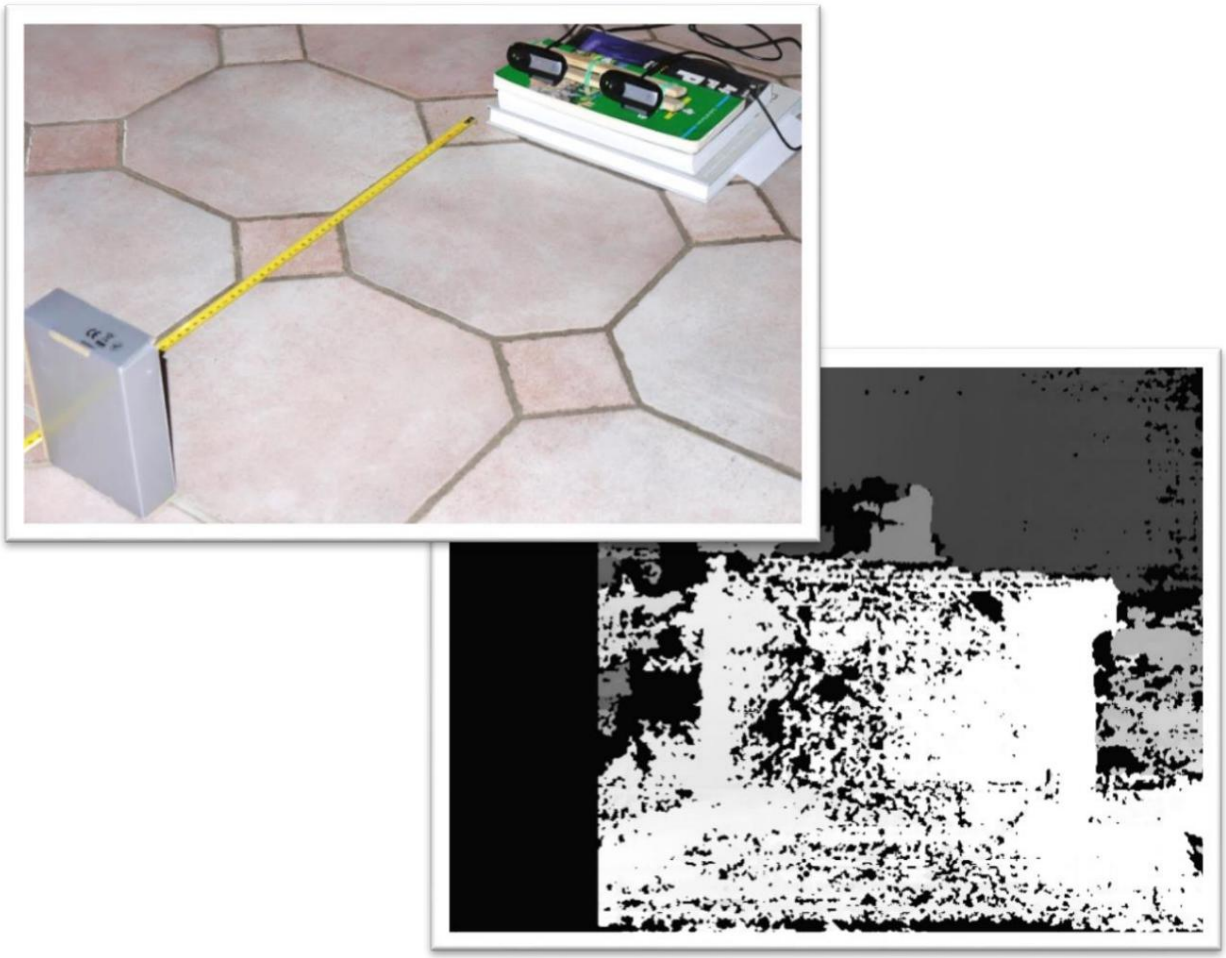
Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Stereo-Vision



Author:
Uhrweiller Frédéric and Vujasinovic Stéphane
Matriculation No. 58566 EIT
Matriculation number: 59092 Mechatronics

E-mail: uhfr1011@hs-karlsruhe.de
E-mail: vust1011@hs-karlsruhe.de

Supervisor: Prof. Dr.-Ing. Ferdinand Olawsky

MMT project coordinator: Prof. Dr. Peter Weber

Project code: 17SS_OL_Cleanroom

Deadline: September 30, 2017

Table of contents

list of figures

code directory

1. Introduction

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

A. Was ist Stereo-Vision?

Stereoscopy is a process that takes two images of the same scene and then creates 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 of the scene.

B. Stereo Vision in Cleanroom Robots

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

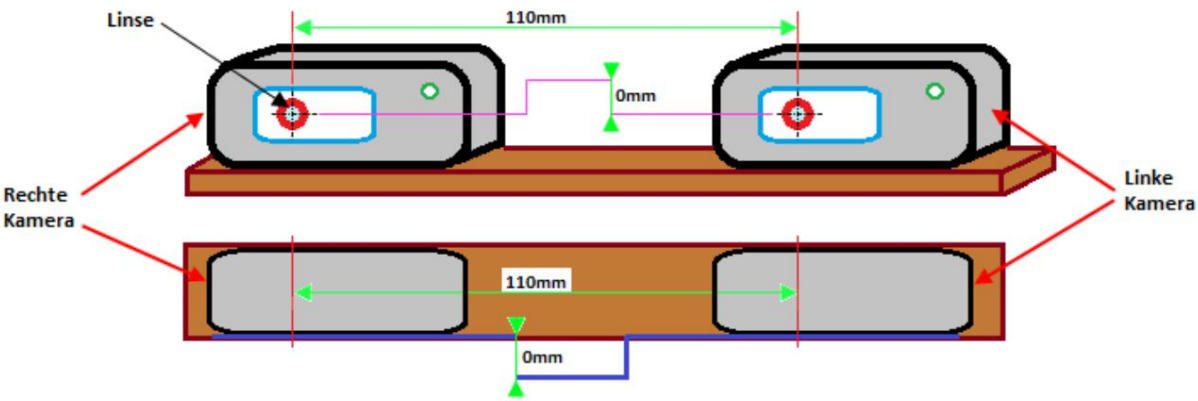


Figure 1: Stereo camera diagram

The larger this distance, the better it is possible to estimate 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 of the Logitech Webcam C170 (Source: www.logitech.fr)

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

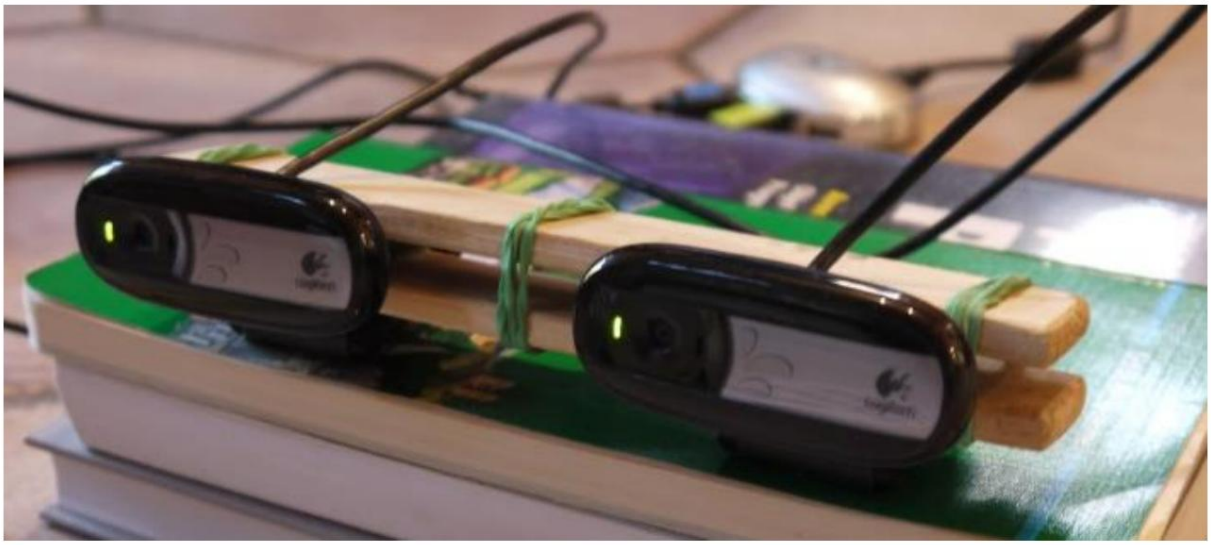


Figure 4: Photo of our self-built stereo camera

2nd camera model

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

The “pinhole camera” is the simplest model. It is a good simplified model to understand how a camera works. In this model, all light rays are stopped by a surface. Only the rays that pass through the hole are captured and projected onto
a surface in the camera is projected in reverse. The image below explains this principle.

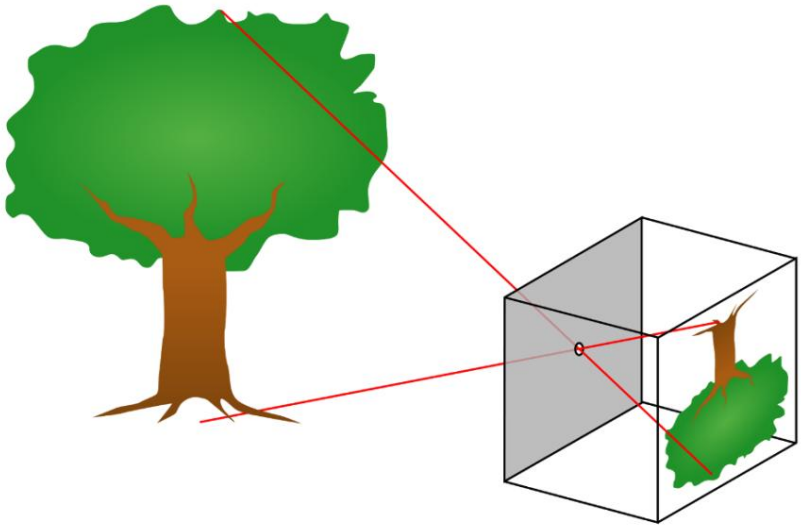


Figure 5: How a camera works

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

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

Two different types of distortions can be distinguished:

- Radial Distortion
- 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 to create a model of the geometry of the camera and a model of the distortion of the lens. These models represent 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, the focal length is the distance between the hole and the surface 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

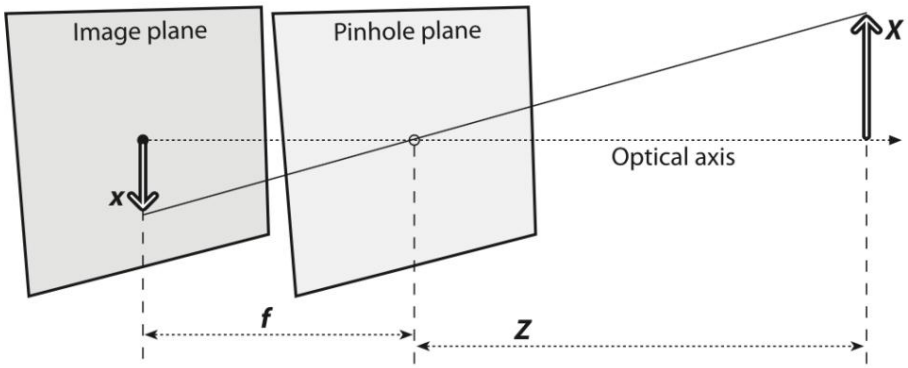


Figure 6: Projected object

Quelle: Learning OpenCV 3 - O'Reilly

Since the lens is not perfectly centered, two parameters are introduced, Cx and Cy for the horizontal and vertical displacement of the lens respectively. The focal length on the X and Y axes are also differentiated because the image surface 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 image surface can be modeled as follows. 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 using 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. Radial distortion which comes from the shape of the lens and Tangential distortion which is caused by the assembly process of the camera.

There is no radial distortion at the optical center and it becomes larger 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. This gives the following formula.

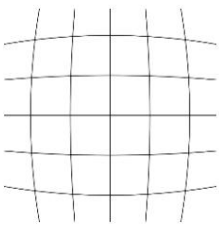


Figure 7: Radial Distortion (What: Wikipedia)

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

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

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

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

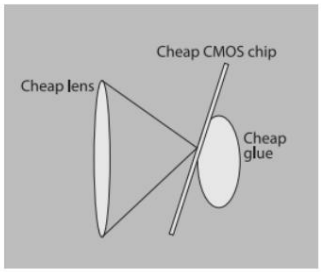


Figure 8. Tangential distortion (Source: Learning OpenCV 3 - O'Reilly)

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 using different views of a chessboard.

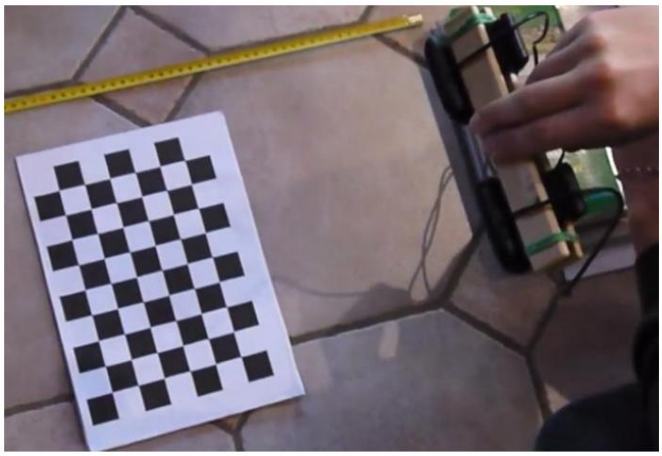


Figure 9: Photo while taking pictures for calibration

The program for taking pictures for later calibration is called
"Take_images_for_calibration.py"
When the corners of the chessboard are detected on both cameras, two windows open with the detected image for each camera. The images are then either saved or deleted by the user. Good images can be identified in which the corners are very clearly visible. These images are later used for 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 searches for 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 an image vector and the object points for the 3d scene are stored in another vector. You then use these Imgpoints and Objpoints in the function *cv2.calibeCamera()* which is at the output
the camera matrix, the distortion coefficients, the rotation and translation vector
returns.

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

After calibration 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. Declaration

Stereo vision makes it possible to detect depth in an image, take measurements in the image and perform 3D localization. To do this, points that match between the two cameras must be found. From this, the distance between the camera and the point can then be derived. The geometry of the system is used to simplify the calculation.

Stereo imaging involves these four steps:

1. Removal of radial and tangential distortion through mathematical calculations. This produces undistorted images.

2. Rectify the angle and distance of the images. This step allows both images to be coplanar on the Y axis, thus facilitating the search for correspondences and requiring the search to be carried out on only one axis (namely the X axis).

3. Find the same feature in the right and left images. This gives a disparity map that shows the differences between the images on the x-axis.
4. The last step is triangulation. The disparity map is transformed into Distances through triangulation.

Step 1: Removal of the distortion
Step 2: Rectify
Step 3: Finding 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 horizontal pixel row of the left image is aligned with the corresponding horizontal pixel row of the right image.

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. The disparity is therefore inversely proportional to the distance.

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

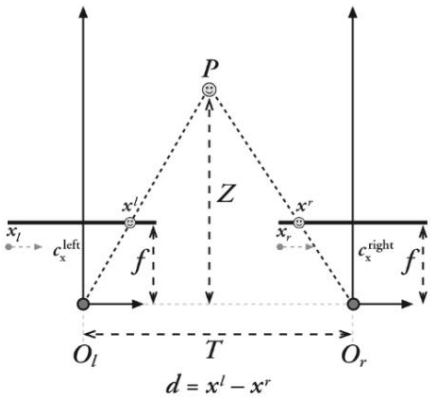


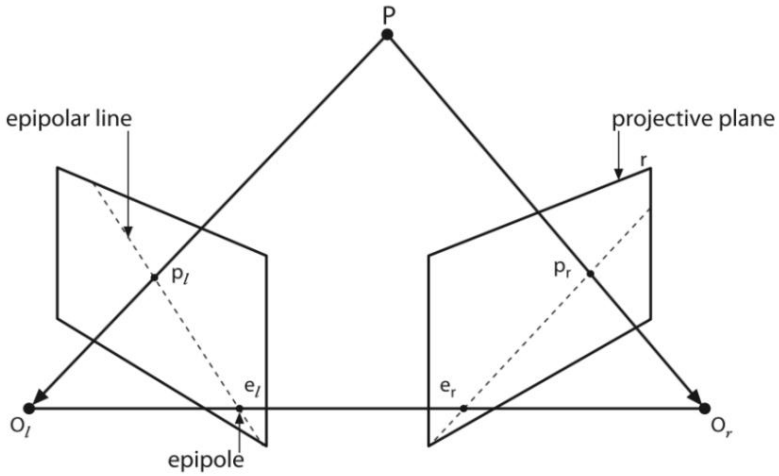
Figure 10: Triangulation (Source: Learning OpenCV 3 - O'Reilly)

It can be seen that there is a non-linear relationship between disparity and distance. When the disparity is close to 0, small differences in disparity lead to large differences in distance. This is the opposite when the disparity is large. Small differences in disparity do not lead to large differences in distance. From this, we can conclude that 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 physically positioned at least approximately parallel.

Before we explain the method of mathematically aligning the images, we must first understand epipolar geometry.

C. Epipolare Geometrie



The image above shows us a model of a non-perfect stereo camera consisting of two pinhole camera models. The epipolar points el and er are created by crossing the line of projection centers (Ol, Or) with the projection planes. 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 passes through the epipolar point and the searched point. This allows the search for the point to be limited to a single dimension rather than an entire plane.

So we can summarize the following points:

- Every 3D point in the view of a camera is included in the epipolar plan
- A feature in a plane must be on the corresponding epipolar line of the other level (epipolar condition)
- A two-dimensional search of a corresponding feature is converted to a one-dimensional search if the epipolar geometry is known. • The order of the points is preserved, ie two points A and B are found in the same order on the epipolar lines 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 each other. It describes the location of the second camera relative to the first camera using translation and rotation parameters. These parameters are not directly readable in the matrix, as it is used for projection. 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, 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 image pl and the one on the right
Image pr is defined as:

pr TEpl=0

One might think that this formula completely 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.

To fully define the relationship between the points, one must therefore take into account the intrinsic parameters.

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

Replace:
$F=(Ml^{-1})^{T}$ *TEM*P^{-1}

And thus receives:
qr TFql=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 to calculate the rotation matrix and the translation vector.

We define the following notations:

- *Pl* and *Pr* define the positions of the point in the left and right coordinate systems respectively.
  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 of the coordinate system of the right
  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 of “stereo calibration”. This involved describing the geometric arrangement of both cameras. The task of rectification is to project the two images so that they are in exactly the same plane

and to align the pixel rows precisely so that the epipolar lines become horizontal in order 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* that must 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 looks for the same points in both images. It tries to minimize the disparities and find homographies that set the epipoles at infinity. This method therefore does not require the intrinsic parameters

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 have no scaling of the image, you only have the information of 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 the translation vector to rotate both projected planes by half a turn so that they are in the same plane. This makes the principal rays parallel and the planes coplanar but not aligned in rows. This will be done later.

In the project we used the Bouguet algorithm.

4. How the programs for stereo imaging work

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 used to take good pictures, which 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 the 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 von openpyxl
  o Package to write data in an Excel file
- “normalize“ of the library sklearn 0.18.1
  o sklearn enables machine learning but in this project we use 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 Importation

B. Video Loop

To work with the cameras you must first activate them. The function
cv2.VideoCapture() activates both cameras by specifying the port number of each camera (two objects are thus created in the program, which can use the methods of the *cv2.VideoCapture()* class).

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

Code 2: Activating the cameras with OpenCV

To get an image from the cameras, the method *cv2.VideoCapture().read()*
used, the output is an image of the scene that the camera is viewing at the moment this function is called. To 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 into Gray images, this is done with the function *cv2.cvtColor()*

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 see the video on the PC, the function *cv2.imshow()* is used, it allows to open a window where the video can be shown.

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

Code 4: Display images

To get out of the infinite loop, a break is used. This is activated whenever the user presses the space bar. The detection that a key has been pressed is done 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 released 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 program **“Take_images_for_calibration.py”** works

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

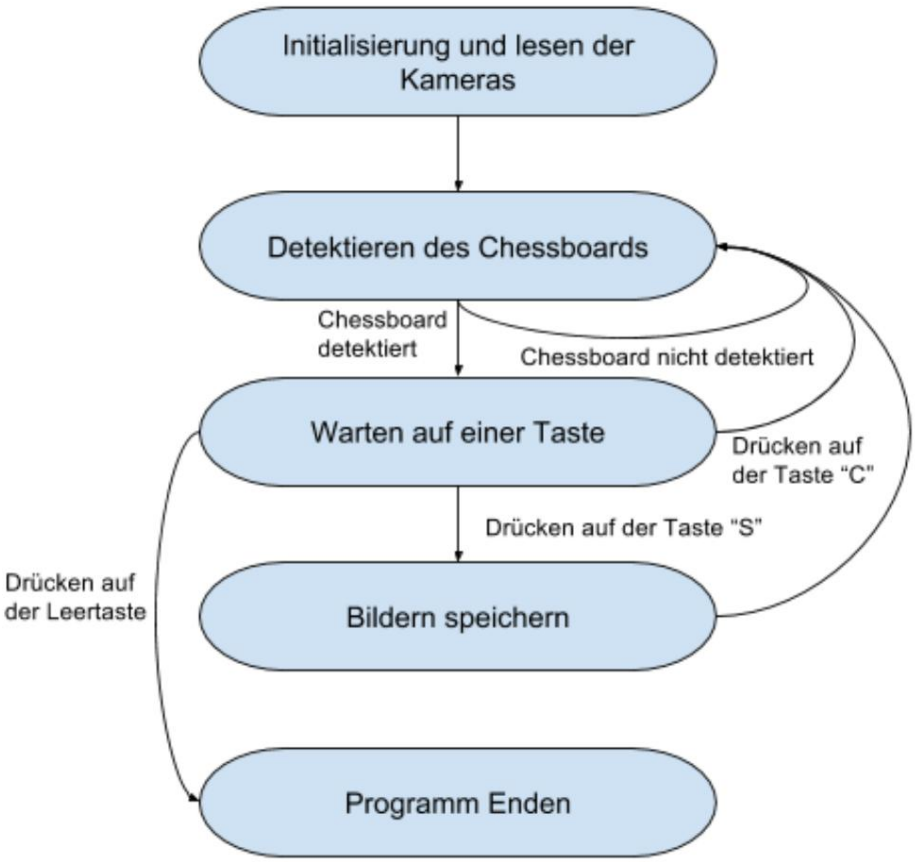


*Diagram 1: How **"Take_images_for_calibration.py"** works*

1. Vectors for calibration

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

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

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

2. Capturing images for calibration

Once the program has recognized the position of the checkerboard 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, you must press the “s” (Save) button to save the images. If the opposite is the case, you can press the “c” (Cancel) button.

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

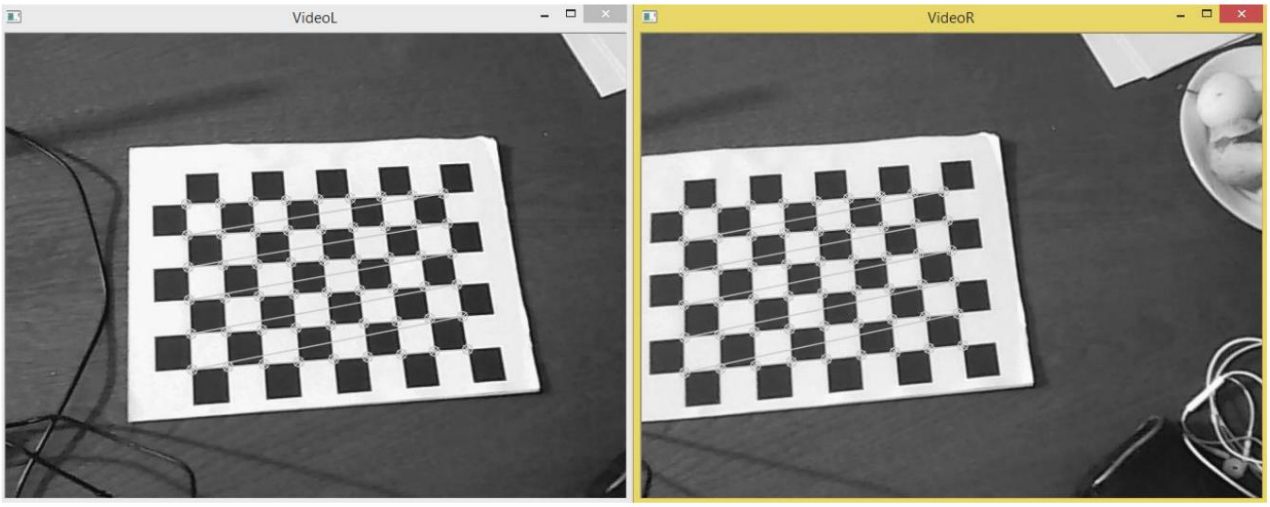


Figure 11: Recognition of the corners of a chessboard

D. How the program works
“Main_Stereo_Vision_Prog.py”

During initialization, the cameras are first calibrated individually to remove distortion. Then the stereo calibration is carried out (removing rotation, aligning the epipolar lines). The images are processed in an infinite loop and a disparity map is generated.

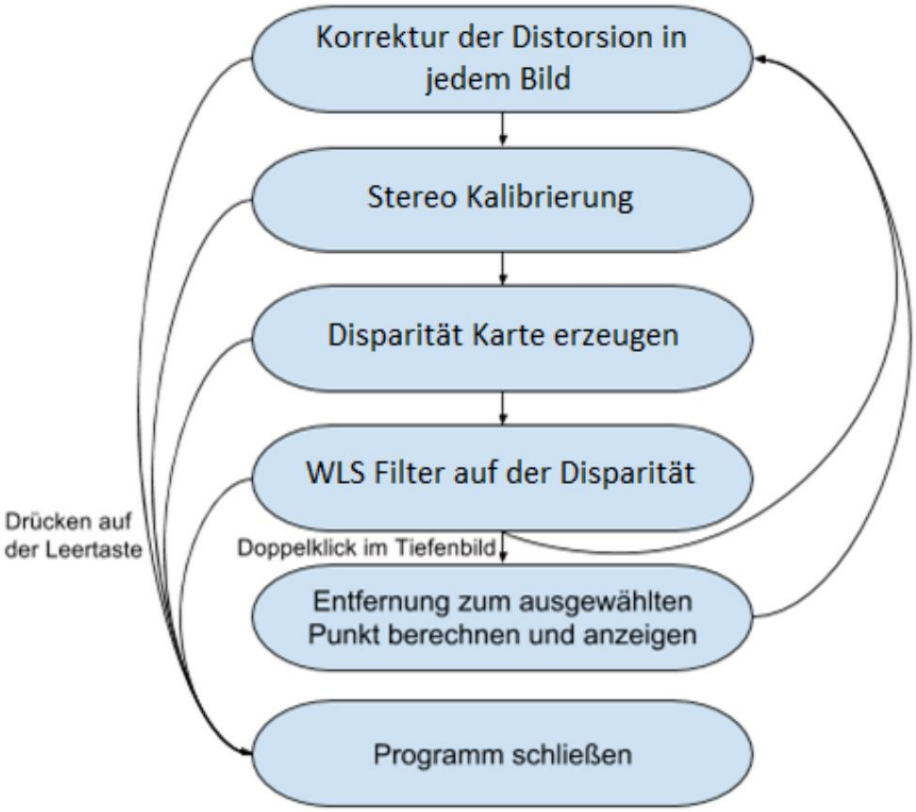


Diagram 2: How "Main_Stereo_Vision_Prog.py" works

1. Calibration of distortion

To correct 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” where the position of the checkerboard corners is stored in **Imgpoints** and **Objpoints .** The function *cv2.calibrateCamera()* is used to create new camera matrices (The camera matrix
describes the projection of a point in the 3D world in a 2D image), distortion
Coefficients, rotation and translation vectors for each camera which are later used to remove the distortion from each camera. To find the optimal
To get camera matrices for each camera the function
cv2.getOptimalNewCameraMatrix() is used (increasing precision).

```
#    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: Calibrating Distortion in Python

2. Calibration of the stereo camera

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

The *cv2.stereoRectify()* function allows the epipolar lines of the two cameras to be placed on the same plane. This transformation makes the work of the function that creates the disparity map easier, because then the block match only needs to be searched for in one dimension. This function also gives the essential matrix and the fundamental matrix that are used in the next function.

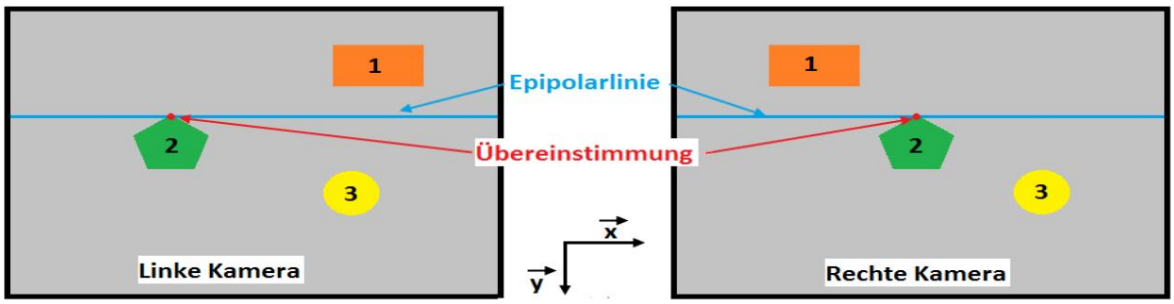


Figure 12: Principle of epipolar lines

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

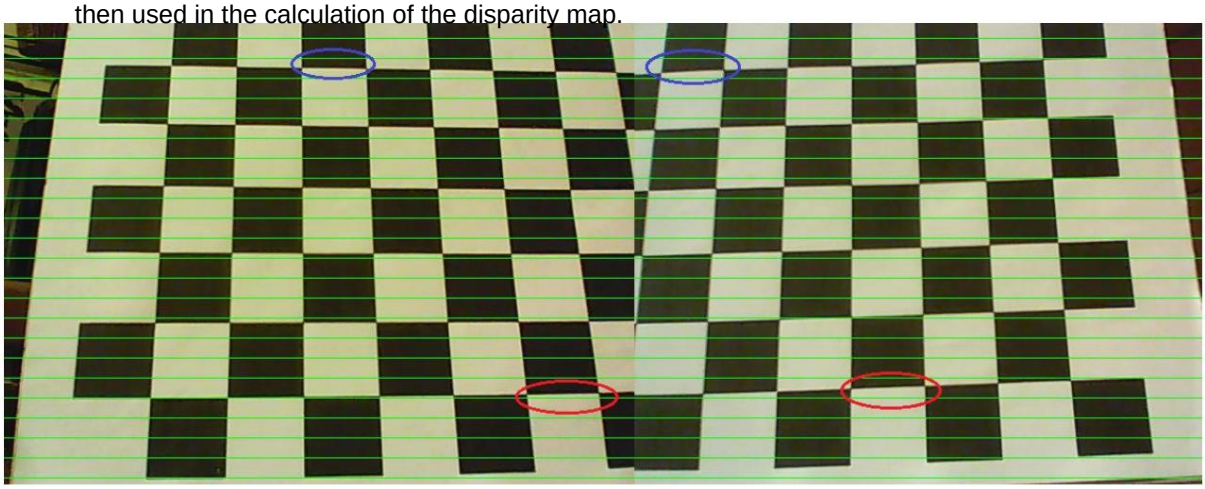


Figure 13: Without calibration

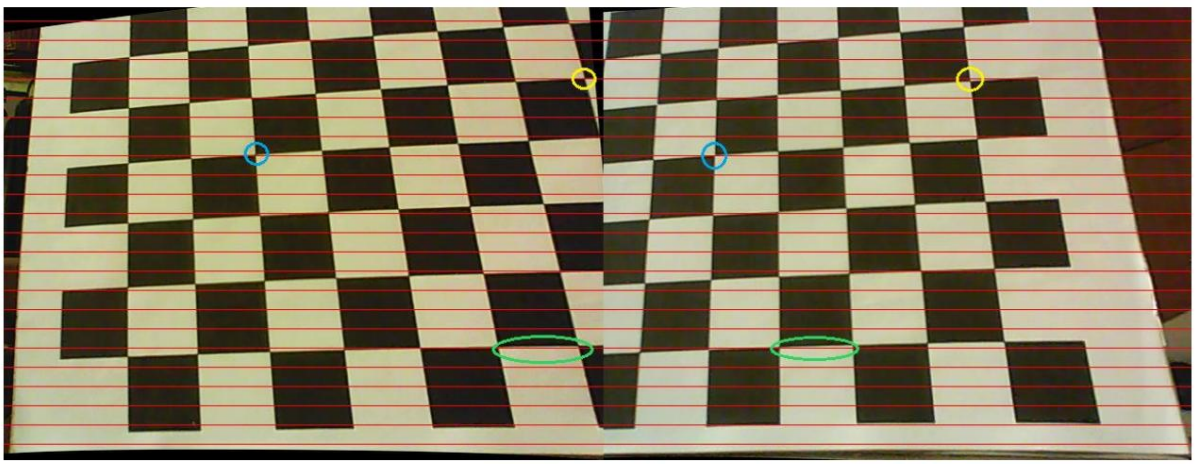


Figure 14: With calibration

3. Calculating the disparity map

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

How the Semi-Global Matching Algorithm works:

The following scene is presented to the stereo camera:

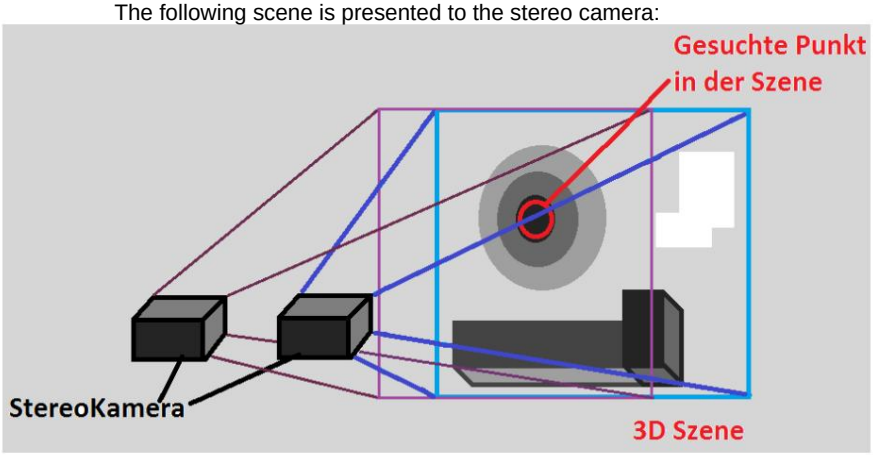


Figure 15: Example of a stereo camera observing a scene

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 of a reference image with the blocks of the match image. If the stereo calibration was done well, for example, a block of row four of the reference image must be compared with all the 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 us use the previous example with the blocks of the fourth row 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 of the base image with all other blocks (4,i) of the fourth row (same epipolar line) of the match image.

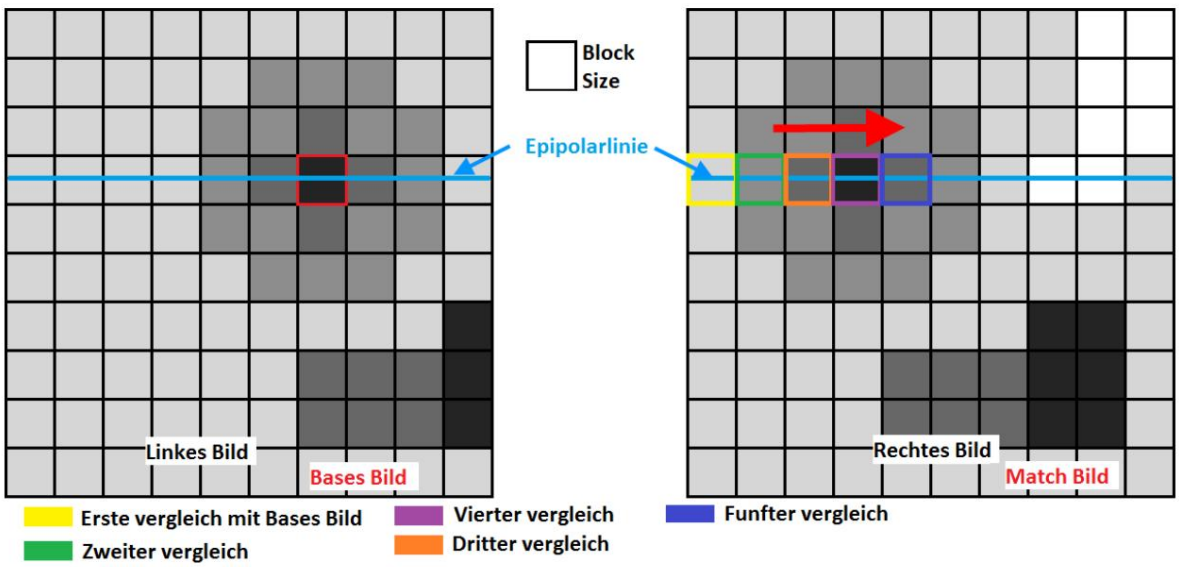


Figure 16: Matching blocks with StereoSGBM

Frédéric Uhrweiler
Stéphane Vujasinovic

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

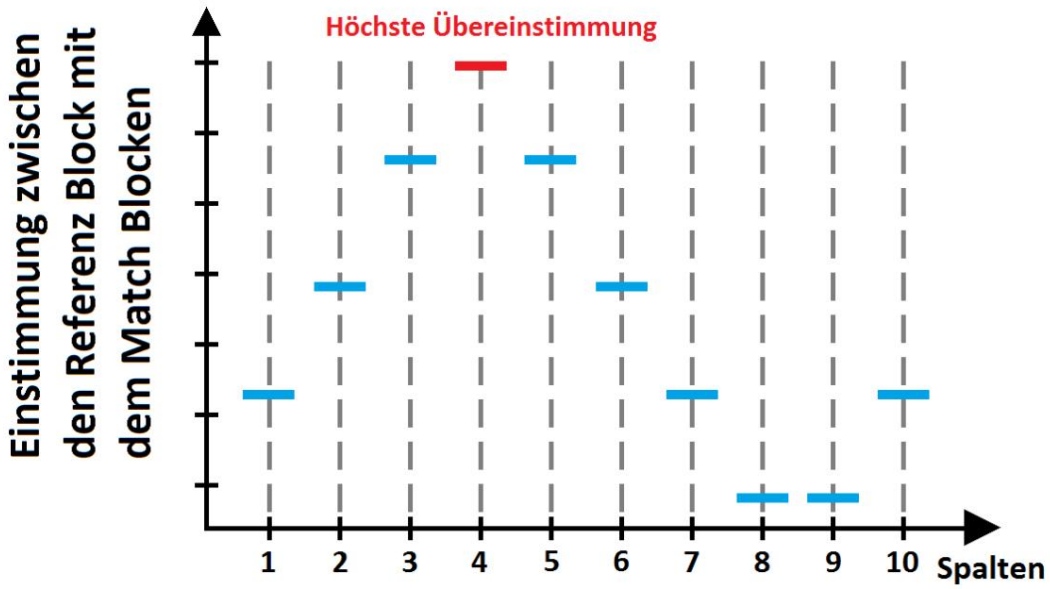


Figure 17: Detection of same blocks with StereoSGBM

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

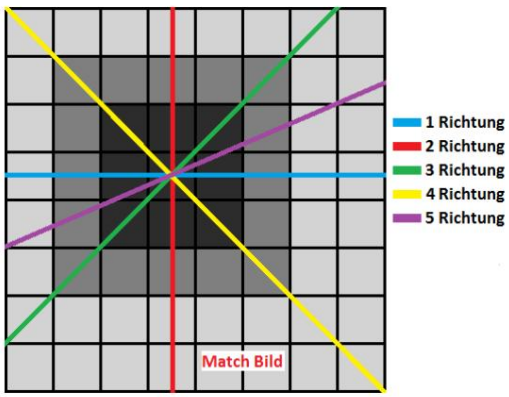


Figure 18: Example of the five directions of the StereoSGBM algorithm in OpenCV

To find the disparity, the coordinates of the match block are subtracted from 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 to calculate the disparity map, it is also possible to work with BGR images but this would require more time for the computer. The calculation of the map is done using a method of 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 defined 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

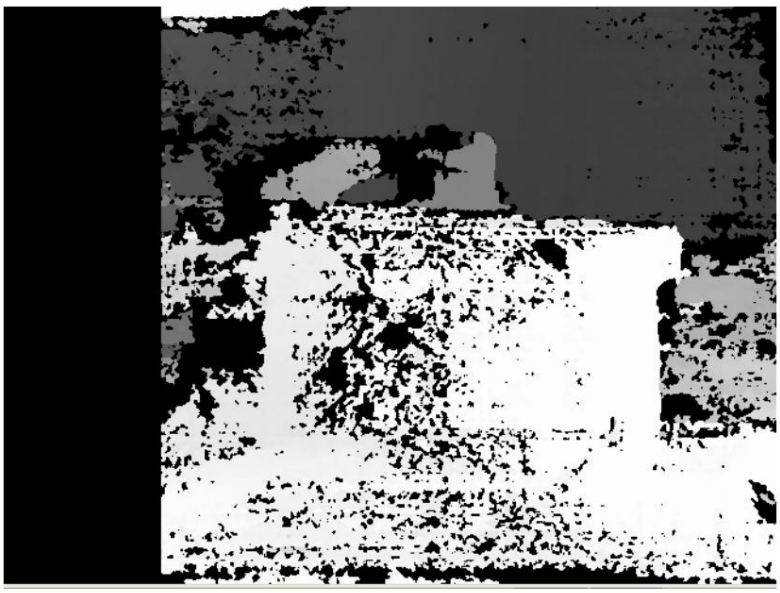


Figure 19: Result for the disparity map

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

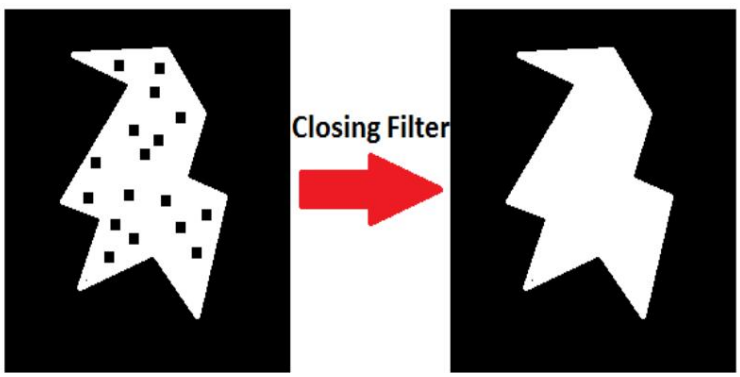


Figure 20: Example of a closing filter

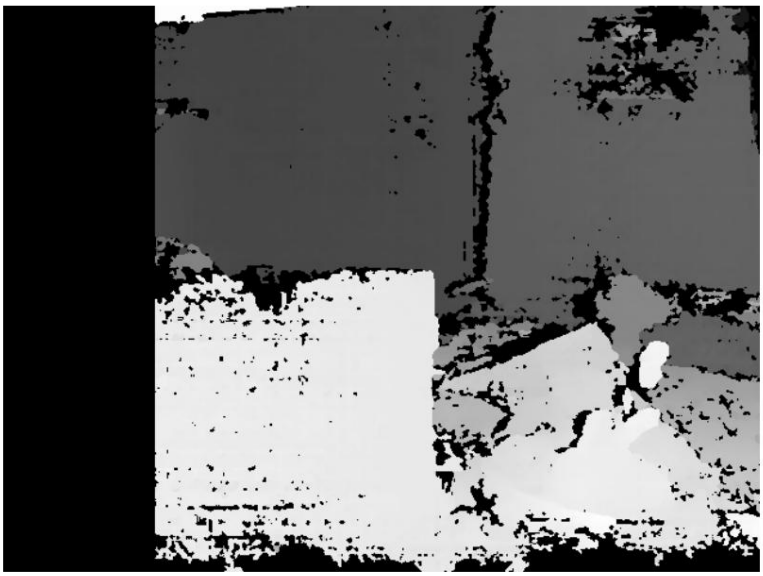


Figure 21: Result of a disparity map after a closing filter

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

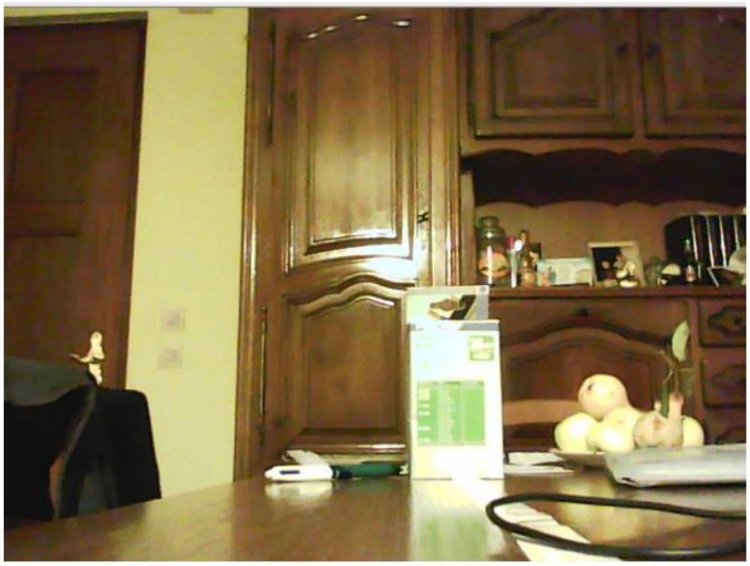


Figure 22: Normal scene seen by the left camera without rectification and calibration

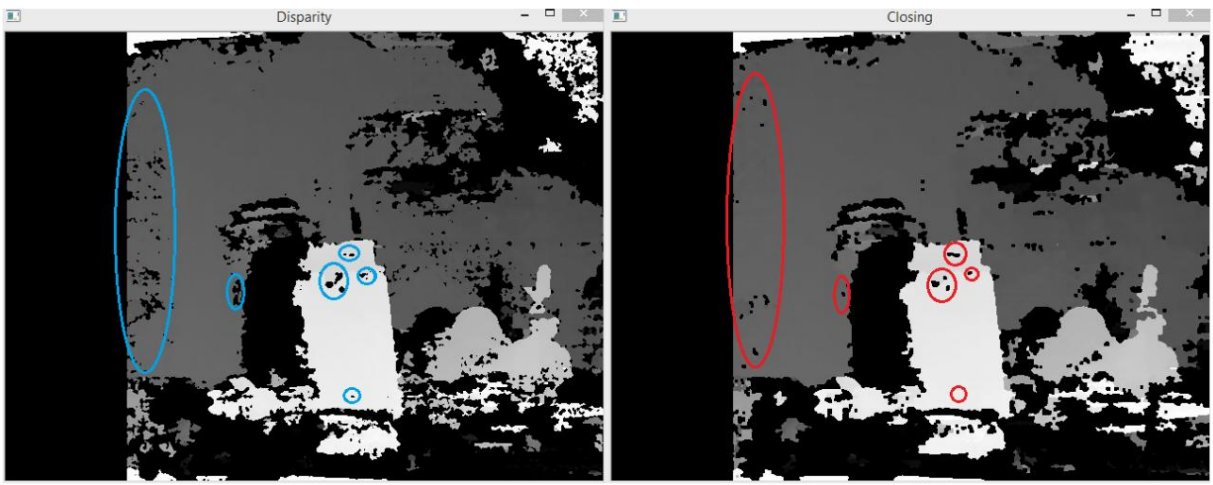


Figure 23: Disparity map of the upper scene without closing filter and with

4. Einsatz des WLS (Weighted Least Squares) Filters

The result is not bad but it is still very difficult to detect the edges of objects because of the noise, so a WLS filter is used. First, the parameters of the filter must be set 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
the disparity map attached to the shapes of the reference image, we set it to 80000 because we got better results with this value. Sigma describes how precise our filter needs to be at the edges of objects.

Another stereo object is created using *cv2.ximgproc.createRightMatcher()* and is based on the first one. These two instances are then used in the WLS filter to create a disparity map.

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

```
# 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: Creating a WLS filter in Python

To then 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: Implementing the WLS filter in Python

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

Figure 24: Ocean ColorMap

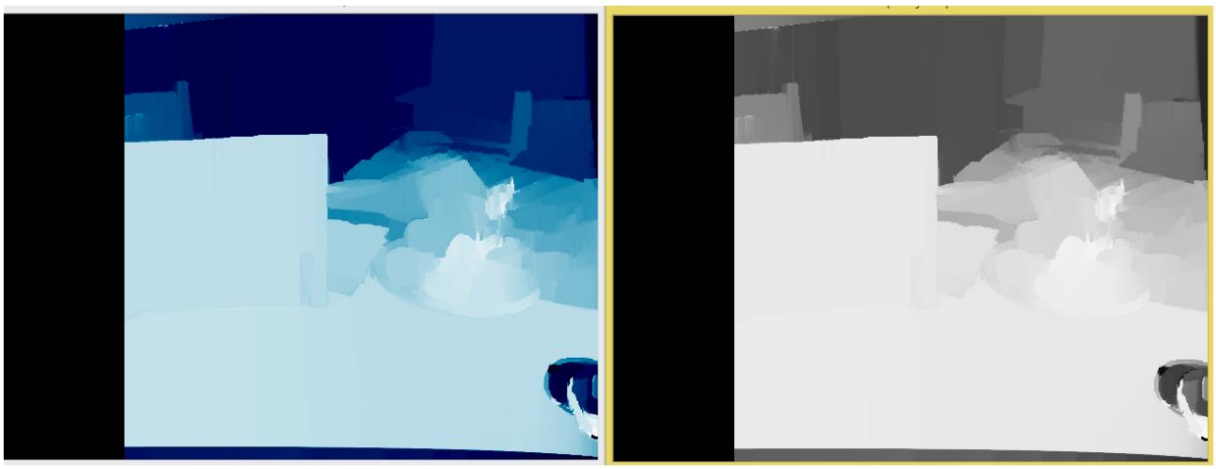


Figure 25: WLS filter on a disparity map with an Ocean Map Color and without

In this way we get an image that can show the edges 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).
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. These (x, y)
Coordinates are then used to get the disparity value from the disparity map and then measure the distance. The returned value is the average disparity of a 9x9 pixel matrix.

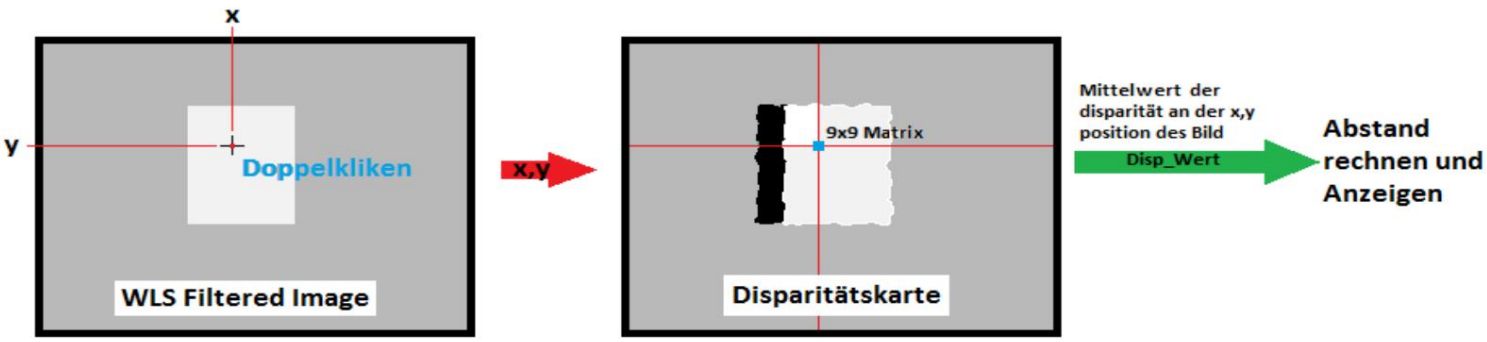
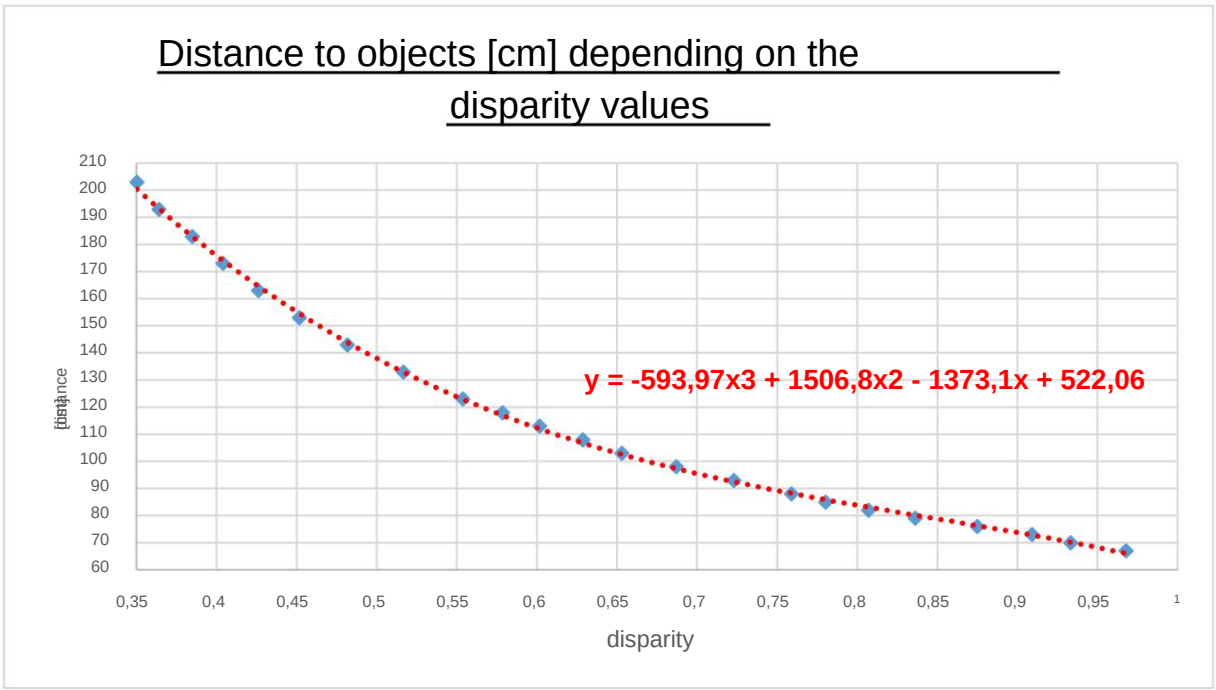


Figure 26: Calculation of the distance with the WLS filter and the disparity map

5. Measuring the distance

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



straight line equation into the Python program

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

*Code 12: Die Regression Formel in **„Main_Stereo_Vision_Prog.py“***

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

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

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

6. Possible improvements

Possible improvements for the program:

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

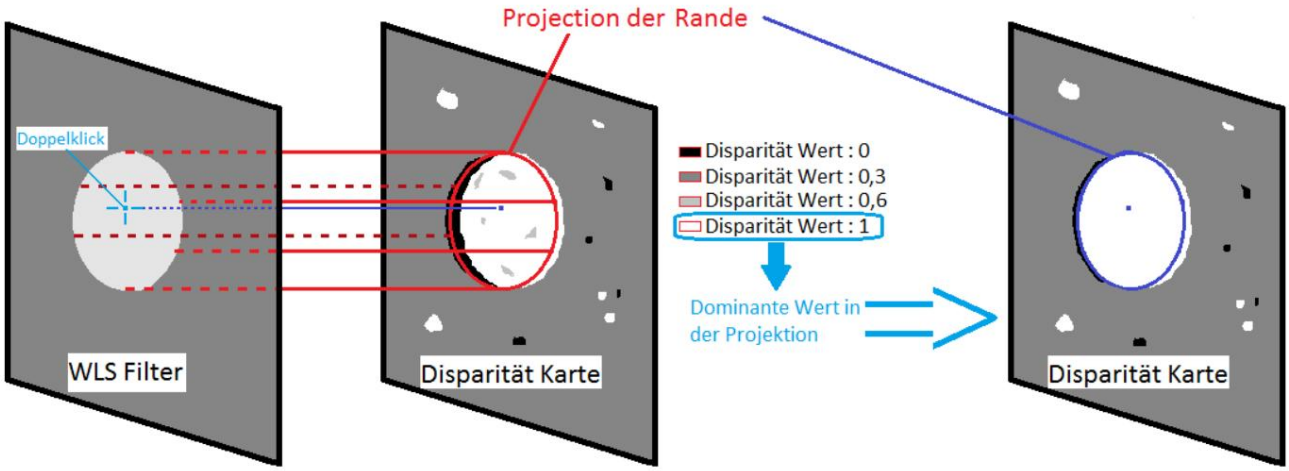


Figure 28:Possible improvement with the first proposal

- Applying a bilateral filter to the calibrated images before generating the disparity map, this way it would be possible to avoid using a WLS filter
  This needs to be checked, but the WLS filter is only used to clearly identify the edges of objects, perhaps there is a better way.

- To reduce the calculation time for generating the disparity map, the calibrated images should be reduced using the function *cv2.resize(cv2:INTER_AREA),* making sure that the values of the essential matrices, fundamental matrices, are also reduced proportionally.

- Generating a depth map could also be beneficial.

- A camera that is more stable than ours, where the rotation of the heads can be really prevented, in this way you would only need 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 allow us to get smoother
  to get pictures when the stereo camera is moving

5. Conclusion

A. Summary

This project work has allowed us to work with the Python language and learn more about the OpenCV library. We have had the opportunity to get familiar with the topic 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 distance using image processing, such as Time of Light (TOF) cameras, but these are expensive and 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, the developed program makes it possible to calculate a distance on the basis of a disparity map.

C. Outlook

In order to ensure that the calculated distance values always remain correct, a new system for the cameras should be developed that avoids any free movement of the cameras. This would mean that 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 his website rdmilligan.wordpress.com

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

Stereo Matching by Nassir Navab and Slides prepared by Christian Unger

Oreilly Learning OpenCV

6th Appendix

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

The Python programs can be found in the folder **“Python_Prog_Stereo_Vision” .**
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 do not have Python installed on your computer.