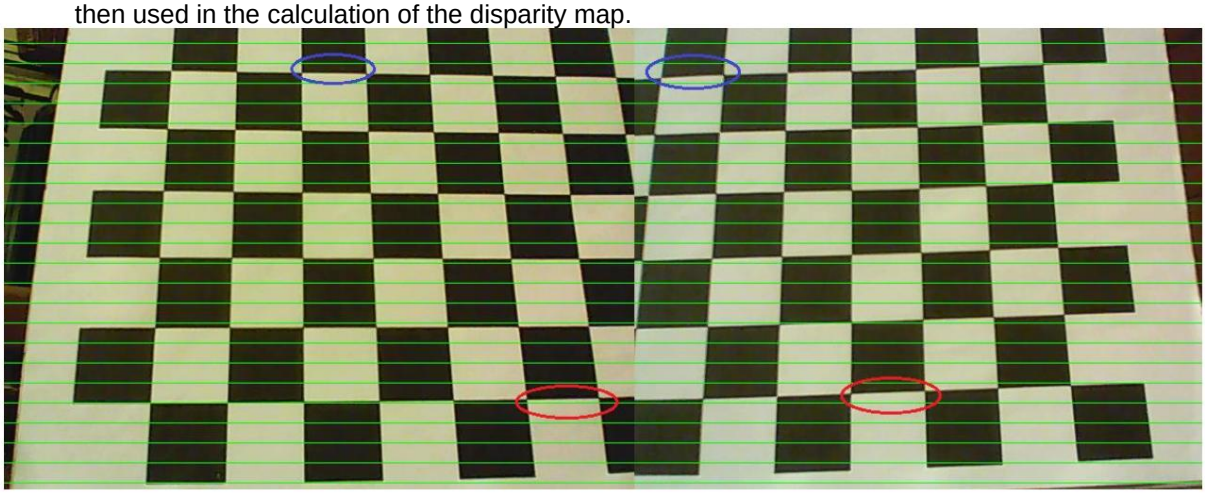


Figure 13: Without calibration

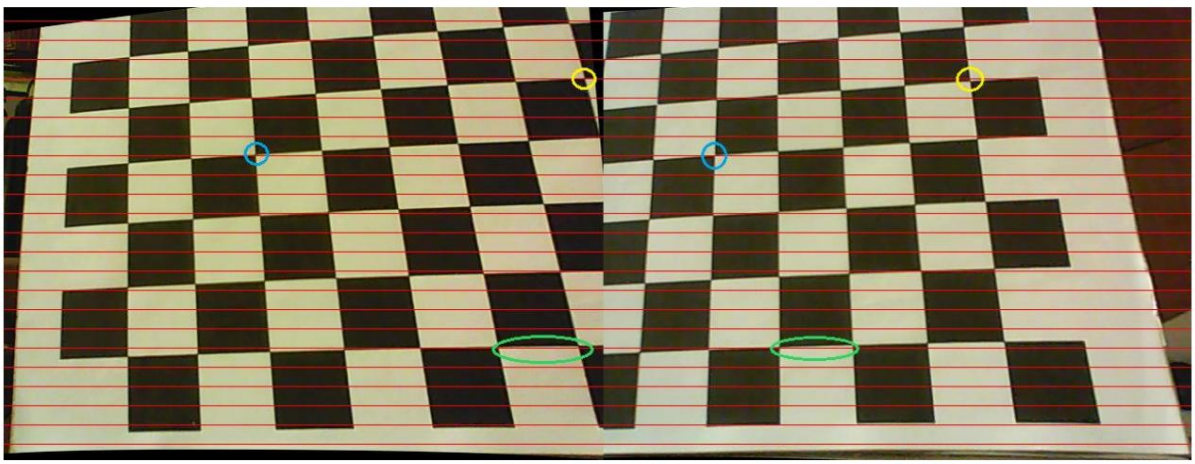


Figure 14: With calibration

Frederic Uhrweiler
Stephane Vujasinovic

3. Calculation of the disparity map

To calculate the disparity map, a StereoSGBM object is created using the function *cv2.StereoSGBM_create().* This class uses a semi-global matching algorithm (Hirschmüller, 2008) to obtain a stereo match between the right and left camera images.

How the Semi-Global Matching Algorithm works:

The following scene is presented to the stereo camera:

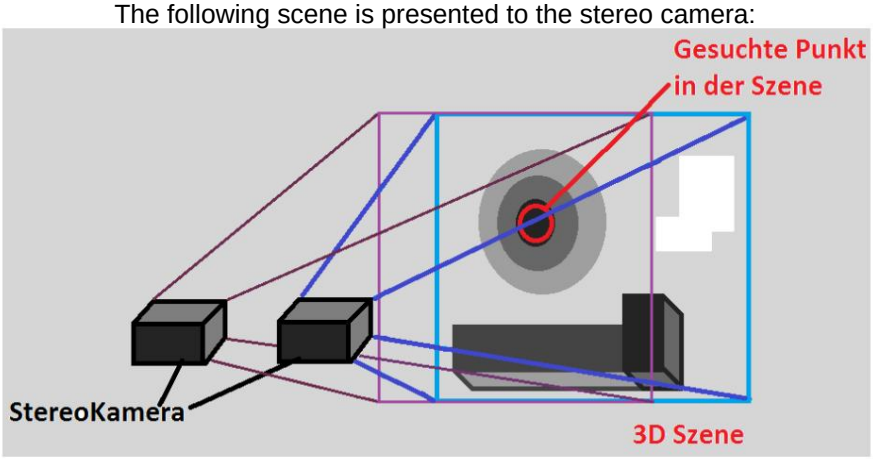


Figure 15: Example of a stereo camera observing a scene

The size of the blocks is defined in the input parameters. These blocks replace the pixels if the size (block size) is greater than 1. The generated SGBM object compares the blocks from a reference image with the blocks from the match image. For example, if the stereo calibration is done well, a block of row four from the reference image must be compared to all blocks of the match image that are only on the fourth row. In this way
the calculation of the disparity map becomes more efficient.

Let's take the previous example with the fourth row blocks to explain how the disparity map is made.

In the figure below, you have to compare the block (4,7) of the fourth row, seventh column from the base picture with all other blocks (4,i) of the fourth row (same epipolar line) from the match picture.

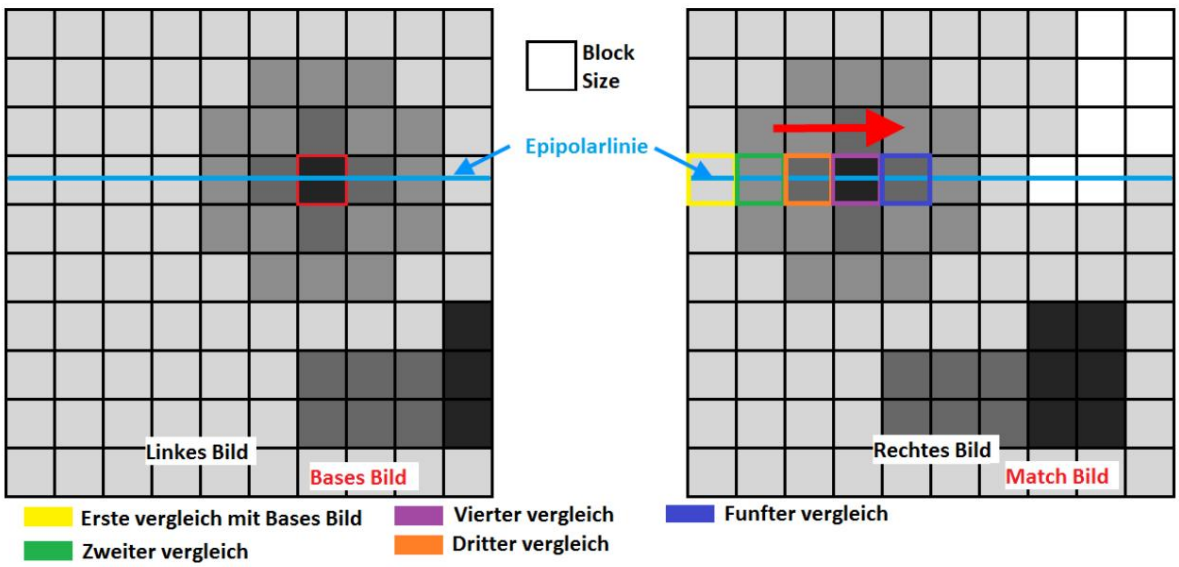


Figure 16: Block matching using StereoSGBM

Frederic Uhrweiler
Stephane Vujasinovic

The greater the correspondence between the reference block and the match block, the more likely it must be the same point in the environment.

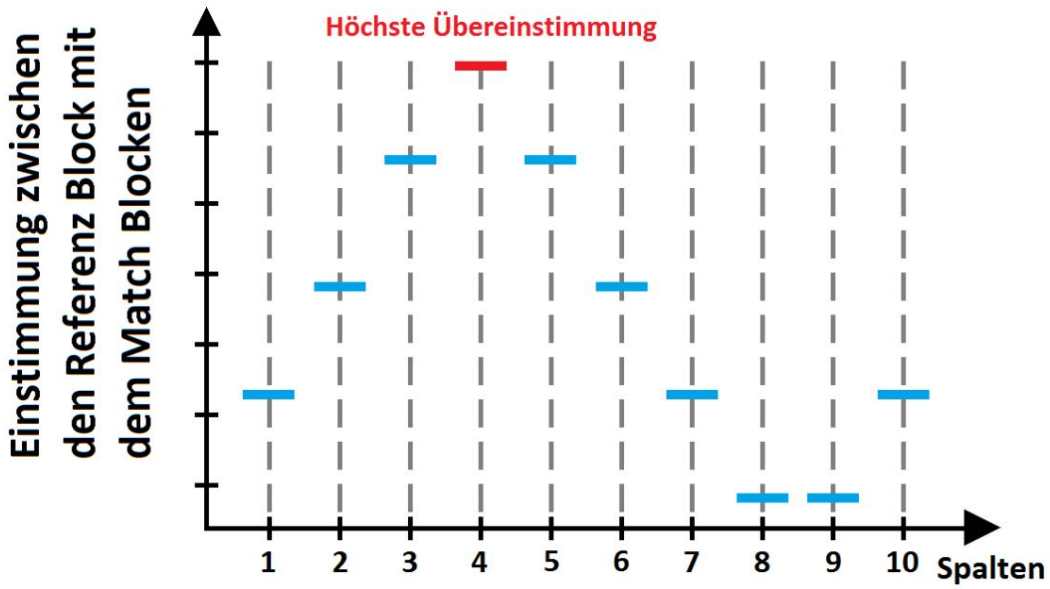


Figure 17: Same block detection with StereoSGBM

You can see in this example that the reference Block(4,7) is the highest match level with the Match Block(4,4).
In theory it should work like this, but in practice 4 other directions are still processed by default in OpenCV to be more precise with the same method as for one direction.

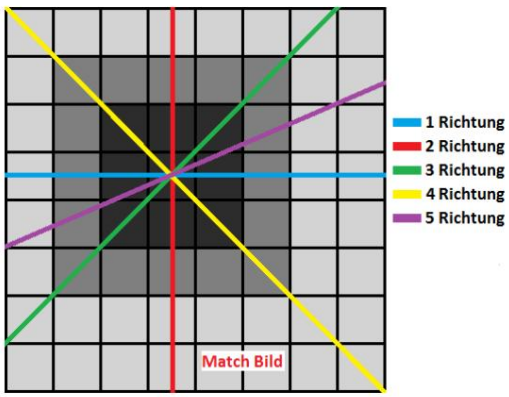


Figure 18: Example of the five directions of the StereoSGBM algorithm in OpenCV

To find the disparity, the coordinates of the match block are subtracted with the reference block, then the absolute value is taken from the result and the larger this value, the closer the object is to the stereo camera.

The program uses calibrated black and white images for the calculation of the disparity map, it is also possible to work with BGR images but this would require more computer time. The card is computed using a method from our stereo created object, cv2.StereoSGBM_create().compute().

```
# Show the result for the Disparity Map
disp= ((disp.astype(np.float32)/ 16)-min_disp)/num_disp
cv2.imshow('disparity', disp)
```

Code 7: Show the disparity map

With our parameters that were set in the initialization, we get the following Result for the disparity map.

```
# Create StereoSGBM and prepare all parameters
window_size = 3
min_disp = 2
num_disp = 130-min_disp
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)
```

Code 8: Parameters for the instance of StereoSGBM

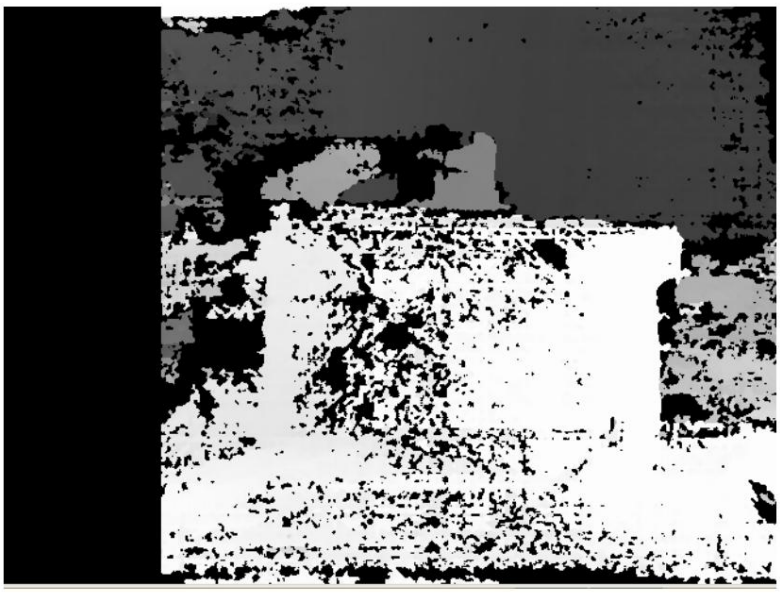


Figure 19: Result for the disparity map

There is still a lot of noise in this disparity map, and a morphological filter is used to eliminate it. A “closing” filter is used with the OpenCV function *cv2.morphologyEx(cv2.MORPH_CLOSE)* to remove the small black dots.

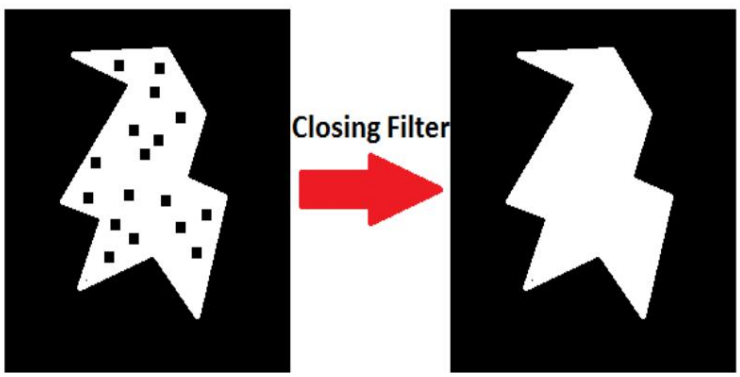


Figure 20: Example of a closing filter

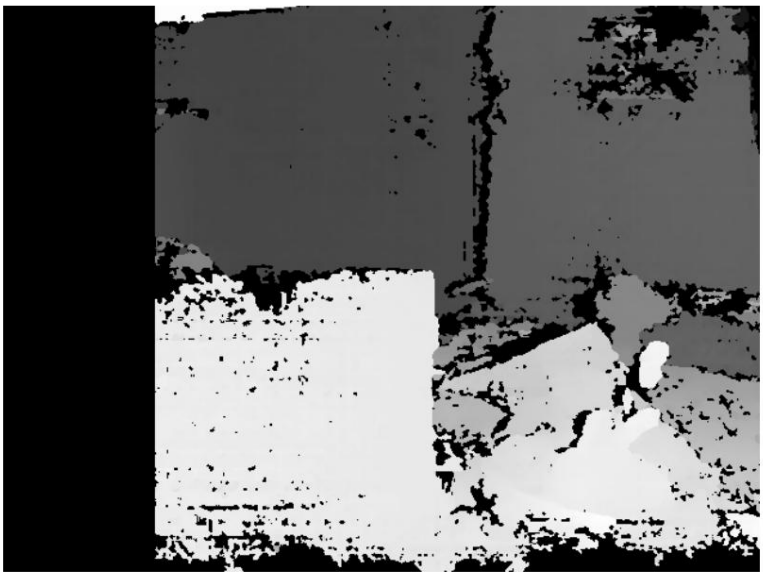


Figure 21: Result of a disparity map after a closing filter

Another example with the same scene to see the difference better.

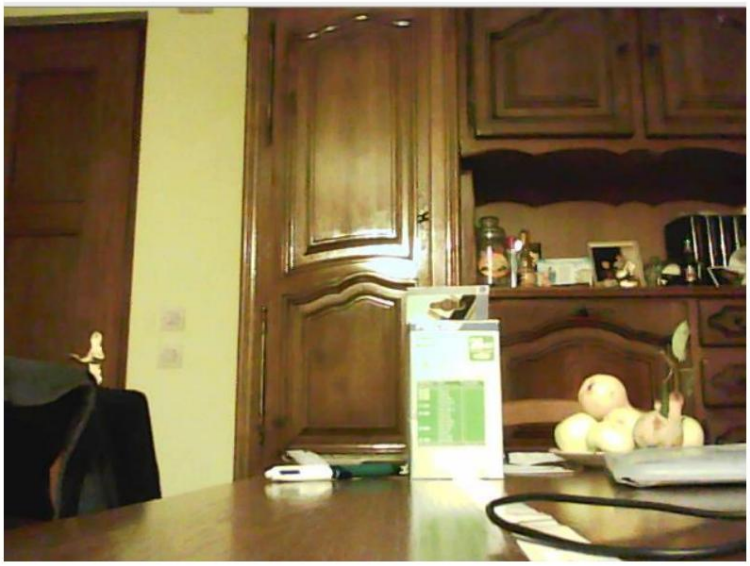


Figure 22: Normal scene seen from the left camera without rectification and calibration

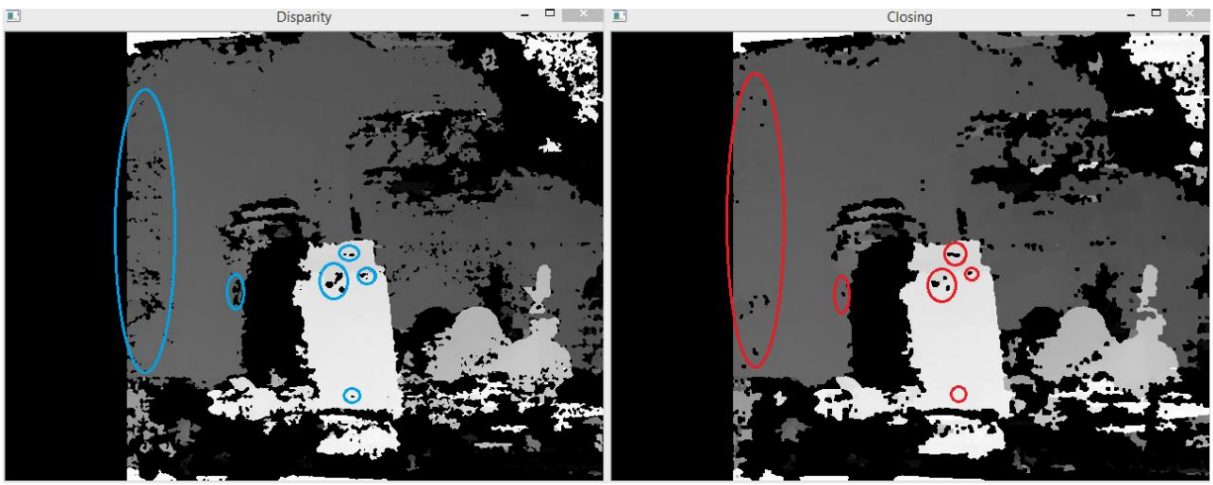


Figure 23: Disparity map of the upper scene without and with closing filter

4. Use of the WLS (Weighted Least Squares) filter

The result is not bad, but it is still very difficult to see the edges of objects because of the noise, so a WLS filter is used. First, the parameters of the filter must be specified in the initialization.

```
# FILTER Parameters for the WLS filter
lmbda = 80000
sigma = 1.8
visual_multiplier = 1.0
```

Code 9: Parameters for a WLS filter

Lambda is typically set to 8000, the larger this value the more the shapes become
attached the disparity map to the shapes of the reference image, we set it to 80000 because there were better results with this value. Sigma describes how strong our filter must be precise at the edges of objects.
Another stereo object is created with *cv2.ximgproc.createRightMatcher()* and is based on the first one. These two instances are then used in the WLS filter to generate a disparity map.

An instance of the filter is created using the *cv2.ximgproc.createDisparityWLSFilter()* function .

```
# Used for the filtered image
stereoR=cv2.ximgproc.createRightMatcher(stereo) # Create another stereo

# Create the WLS filter
wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=stereo)
wls_filter.setLambda(lmbda)
wls_filter.setSigmaColor(sigma)
```

Code 10: Creation of a WLS filter in Python

Then to apply the instance of the WLS filter the following method is called *cv2.ximgproc.createRightMatcher().filter(),* the values of our filter are then normalized with *cv2.normalize().*

```
# Using the WLS filter
filteredImg= wls_filter.filter(dispL,grayL,None,dispR)
filteredImg = cv2.normalize(src=filteredImg, dst=filteredImg, beta=0, alpha=255,
filteredImg = np.uint8(filteredImg)
```

Code 11: Implementation of the WLS filter in Python

An Ocean ColorMap was used to get better visualization with *cv2.applyColorMap().* The darker the color, the farther our subject is from the stereo camera.

Figure 24: Ocean ColorMap

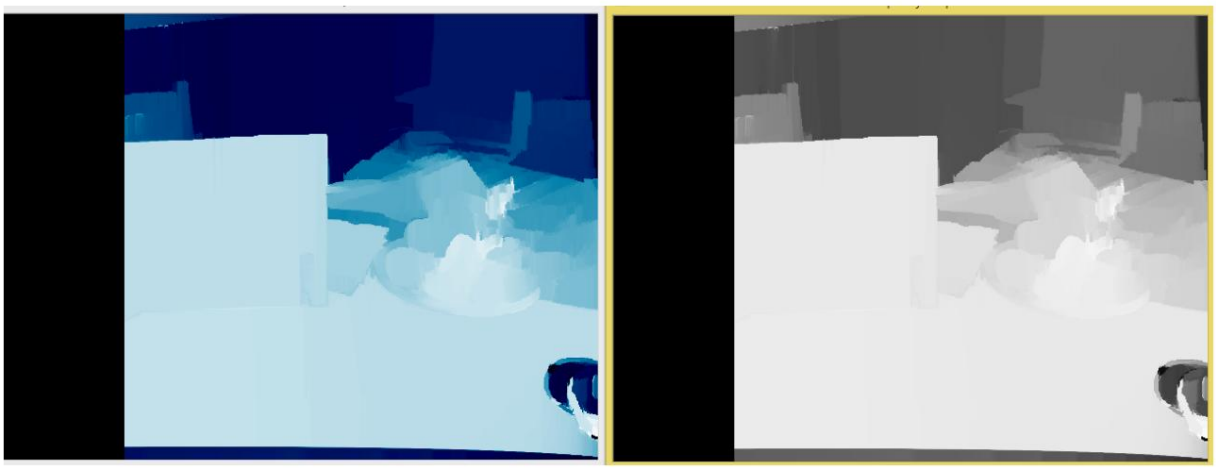


Figure 25: WLS filter on a disparity map with and without an ocean map color

In this way we get an image that can show the edge well but is no longer precise enough and no longer contains the good disparity values (the disparity map is encoded in float32 but the WLS result is encoded in uint8).
In order to use the good values to later measure the distance to objects, the coordinates x and y of the WLS filtered image are taken with a double click. This (x,y)
Coordinates are then used to get the disparity value from the disparity map and then measure the distance. The value returned is the mean of the disparity of a 9x9 pixel matrix.

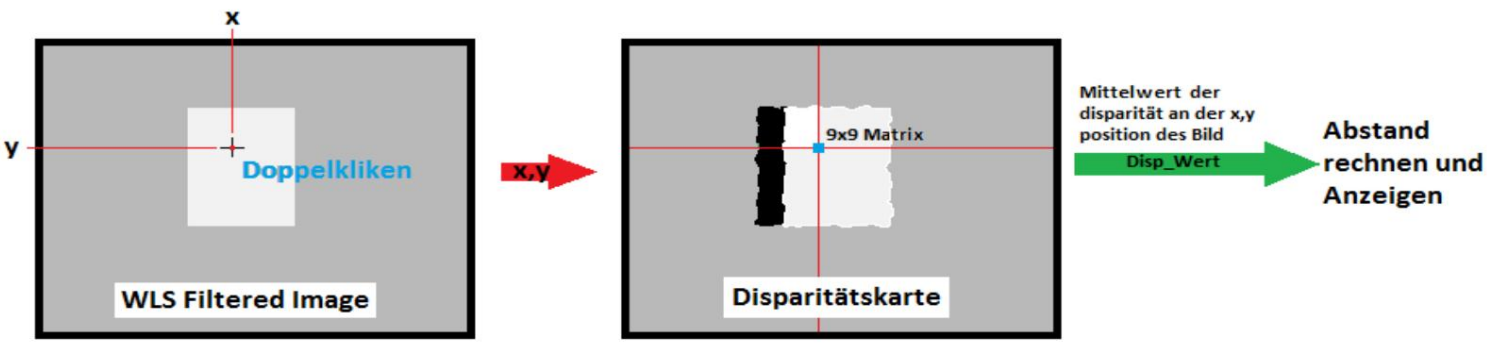
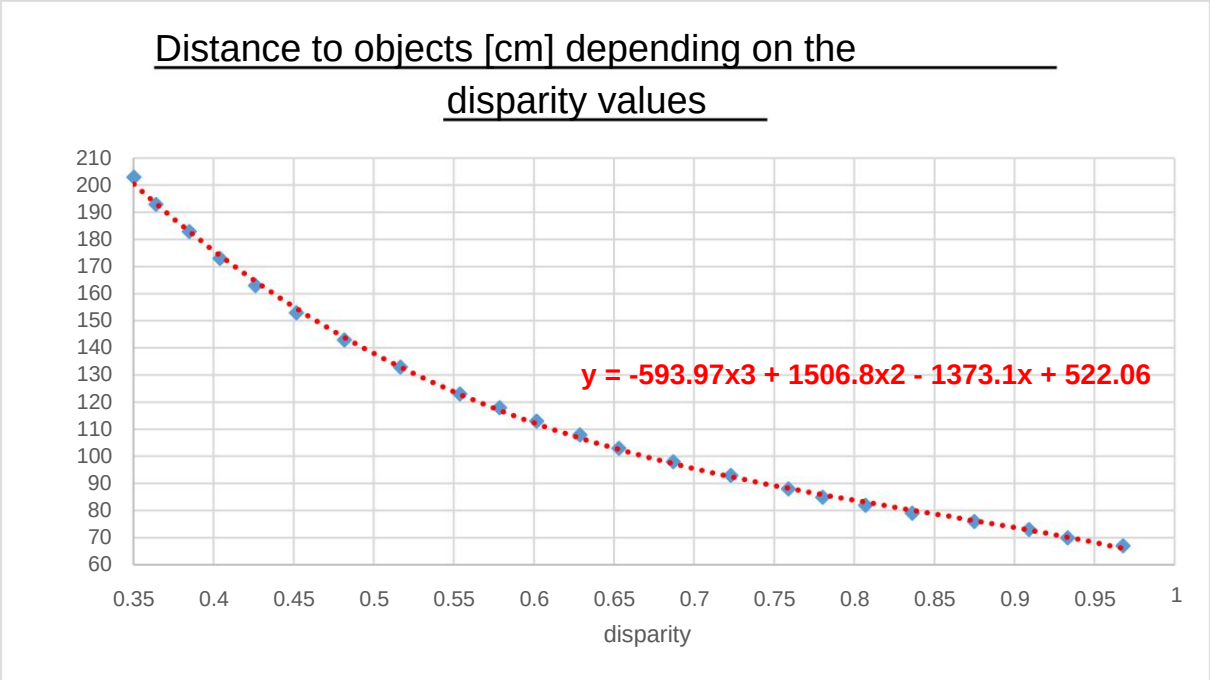


Figure 26: Calculation of the distance with the WLS filter and the disparity map

5. Measure the distance

After the disparity map has been generated, the distance needs to be determined. The work consists in finding the relation between the disparity value and the distance. To this done, we experimentally measured the disparity values at several points in order to determine a regression.



Line equation in the Python program

```
# Equation for the distance mesurements
Distance= -593.97*average**(3) + 1506.8*average**(2) - 1373.1*average + 522.06
```

*Code 12: The regression formula in **"Main_Stereo_Vision_Prog.py"***

In order to get this straight line equation of the regression, the *package openpyxl*
Used to save the disparity values in an Excel file. The lines in the program that caused the values to be saved were annotated in the program, but you can uncomment them if you need a new line equation.

The distance measurement is only valid from a distance of 67cm up to 203cm to get good results. The precision of the measurement also depends on the quality of the calibration. Our stereo cameras have been able to measure the distance to an object with a precision of +/- 5 cm.

Figure 27: Experimental measurement of the disparity depending on the distance to the object

6. Possible improvements

Possible improvements for the program:

- Take the form of the WLS filter and that to project on the disparity map.
  This projection would then be taken to take all of the disparity values found in the shape and the value that occurs most frequently is then set as the value for the entire area.

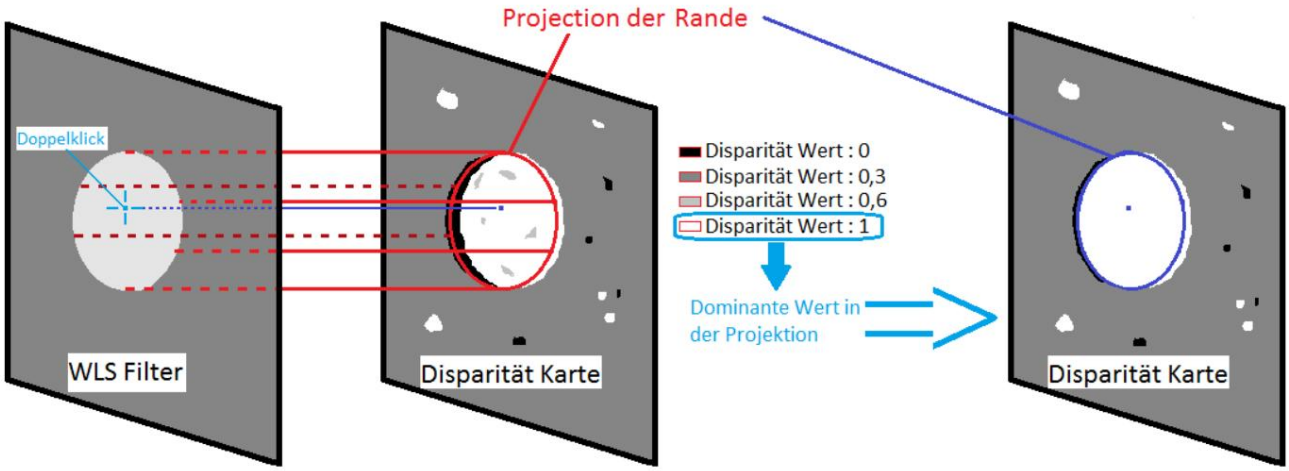


Figure 28: Possible improvement with the first suggestion

- Use of a bilateral filter on the calibrated images before the disparity map is generated, this way it would be possible not to use a WLS filter to use. That has to be checked, but the WLS filter is only used to see the edges of objects well, maybe there is a better way.

- In order to reduce the calculation time for the creation of the disparity map, the calibrated images should be reduced with the function *cv2.resize(cv2:INTER_AREA),* one should then make sure that the values of the essential matrices, fundamental matrices are also reduced proportionally.

- The creation of a depth map could also be beneficial.

- A camera more stable than ours where the rotation of the heads can really be prevented, this way one would only have to save the values of the matrices from the stereo calibration to be able to use them again. A lot of time could be saved in the initialization.

- Running the program on the GPU would also enable us smoother
  Get pictures when the stereo camera is in motion

5. Conclusion

A. Summary

This project work enabled us to work with the Python language and learn more about the OpenCV library. We have had the opportunity to familiarize ourselves with the subject of stereo vision in this project. A new topic that we are both very interested in and that is still very new compared to other distance measurement techniques that are available. There are also other methods to measure the distance with image processing, eg Time of Light (TOF) cameras, which are expensive and with which there is very little documentation. We also decided to use simple cameras because of the price.

B. Conclusion

In the project work carried out here, it is possible with the developed program to calculate a distance on the basis of a disparity map.

C. Outlook

So that the calculated distance values always remain correct, a new system for the cameras should be developed that avoids all free movements of the cameras. Thus, only matrix values would be used to perform the calibration faster. The straight line equation used would always return the exact distance to the object with greater precision, with less effort.

bibliography

OpenCV Python Tutorials

OpenCV documentation

Stack Overflow

RDMILLIGAN on its website rdmilligan.wordpress.com

Stereo Vision: Algorithms and Applications by Stefano Mattoccia, Department of Computers Science(DISI), University of Bologna

Stereo matching by Nassir Navab and slides prepared by Christian Unger

Oreilly Learning OpenCV

6. Appendix

Video about our project: https://youtu.be/xjx4mbZXaNc

The Python programs can be found in the **Python_Prog_Stereo_Vision** folder .
This consists of the programs:

- **Take_images_for_calibration.py”**
- **”Main_Stereo_Vision_Prog.py”**

The two programs in a .txt version can also be found in the same folder if you don't have Python installed on your computer.