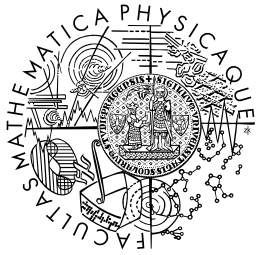


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jana Bátoryová

Visual Localization of an Object in 3D Space

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	prof. RNDr. Roman Barták, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

This thesis is dedicated to all students who were or are struggling with their thesis like me, and to their friends and families who support them. I encourage you not to give up, and continue.

I want to acknowledge and thank my supervisor for the topic and his patience during last two years. A great appreciation belongs to Michal Koutný as my consultant, since he has helped me a lot during the last year. A great appreciation belongs also to my dearest Dominik Smrž, since he was with me during the dark and bright days of writing this thesis. I thank my parents for never-ending support. They gave me the opportunity to study. To all my friends, thank you for your understanding and encouragement in many moments of crisis. Also, thanks to Martin Faltus for lending me a robot for the experiments.

Title: Visual Localization of an Object in 3D Space

Author: Jana Bátoryová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The purpose of this project is to propose and implement a system for object localization using a stereo vision – two cameras. The system computes the position of the cameras relatively to each other using a calibration pattern. Then a user selects an object to track. Different algorithms can be used for tracking. Both detection-based and sequence-based approaches can be used. When the object is found in the view of both cameras, the system estimates a position of the object in three-dimensional space using triangulation and displays the results live.

Keywords: 3D, localization, tracking, stereovision, opencv

Contents

Introduction

Living in the 21st century, we are witnessing a huge improvement of computers and their abilities. Since the beginning of the era of computers, people have been creating machines and programs that solve problems more efficiently than humans.

From the beginning, each human has two eyes, which provide stereo vision. Stereo vision allows us to estimate how far and where exactly an object in 3D space is.

Our goal is to propose a system that will have similar capabilities using a computer and two web cameras. The system can be used to monitor and track a movement of objects. For example, it can provide a trajectory of the quadcopter flight or of another autonomous robot. These obtained data can be used for additional analysis.

To achieve the above goal, we need to solve a problem of object detection, tracking, and stereo vision by providing image streams from two cameras. We focus on an overall process from the camera calibration, through object detection in both cameras, to an estimation of the object position in 3D space.

We set as our technical goals the following requirements:

- automatic calibration,
- choice of a tracker,
- easy way to add and test a new tracker,
- reproducible run - working with videos, in addition to live stream,
- saving results from calibration and localization for later use,
- displaying results live.

This thesis starts with an overview of the work done in the area of computer vision and object localization. Then the thesis covers three steps of the object localization. The first step of the localization is to obtain information about the cameras. The Chapter Calibration covers the process of obtaining parameters to describe the camera model by viewing a specific pattern. Enough images of the pattern provide enough information to get the parameters describing the camera. Furthermore, positions of the pattern, where both cameras see the pattern, can be used for stereo calibration. The stereo calibration provides information about the relative positions of cameras to each other. We will later use this relative position and the camera parameters in the localization process.

The second step is to track an object. Firstly, a user marks an object in camera's views. Then the object position estimation is provided by a tracker algorithm. Many different approaches may be used for tracking. In this thesis, we distinguish two groups of tracking algorithms: those working on with a sequence of images (sequence-based) and those working only with one image (detection-based). We describe several tracking algorithms from both groups. At the end of the chapter, we provide an empirical comparison of mentioned trackers.

The chapter Localization covers the steps needed to estimate a position of the object in three-dimensional space. At the beginning of the chapter, we choose and describe a coordinate system, using results from stereo calibration. Then we derive projection matrices, which transform from the world coordinates to the image coordinates. As the next step, we introduce simple triangulation,

which estimates the position of the object in 3D by providing calibration results, projection matrices and the coordinates of the object in both camera images.

At the end of the thesis, we provide results of evaluation of the described system. In the first set of experiments, we omit the trackers, and we evaluate the accuracy of calibration and localization. The second set of experiments presents results from the tests of the program as a whole, from calibration through tracking until localization.

In the Appendix, we include user documentation for running the program and programmer documentation for a better understanding of the code.

1. Related works

The problem of retrieving an object position in 3D is well known and discussed in many papers. Applications can be found in many different areas, for example in sports in unclear situations, in the robotics for better navigation and also in augmented reality for more impressive effect.

1.1 Stereo vision systems

In the paper by Zheng, Chang, and Li [2010], the authors research stereo vision. The proposed system has a requirement of the parallel alignment of cameras both to each other and also parallel to the floor. The authors test the accuracy of the system and also the disparity of the results in different distances. They provide several experiments measuring these disparities moving an object in an environment with one-color background.

The paper by Black, Ellis, and Rosin [2002] tries to solve a problem of predicting the object position under full occlusion. The authors of the Yonemoto, Tsuruta, and Taniguchi [1998] solve a problem of correct localization of the object consisting of multiple parts. These parts of the object often fall apart when trying to include them in virtual reality (for example a snowman can be mapped as three balls apart of each other). The proposed system estimates parameters and fixes points for each part independently. Then the system uses correspondence of the points within the movements.

A system for the object tracking and localization in the parking house was introduced in the paper Ibisch, Houben, Michael, Kesten, Schuller, and AUDI AG [2015]. The proposed system uses several cameras that share parts of their views. For object detection, they use a method of the background subtraction. We use a similar method in our work and compare it with other methods. Because the system's primary use is in the parking house, they also propose a method to cope with the change of lighting caused by a car lights. Their primary goal is to predict possible car collisions and therefore to provide data for a warning system.

1.2 Use of the systems in sports

An excellent example of using a sytem for tracking an object in 3D space is to solve unclear situations in sports, where sight of the naked human eye is inconclusive. Over the last few years, one of the most famous systems is Hawk Eye (shortly described in Owens, Harris, and Stennett [2003]). The system provides access to the trajectory of a ball and can replay it to referees. Furthermore, even the most popular soccer competitions are experimenting with the system for detecting if the ball crossed the goal line or not. This system needs an expensive setup equipped with high-speed cameras and the software itself is of high price. On the contrary, we propose a low cost alternative using web cameras.

1.3 Use of the systems in robotics

The principles described in previous sections have an extensive usage in robotics: for better navigation, object manipulating and so on. For example, in the robotic competition RoboCup, competitors of the Soccer, developed many systems for ball tracking. Robots competing in this category are usually equipped with an omnidirectional camera. This camera is pointing vertically up, where the image reflects in a mirror. This principle provides a full 360° image of the environment provided by only one camera. In this case, it is possible to track an object by a single camera, since additional information about the ball (such as color and size) is provided in advance. On the other hand, such system is not static, the background changes with every movement of the robot. Nevertheless, usage of additional camera may significantly improve the precision. For more information, we refer to the paper Käppeler, Höferlin, and Levi [2010].

1.4 Dove-eye

The idea of this work is based on its predecessor, Dove-eye presented in the paper by Barták, Koutný, and Obdrzálek [2016]. This project uses automatic calibration process to get the information about cameras and provides several ways to track an object, and the localization results are displayed live.

1.5 Our approach

Starting with the ideas used in Dove-eye (Barták, Koutný, and Obdrzálek [2016]), we developed a new implementation for this task. We improve the tracking process by providing many different trackers, which gives an opportunity to choose the one which is best suitable for given environment. We provide an easy way to correct the tracker by its reinitialization. Furthermore, no knowledge of the internals is needed to add a new tracker. In difference to some previously mentioned papers, we do not require special alignment of the cameras.

We also implement an algorithm similar to the one mentioned by Ibisch, Houben, Michael, Kesten, Schuller, and AUDI AG [2015] and others. As opposite to the paper by Käppeler, Höferlin, and Levi [2010] we do not use any previous information about the object tracked.

This thesis also considers a case of tracking multiple objects. It provides a way, to initialize tracking objects. The comparison of the trackers is provided including their speed, accuracy, but also ability to track multiple objects.

We conclude the thesis with several experiments, including experiments with an autonomous robot, to test suitability of the system for applications in robotics.

2. The proposed system

To be able to locate an object in 3D with no previous knowledge of the object (such as its size, color and so on), two views of the same object are needed. The views can be obtained by one moving camera or by multiple cameras. In this thesis, we take a closer look at the second approach.

We propose a system with two cameras, connected to a computer via USB. Two cameras provide enough information for object localization and make the project usable also on low-budget. The placement of the cameras is important, but no precise alignment is required. The cameras should share a significant part of the view (see an example of the setup in the Figure 2.1).

The program will be able to calibrate cameras to get their parameters. This calibration will be based on showing a specific pattern to both cameras. From this calibration, we obtain information for both cameras, but also their distance to each other and the rotation between them.

After this calibration, the user marks the objects in both views of the cameras. At this point, the tracker will start to estimate an object position in camera view based on the images. We provide several trackers, so the user can choose one to work with.

When we obtain information from calibration and the object position from both images, we will estimate the object position in 3D space by simple linear triangulation.

Each described step is crucial to get a system, which will be able to localize a point in 3D space. Each step is discussed separately in the following chapter.

Now we provide a short insight into used tools and the notations, which are used to describe our solution to the problem.

2.1 Tools

For the computer vision task, we use an *OpenCV* library. OpenCV is an open source computer vision library with many algorithms implemented (for example calibration, triangulation, and so on). We use trackers which are now available only in the *contribute* version. For more information and examples of usage visit their webpage[1].

To provide a comparison of trackers, we also include one tracker from *Dlib*[2] library. It is also open source and an interface for Python is provided. Since this library focus is on machine learning, it does not contain as many functions for computer vision as the OpenCV library yet.

2.2 Notations

The following section describes some procedures using mathematical notion. To avoid ambiguity we provide the overview of the used notation here.

[1] `https://opencv.org/`

[2] `http://dlib.net/`

Figure 2.1: Example of camera setup

Vector

A word *vector* denotes a column vector in a shape of $n \times 1$.

Block matrix

A matrix operation $W = (A|B)$, where A is a matrix $m \times n$ and B is a matrix $m \times p$, creates matrix W, where the first n columns are the entries from matrix A and the last p columns consist of the columns of matrix B. Example:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, B = \begin{pmatrix} 5 \\ 6 \end{pmatrix}, (A|B) = \begin{pmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{pmatrix}$$

Homogeneous coordinates

In computer vision, homogeneous coordinates are used frequently instead of Cartesian coordinates. Cartesian coordinates are the most common coordinate system. Homogeneous coordinates have one element added. This element is a scaling factor. As an example, vector $(2x, 2y, 2z, 2)^T$ represents same point as $(x, y, z, 1)^T$ in homogeneous coordinates. The same point is equal to $(x, y, z)^T$ in Cartesian coordinates. Unlike the Cartesian coordinates, a single point can be represented by infinitely many homogeneous coordinates.

Similarly it works for coordinates in any n-dimensional space. In conversion from Cartesian to homogeneous coordinates we simply add a new element at the end equal to one. In the opposite direction, we divide first $n - 1$ values by the nth.

In homogeneous coordinates the origin is left out. Homogeneous coordinates provide us also a way how to represent a point in the infinity by finite coordinates. These points in the infinity are represented with the scale factor equal to 0.

This coordinates representation is tightly bounded to a projective plane, on which the mathematical theory behind is based. For us, it is just important to know how to convert them to Cartesian coordinates and that algorithms for computer vision often use them.

3. Calibration

Our goal is to create a system which does not need any prior information about the position of cameras. Since we do not know how far away the cameras are, nor angles between them, we firstly use calibration to obtain this information.

By calibrating cameras with a well describable pattern, we can obtain information about the single camera, such as what distortion its lens cause, or how the world points are projected to an image plane of the camera.

After discovering these parameters for both cameras, we will continue on stereo calibration. Stereo calibration will provide us information about their position in space relative to each other.

For these calibration processes, we use algorithms implemented in OpenCV. Therefore, we provide only a short overview of the process, and we describe only results obtained from calibration essential to us.

3.1 Intrinsic parameters

Each camera type is different. Moreover, each camera of a specific type is different due to a manufacturing process. Therefore we introduce intrinsic camera parameters, which help us to model the camera precisely.

Intrinsic camera parameters define a transformation between the world coordinates and the coordinates within an image plane (in pixels). Physical attributes of the camera influence this transformation.

These parameters include focal length, a position of the principal point and distortion coefficients. All these parameters are needed to get a correct transformation between the point in the space and the point at the image plane. Sometimes even more parameters are used for a better description of the model of the camera.

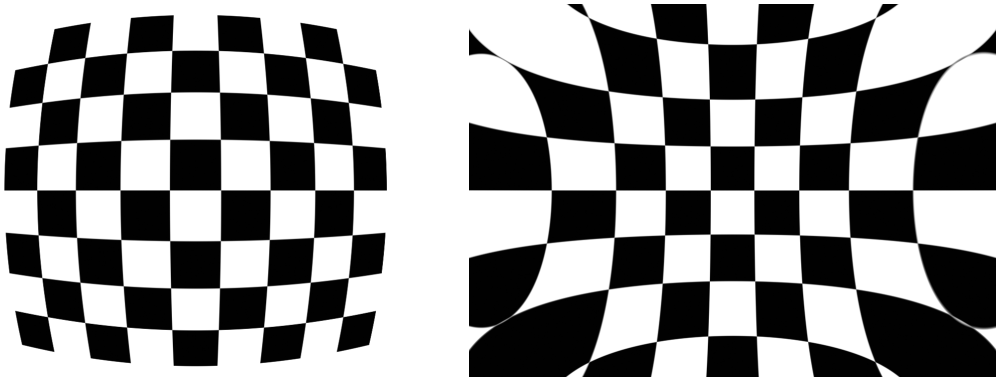
3.1.1 Camera matrix

Camera matrix will provide us a transformation from world coordinates (with the origin in the camera) to image plane coordinates (seen in the image taken from the camera). If the coordinates in 3D are available, we can obtain 2D coordinates by simple multiplication by camera matrix. After multiplication we obtain 2D homogeneous coordinates.

As the next step, we describe the camera matrix obtained by OpenCV calibration procedure. In the OpenCV implementation the camera matrix is 3×3 matrix of the following format:

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Where f_x, f_y denote focal lengths expressed in pixel units. It is usual to introduce two of them, separately for both axes, since pixels usually are not perfect squares but rather rectangles. Therefore the same focal length, has different length in pixel units over a given axis.



(a) Effect of positive radial distortion (b) Effect of negative radial distortion

Figure 3.1: Effects of lens distortion on the image of the chessboard

We refer the ray of the view of the camera as the principal ray. The point where this ray intersects the image plane is called principal point[1]. Parameters c_x and c_y define coordinates of the principal point in the image plane. It should be the center of the image, but assembling process of the camera might cause a small displacement.

3.1.2 Distortion coefficients

Cameras are equipped with lenses to obtain sharp image instead of blurry. The lens may causes various distortions. Fish-Eye lenses are known for their distortion. Even web camera lenses have distortion, but not as visible as the camera with a fish-eye lens. It is important to correct these distortions.

Two distortion types cause a significant effect on the image. The first one is the radial distortion, creating barrel effect and the second one tangential distortion.

The radial distortion is caused by the camera lens (the effect of radial distortion is displayed in the image 3.1). It can usually be described by three parameters. Highly distorted images (like from fish-eye) often need more parameters. Since our system use web cameras, we will use only three parameters.

In the ideal camera, the lens would be placed parallel to the chip. Since such precision is not possible, due to an assembling process, tangential distortion arises. For this distortion, we use two parameters.

We describe both distortion effects by five parameters. More about the meaning of the parameters could be found in the book by Bradski and Kaehler [2008].

These nine parameters (four for camera matrix and five for distortion coefficients) describe the camera model. They may be used for example for projecting a point from 3D space to image coordinates in the camera (OpenCV function `projectPoints`). We will use them to get better results for stereo calibration and for localization, since they provide us a more accurate model of the camera.

[1]For more information visit `https://en.wikipedia.org/wiki/Pinhole_camera_model#The_geometry_and_mathematics_of_the_pinhole_camera`

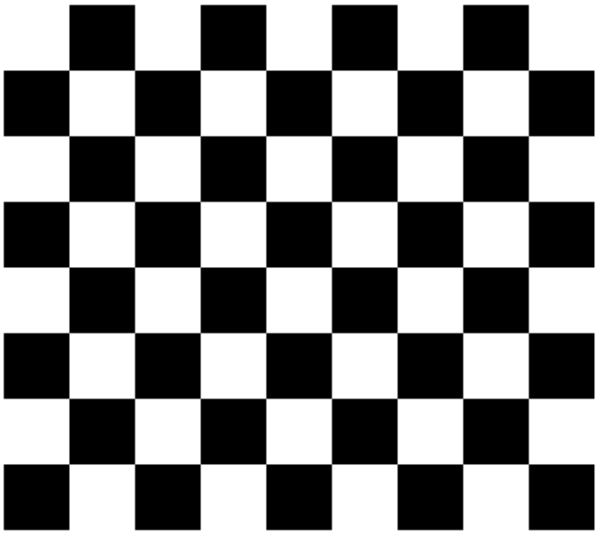


Figure 3.2: Example of 7×8 chessboard used for the calibration

3.2 Stereo calibration

After performing a calibration of both cameras separately, we also need information about their relative position to each other. By the relative position, we understand translation from the first camera to the second and the rotation of the camera (among three axes). Using this information, we can transform a view of the one camera, by moving it by translation vector and rotating accordingly. Therefore, the position of the second camera can be described by six parameters, three as an angle around each axis to rotate and three for translation vector in respect to the first camera.

Stereo calibration routine in OpenCV can also perform mono calibration for both cameras, but we pre-calibrate cameras on itself for better precision and better convergence of the algorithm.

Stereo calibration routine will provide more information. For the goals of this thesis, only rotation matrix and translation vector are important.

3.3 Calibration process

In previous sections we shortly discussed theoretical background behind the calibration. In this section we will focus on the implementation.

In OpenCV a chessboard is commonly used, since it is a planar object (easy to reproduce by printing) and well described. OpenCV also provides methods for automatic finding of the chessboard in the picture, so no human intervention is needed to select the chessboard corners from the image (an example of the chessboard is provided in a Figure 3.2). Therefore, we also use the chessboard pattern for calibration.

3.3.1 How many images?

Now we know that, we can use a chessboard to calibrate the cameras. However, the question of how many images are needed for the calibration remains. We discuss only the images where the full chessboard is found. For enough information from each image, we need a chessboard with at least 3×3 inner corners. It is better to have a chessboard with even, and odd dimension (for example $7\times$ 8 inner corners) since it has only one symmetry axis, and the pose of the object can be detected correctly. It is important that the chessboard indeed has squares, not rectangles (so it is better to check it after printing). A bigger chessboard is easier to recognize, so we recommend at least format A4.

We need only one image for computing the distortion coefficients. However, for the camera matrix, we need at least two images. For stereo calibration, we need at least one pair of images of the chessboard.

We know the minimum number of images needed for calibration. On the other hand, we use more images for the robustness of the algorithm. We need a "rich" set of views. Therefore it is important to move the chessboard between snapshots. With more provided information to the algorithm, it compensates the errors in the measurements (like a wrongly detected chessboard in one of the images). Therefore also the computation time increases. Since it is enough for a given setup of the cameras to perform a calibration only once and then use results from it again, we recommend to do the calibration on more images and wait a little bit longer for the results.

4. Tracker

We consider a tracker to be an algorithm for detection a position of an object in an image. We present a few tested trackers and their results in fulfilling this task. Firstly, we provide a short description of a simple straightforward tracker and then we describe more complicated trackers.

Tracker returns an object position in the image. We differentiate between two types of tracking algorithms: *detection-based algorithms* and *sequence-based algorithms*. Detection based algorithms detect an object on each image separately. On the other hand, sequence-based algorithms obtain, store and process information from a sequence of the past images and use it for more accurate tracking, while requiring same or even less computation time compared to the detection-based algorithms. For this task, we test a few trackers in both categories. The evaluation of the trackers is at the end of this chapter.

4.1 Detection based algorithms

We denote all trackers, which detect an object on each image separately as detection-based algorithms. These algorithms use only information obtained from the current image and the information from the image used to initialize the tracker.

4.1.1 Simple Background Tracker

Simple Background Tracker takes a photo of the background at the beginning and we denote it a *pattern*. In order to detect an object in an *image*, a comparison of the *image* and the *pattern* is done. We make this comparison by taking a sum of an absolute difference for each color channel (Red, Green, Blue) in the images for each pixel.

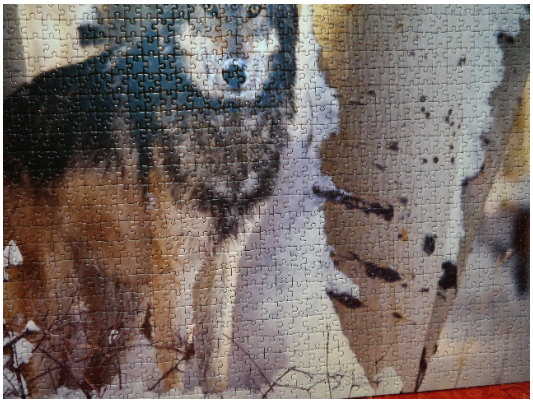
As a result, we get a mask where higher values mean bigger difference between the colors of the *pattern* and the *image* at given pixel. We assume it is caused by an object in front of the camera.

One may expect to get the mask by *thresholding* (see the results in Figure 4.1d). There is still a lot of noise. The noise is in the form of small dots and lines. Therefore we use *blurring* to remove the noise. We *threshold* it again to get the binary mask (see the results in Figure 4.1f). At this point, we will find a contour with the biggest area using OpenCV library[1]. The center point of the bounding box of this contour will be our estimation of the position of our object in the image. The whole process is illustrated in the Figure 4.1.

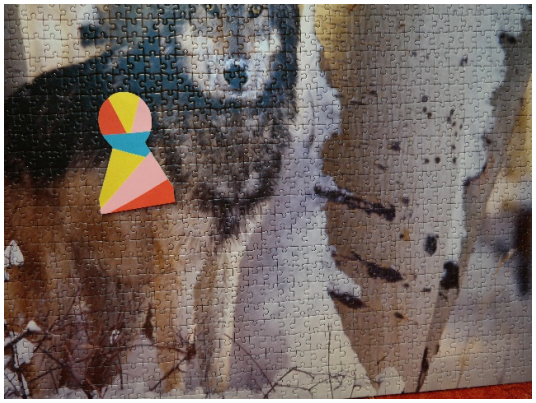
It is shown in the image that lighting slightly changed between the background photography and the object photography (at the edge of the puzzle pieces). It causes noise which we reduce by thresholding and blurring the image.

As an advantage of this algorithm, we consider its simplicity and straightforward implementation. Furthermore, with a static background with only one object moving it can reliably track an object without any information about it.

[1] used OpenCV functions: `cv2.findCountours` and `cv2.countourArea`

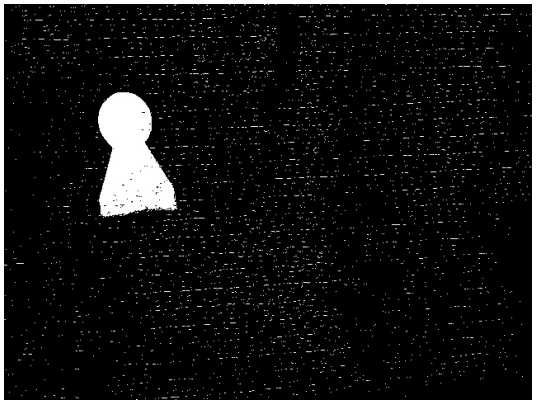


(a) Background

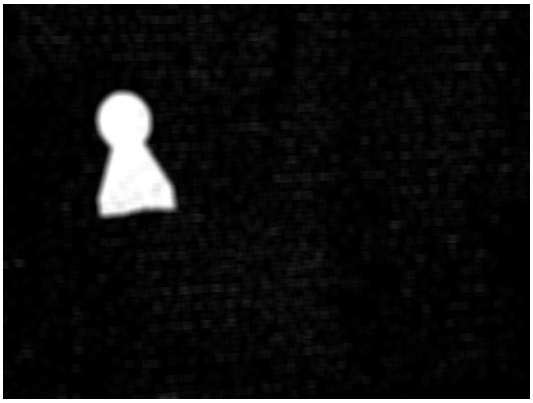


(b) Object

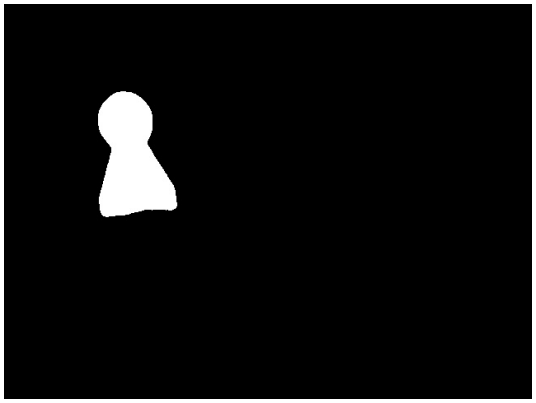
(c) Sum of diffs in each channel (RGB)



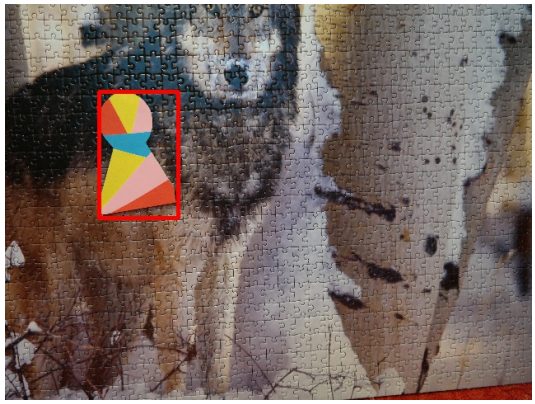
(d) Thresholded



(e) Blurred



(f) Thresholded



(g) Bounding box of the largest contour found.

Figure 4.1: Process of the simple background tracker

On the contrary, it cannot recover even from a little light changes or camera movement. Also, an object moved by a human, for example, cannot be tracked reliably since the tracker recognizes the hand as a moving object too.

4.1.2 HSV tracker

HSV tracker uses the idea of tracking an object by its color. Given an input object described by a bounding box, we find the average color within the bounding box. On a position request, we return a center of the largest area with the color of the object.

We choose the color coding via HSV (Hue, Saturation, Value) because unlike the RGB (Red, Green, Blue) coding it can represent the color as hue value, not three values of mixed colors. The approach of HSV color coding preserves one value – hue value, even though the color is lighter or darker (like shadows in the image). On the other hand, shadows may cause a difference in all three components in the RGB coding. Therefore an object description can be simplified to a single value (hue).

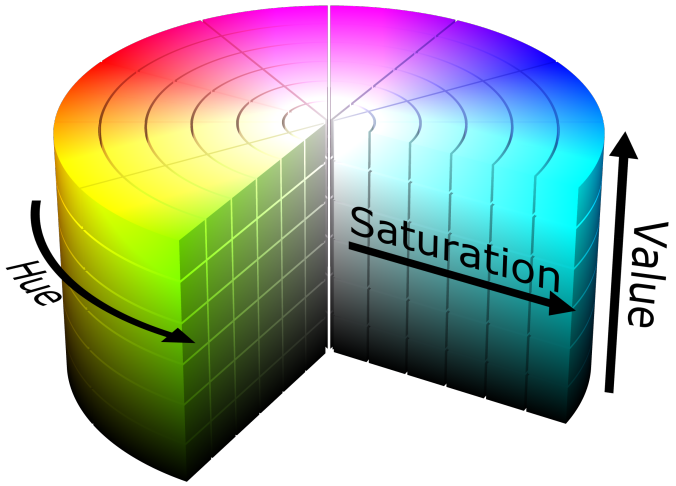
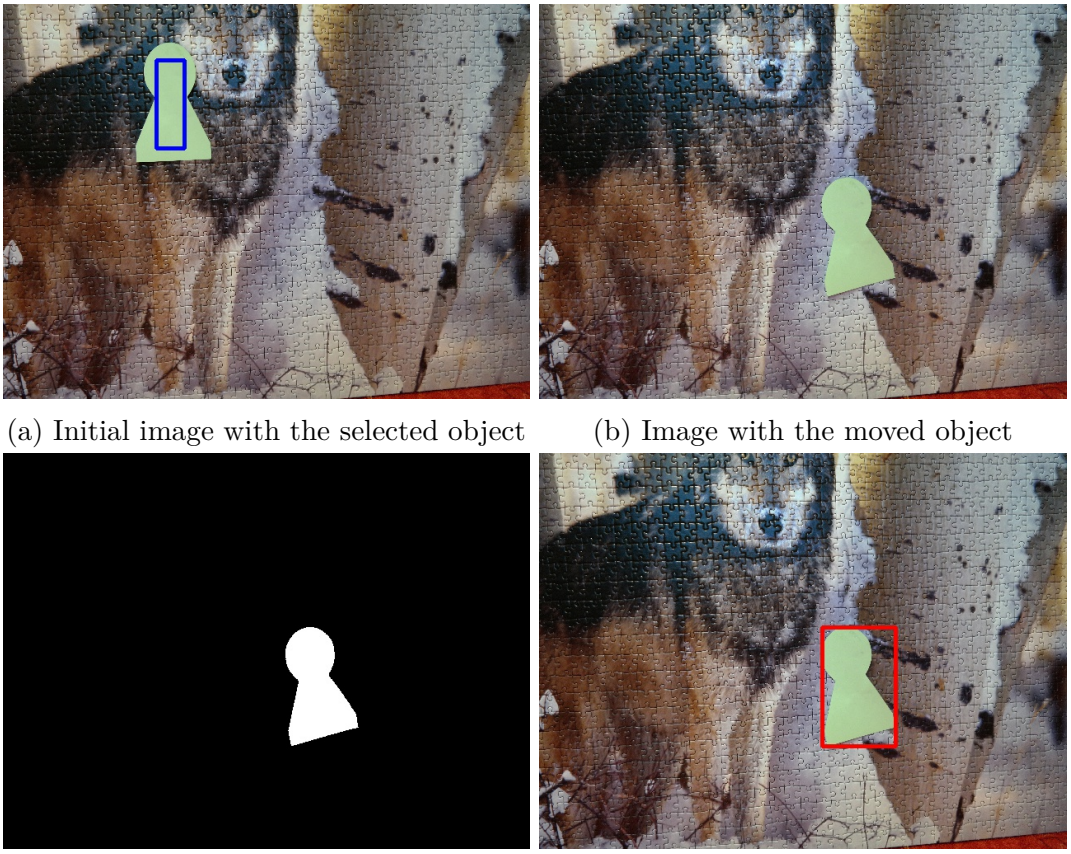


Figure 4.2: "HSV cylinder" by SharkD is licensed under CC BY 3.0 [2]

We describe the algorithm in a few steps. First, we convert the template image (bounding box) from the RGB color space to HSV. Then we calculate the average color in the template. When we look for the object in a image, we create a binary mask by the hue value of the each pixel. The pixel value in the mask is equal to one, if its color is in allowed range from the color detected for original object, otherwise it is equal to zero. Now, the tracker similarly as the Simple background tracker, finds the biggest area in the mask, which is equal to one. The tracker estimates the center of this area as a position of the object.

We take a closer look on finding an average color (i.e hue value) in the template. Since the coding of hue part is placed in the circle, it is not enough to take a commonly used average. It would cause that image full of warm red (the hue value of this color is circa equal to 15) and cool red (the hue is circa 345) would

[2]source: `https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png`



(a) Initial image with the selected object

(b) Image with the moved object

(c) Mask created by looking for similar colors

(d) Bounding box of the largest contour found.

Figure 4.3: Process of the HSV tracker

average to mid cyan (hue 180), instead of red (hue 0). To get a more reasonable average, we take the hue value of each pixel as a unit vector (the hue value is encoded as the angle between given vector and vector $(1, 0)$). We sum these vectors and get a vector (x, y). We find the corresponding angle for this vector using *atan*2 function. The following formulas describe the process of getting the average angle.

$$x = \sum_{\alpha} \cos \alpha$$
$$y = \sum_{\alpha} \sin \alpha$$
$$\alpha_{avg} = \text{atan2}(y, x)$$

The use of this algorithm for one-colored object is displayed in the Figure 4.3. We consider as a disadvantage that no other object of the same color can be placed in the view of the camera.

4.1.3 Pattern matching

The pattern (or template) matching algorithm slides over the input image and compares the template with a patch in the input image. By *patch* we understand an region in the image with the same size as the template (displayed in the Figure

4.4).

For a comparison between the template and the patch we look for a function which tells us how different the subimages are. Such a function is usually called a loss function. We use the function R based on square distance as our loss function. More precisely, we take the sum of square distances of all pairs of corresponding pixels in the template and the patch. Therefore, we compute R as

$$R(x, y) = \sum_{x',y'} ||T(x', y') - I(x + x', y + y')||^2$$

where x' and y' denote points from the neighbourhood of the x, y (bounded by template size). T denotes our pattern and I denotes the image.

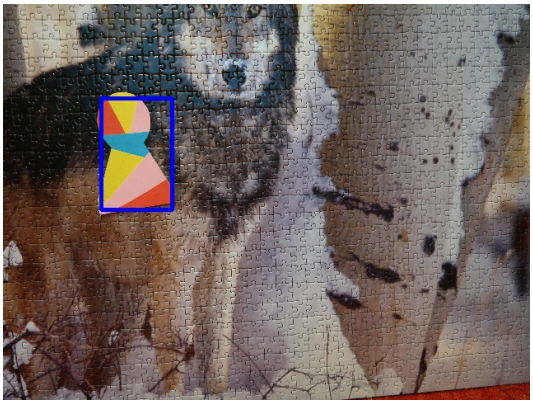
Figure 4.4: Naming convention showed on an example

The Pattern Matching tracker computes the loss function for the patch defined by its top left corner. Each possible position of the patch is computed. From the computed value for each pixel the pixel with the lowest value (i.e. shortest distance) is our estimation of the position of the object.

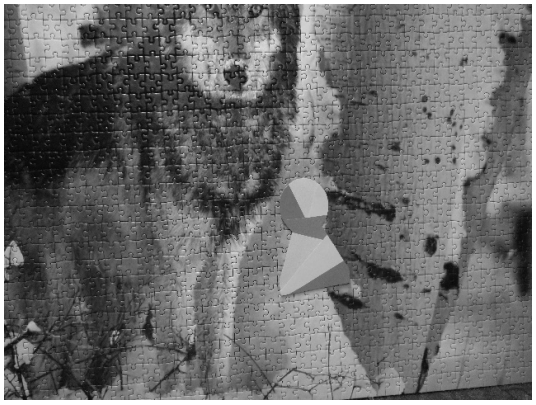
Finally, we choose an appropriate representation for the pixels (i.e. what precisely $T(x, y)$ and $I(x, y)$ stand for). In general, we can choose any reasonable vector (such as a tuple of RGB channels). We choose to use a standard OpenCV conversion to grayscale[3] ($T(x, y)$ and $I(x, y)$ thus give an intensity after such conversion).

Because this algorithm works with grayscale images, much information is lost during conversion. This disadvantage summed up with no ability to recognize rotated or slightly changed objects results in unsatisfying tracking results. An example of a correct match and also an incorrect one is displayed in the Figure 4.5. We can see the masks which display the value of the loss function. The pixel color represent the value of the loss function for the template and the patch which has top left corner in it. Lower values of the loss functions are displayed as darker points. Therefore, the black spots indicate patches which are similar to the pattern.

[3]see `https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html`

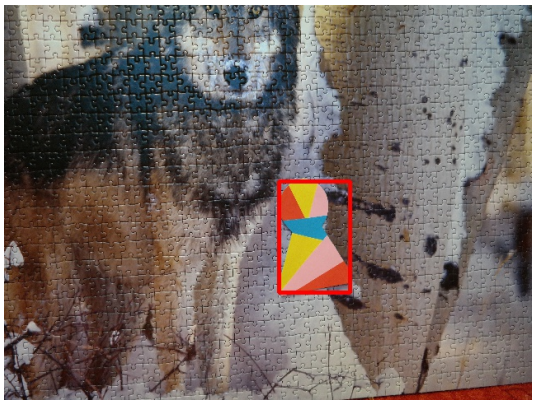


(a) Initial image with selected object



(b) Converting it to black white

(c) Mask created by applying metric



(d) Darkest point (lowest value) from mask is chosen

(e) Mask with darkest point on the left

(f) Incorrectly matched pattern

Figure 4.5: Process of the pattern matching

4.2 Sequence-based algorithms

We define a sequence-based algorithm as a tracking algorithm which uses information from a sequence of images.

Using the advantage of information from previous frames one could create not only more stable but also faster trackers. Furthermore, trackers of this class usually preserve identity, which means that also in the case of multiple moving objects in a frame it keeps tracking the original one.

Examples of the information we can obtain from the sequence of the images are:

- velocity – We can estimate the speed and direction of the movement from previous images. This information can reduce the searching area to smaller one and increase the speed of the algorithm.
- appearance – The object may rotate and change its shape or color. The tracker able to learn can be tolerant to such changes.

An algorithm using this information can cope with occlusion (one object covers another) – what detection algorithms are usually not able to do.

In the next sections, we will present a few trackers implemented in OpenCV. We provide a short overview of the trackers available.

Boosting tracker

Boosting tracker is based on online AdaBoost. It considers a bounding box as a positive sample and patches of background as negative ones. For a new image, the classifier runs on every pixel in the neighborhood of the previous location, scoring every pixel. The location with the highest score is chosen as a new location. The implementation in OpenCV is based on Grabner, Grabner, and Bischof [2006].

MIL tracker

The MIL is an abbreviation for Multiple Instance Learning. In comparison with the Boosting tracker, it does not keep only one image of the positive example but a set of images. The tracker considers a small neighborhood of the current position as possible positive examples. It helps the tracker to cope with the occlusion. OpenCV implementation is based on Babenko, Yang, and Belongie [2009].

KCF tracker

KCF stands for Kernelized Correlation Filters. Similarly, as the MIL tracker, it uses more positive samples and their large overlapping regions. Implementation provided by OpenCV is based on Henriques, Caseiro, Martins, and Batista [2012].

TLD tracker

Tracking, learning and detection, these are the three components of the TLD tracker. The tracker works frame to frame, and detection is run to correct the tracker if necessary. The learning process estimates detector errors and updates

it to avoid these errors in the future. The implementation in OpenCV is based on Kalal, Mikolajczyk, and Matas [2012].

MEDIANFLOW tracker

This tracker focuses on forward-backward error trying to minimize it. The implementation in OpenCV is based on Kalal, Mikolajczyk, and Matas [2010].

MOSSE tracker

MOSSE tracker is proposed for fast object tracking using correlation filter methods. Firstly, it performs Fast Fourier Transform of the template and the image. The convolution then is performed between the images and the result is inverted by Inverse Fast Fourier Transform (IFFT). The position is estimated by the highest value of the IFFT response. More about the tracker is available in the paper by Bolme, Beveridge, Draper, and Lui [2010].

OpenCV note

OpenCV-contribute implements all above-described sequence-based trackers. An overview of OpenCV trackers is provided by Mallick [2017].

Correlation tracker

We decided to include a tracker implemented in Dlib. The OpenCV trackers did not performed as we expected, because they often lost the tracking object. We looked for available alternative and decided to test Dlib tracker. This correlation tracker is based on the paper by Danelljan, Häger, Khan, and Felsberg [2014].

4.3 Trackers evaluation

In the previous section, we described many different trackers. We now provide their evaluation with the goal to find the best-performing trackers. The trackers differ in the approach to obtain information from the video stream, so it is important to test them under real conditions.

Firstly, we define the key capabilities which are essential for our project. Then we describe several experiments and their results. The experiments work with only one camera, since in this section we are not evaluating the localization but only trackers. At the end of this section, we provide a few pieces of advice how to choose the best tracker for a specific environment.

Key capabilities of the tracker

The trackers differ, therefore they can outperform others in some specific situations. To be able to compare two trackers, we provide a list of the most important properties in the project:

- accuracy,
- speed,

- ability to recover from occlusion,
- ability to track multiple objects.

The following paragraphs describe their importance and also the way we measure them in our experiments.

Accuracy

A good tracker has to be accurate. The tracker which is not accurate does not satisfy its purpose. Furthermore, the accuracy is important to obtain good localization results, since the inaccuracy of the estimated position of the object in the image cause inaccuracy in the estimated position in the 3D space.

It is difficult to say what accuracy is in the tracker case. Hence, we measure *inaccuracy* instead. Inaccuracy in our case is represented by the distance (in pixels) from the true position of the object and the one provided by the tracker. Higher values mean less accurate tracker.

To obtain robust results (not depending on one image), we take many images and compute the inaccuracy for each of them. Then the mean of this inaccuracy is the estimated inaccuracy for the tracker. The value is dependent on the resolution of the images. In our case we use images 640×480 px.

The last remaining problem is to get the right position of the object. One approach is to get the position by selecting the object by a human. This approach is very time consuming because only a one-minute long video contains approximately 1800 frames.

Instead of this approach, we select the best tracker. We do it by looking at the tracking performance on the video of all trackers. Then we choose a one tracker, which we think performed the best (most accurate) on the whole recording. This representative tracker has inaccuracy equal to 0, since it is compared to itself. The inaccuracy of the other trackers is computed as a distance between the selected object by this tracker and the representative tracker. This inaccuracy is then computed for each frame of the video and we compute sample mean and sample standard deviation to compare results of different trackers.

Speed/Computational time

Trackers differ not only in their accuracy but naturally also in the time needed to process an image to find the object. To measure a tracker's speed, we measure this time. We measure it as a number of ticks passed during calling tracker update (get the position on the given image). Then we take the number of the ticks per second and divide it by the number of ticks needed for tracker update. This way we obtain the number of the images that could be processed in one second by this tracker. Again, we do it on multiple images and then take a mean of these values.

For the number of images that could be processed in one second, we use a traditional shortcut FPS (frames per second). A higher number means a quicker tracker, therefore less time needed for each image.

This value is highly dependent on the hardware and the overall load of the system. Therefore, we use the value of FPS as a rough guess.

The results were computed on a system with Intel(R) Core(TM) i5-7300HQ CPU (2.50GHz, 2496 MHz, 4Core), 16GB RAM running Microsoft Windows 10 Enterprise.

Assessing the tracker speed is important. We consider a tracker too slow if it is not able to track live (>30 FPS). Even if the tracker has 30 FPS in the experiment, it might be too slow in the application, since the application has to perform many other tasks. Slow tracker may use too much computational power, therefore the application may not run smoothly.

Recovering from the occlusion

We refer to occlusion as a partial or full coverage of the tracked object by another object or leaving the view of the camera.

It might come handy to have a tracker which can recover from the occlusion during tracking. It is not easy to keep the object visible in both cameras. Therefore, we are interested also in the ability to recover from the occlusion.

Another useful ability is to detect if the object is lost. It is better to report object loss than providing incorrect results. We tested it together with recovering.

Again, we chose a representative trackers - the tracker which is able to detect object loss every time it happen and also can recover from occlusion. The Simple backgroundtracker performed very well in these tasks too, so we chose it as our representative.

We now describe how we decided if the tracker is able to report object lost. Always, when the representative tracker lost the object, we looked at the results of the tested tracker. Then we computed ratio between the number of reported object lost by tested tracker and the number of reported object lost by representative. If this ratio was higher than 90%, the tracker was able to detect object lost reliably. We include the ratio in our results.

The second capability to test is if the tracker can recover from occlusion. We took all the images that representative tracker was able to detect an object. Then we took the number of images, where the tested tracker was able to detect. We filtered out the results, which were further than given threshold from the representative tracker (tracking result was not correct). The ratio between the number of correct tracking results to the number of the images with the object (given by representative tracker) is our result. If this ratio is higher than 80%, then we say that the tracker is able to recover.

Tracking multiple objects

As an addition to our project, the program can localize multiple objects. When working with multiple objects, it is important to remember that not all trackers may be able to track correctly multiple objects at the same time. Furthermore, not many trackers can handle when the tracking objects occlude each other.

To decide if the tracker is able to track multiple objects or not is easily decidable by simple experiment. We created experiment with moving objects and observed the tracker's behavior. We observed, if the tracker was able to keep tracking both objects and if it remained to track same objects after their occlusion. An example of successful and unsuccessful tracker is displayed in the Figure 4.9.

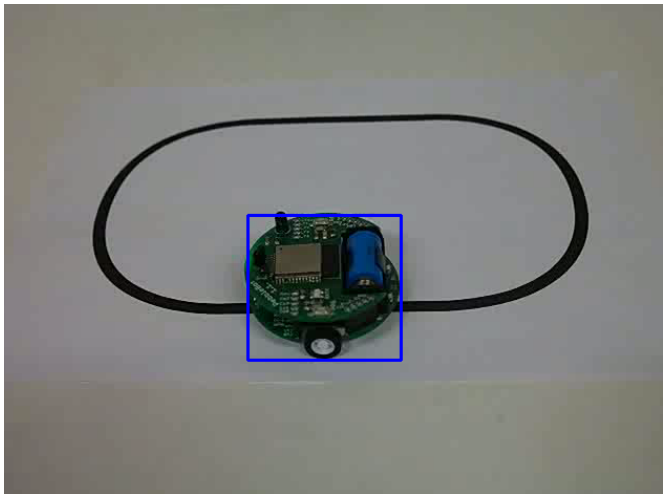


Figure 4.6: Selecting object to track on the video

With tracking multiple objects, the speed of the program may decrease. It is important to remember, that each object in each camera view has own tracker. Therefore the tracking part takes more time.

4.3.1 Experiments and results

We now describe the experiments used to measure tracker statistics. Since the main use of our project can be tracking robots and to record their trajectory, we decided to test the trackers under similar circumstances.

In some experiments we use an autonomous robot which has oval shape, 3 cm in height and 6 cm in diameter. This robot is able to follow a black line. We used an oval as the shape of the line (see the Figure 4.6).

In order to test the trackers under same conditions, we recorded a video of the robot. We then found the bounding box for the object at the start and initialized the trackers on the same video with same bounding box at the beginning. We made only a few exceptions because of requirements of some trackers which are mentioned in the corresponding experiments.

Speed and accuracy

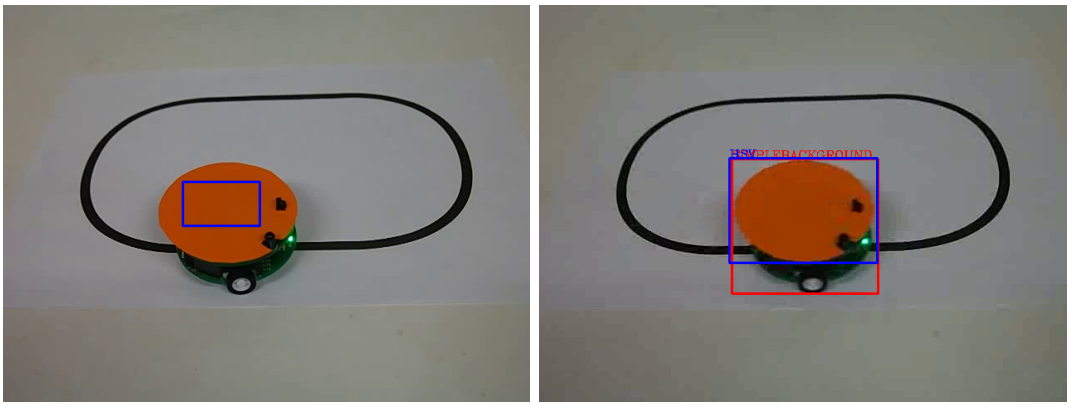
The goal of the first experiment is to estimate tracker speed and accuracy. We used a video of the robot moving along the oval (see the Figure 4.6). For the accuracy measurement we chose a Simple Background tracker as our representative, since it provided an excellent example of the correct tracking on this video. The representative tracker is always displayed as a red bounding box and the blue is used for the tested tracker.

Since the Simple Background trackers need empty background when initializing, we passed an image without a robot to it.

We consider this experiment as quite challenging for the trackers, since the robot is not only moving but also changing its appearance quite fast. In a second or two, its top platform is mirrored. On the other hand, the object does not change size and the background is very clear.

Tracker	Speed [FPS]	$\bar{X}$ [px]	σ_X [px]
Boosting	9.14	16.22	8.36
Correlation	107.40	32.97	16.47
HSV	507.31	38.50	10.89
Medianflow	324.13	80.45	87.63
MIL	11.27	73.90	106.86
Mosse	773.63	157.38	115.13
Pattern matching	148.20	8.86	4.04
Simple background	116.59	0.00	0.00
TLD	15.16	11.26	6.75

Table 4.1: Estimation of the trackers speed and inaccuracy.



(a) Selecting bounding box for HSV tracker

(b) Simple background tracker (red) and HSV tracker (blue)

Figure 4.7: Testing HSV tracker

The results of the experiment are available in the Table 4.1. We provide sample expected value (denoted $\bar{X}$) and sample standard deviation (denoted σ_X) of the inaccuracy. Both values are in the pixels. For the HSV tracker we chose to track the blue battery which has one-color area.

From the results, we can see that MIL, Boosting and TLD trackers are too slow for our application which runs live.

We can also see that, although the MIL and BOOSTING trackers were slow, they performed quite well regarding accuracy. The trackers which have mean value of inaccuracy more than 100 px lost the object at some point and were not aware of it.

As a result from this experiment, we consider Correlation, Simple background, Medianflow, Pattern matching as usable for our purposes.

To test HSV tracker, we modified the experiment by placing an orange paper to mark the top of the robot (see Figure 4.7a). The obtained values for this tracker and the others are listed in the Table 4.2. For improving the performance of the HSV tracker, it is better to select smaller area with the same color. Therefore, the tracker will track only this are, not the whole object. The difference between tracked are by Simple Background tracker and the HSV tracker is displayed in the Figure 4.7b. The red line symbolises Simple Background tracker and the blue one the HSV tracker.

Tracker	Speed [FPS]	$\bar{X}$ [px]	σ_X [px]
Boosting	24.04	10.40	4.71
Correlation	111.68	211.09	168.03
HSV	486.79	20.26	2.05
Medianflow	331.74	163.55	132.94
MIL	11.53	24.43	20.77
Mosse	535.84	154.97	121.61
Pattern matching	148.64	136.91	182.31
Simple background	118.97	0.00	0.00
TLD	16.93	28.28	15.10

Table 4.2: Estimation of the trackers speed and inaccuracy in the experiment with orange cap.

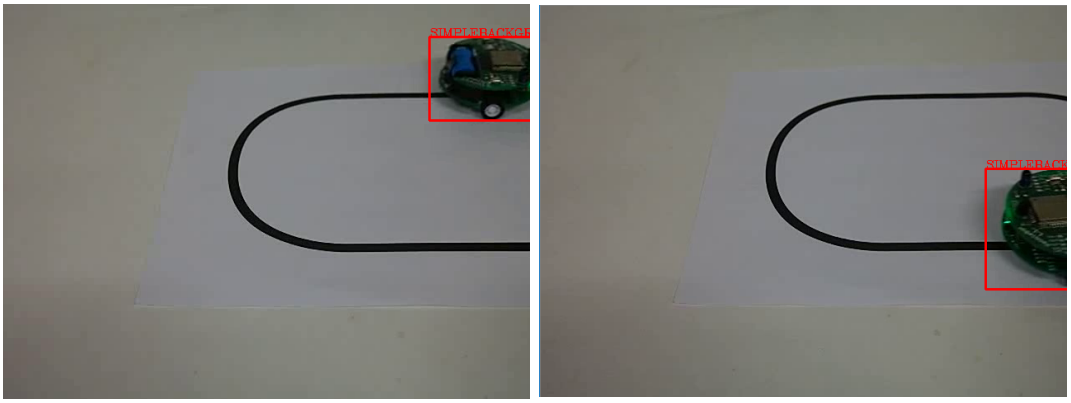


Figure 4.8: Example of successfully recovering from the full occlusion

4.3.2 Object under occlusion

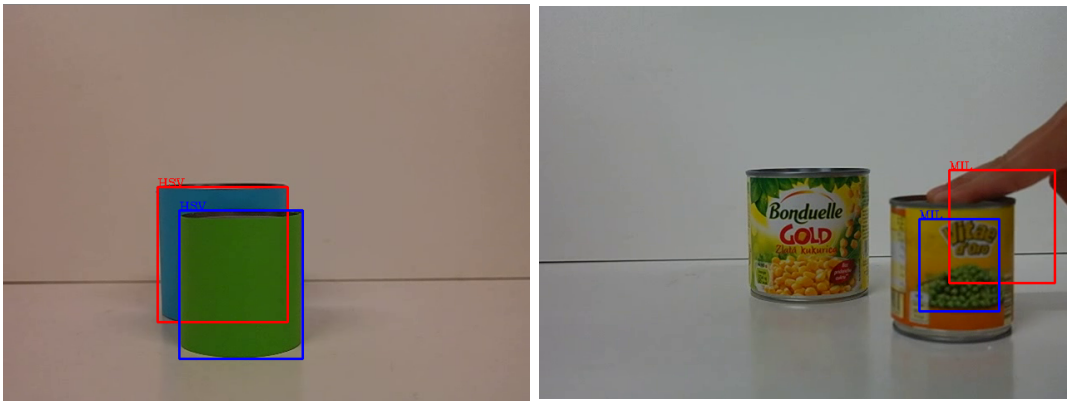
As we have mentioned earlier, it might come handy to have a tracker which is able to recover from occlusion. We prepared an experiment which lost the robot from the view. We were interested if the tracker is able to report object loss (or returns wrong results) and if it was able to track the object again after coming back to camera view. The example of the successful recover from the occlusion is displayed in the Figure 4.8.

In the results 4.3 we display also a ratio how successful the tracker was in given situation. We set a threshold for successful tracking to 80 px, what is approximately the diameter of the robot in the used video.

4.3.3 Tracking multiple objects

The last test for the trackers is to checht their ability to track multiple objects. Simple detection-based trackers usually choose the best position with the biggest area satisfying a given condition therefore tracking multiple object may not be possible.

In a situation, when tracking multiple objects is needed, sequence-based trackers may outperform detection-based trackers. We decided to test their ability to keep tracking the same object, the results are listed in the Table 4.3 in the fourth and fifth columns.



(a) Successful (HSV) (b) Unsuccessful (MIL)

Figure 4.9: Example of tracking multiple objects

The Simple background tracker was not able to track multiple objects at all. When using other trackers, there was no difference when tracking the objects separately, or together.

Most of the trackers are not able to recover from mutual occlusion. In the video we observed a situation, when one object covered most of the other one. In such situation it often occured, that tracker originally tracking covered object started to track the object in front and continue to do so even when the covered object was visible again.

4.3.4 Summary

In the previous section we provided a comparison of the trackers. The goal was to find the best performing trackers, but the results show, that it might be better to choose the tracker depending on the environment.

In the speed results we saw, that the trackers MIL, Boosting, TLD were too slow (less than 30 FPS). Because of their speed, we prefer to not using them.

If the object has an area, which is one-colored, the HSV tracker is a great choice. The HSV tracker performed very well, it is a fast and stable tracker. Unfortunately, the disadvantage of this tracker is not only a requirement for object to have one-colored area, but also this color should not be present in the background.

If an object is moving autonomously and the only moving object in the view of the cameras is the tracked object, then we can use Simple background tracker.

Pattern Matching tracker works well with objects which preserve the shape and the size during the tracking.

In conclusion, it is best to try several trackers and find the one, which perform the best in given environment. We provide some tips how to find the best tracker:

- if the object the only moving object – use Simple background
- if the object has onecolor area – use HSV
- if the mutual occlusion of the objects may appear – try TLD
- otherwise, it is worth to try Medianflow, Correlation and sometimes also Mosse and Pattern matching

Tracker	Report object lost (ratio of successful detections)	Recovers from full occlusion (ratio of successful tracking)	Ability to track multiple objects	Able to track correctly after mutual occlusion of tracked objects
Boosting	No (0%)	No (14.89%)	Yes	No
Correlation	No (0%)	No (16.40%)	Yes	No
HSV	Yes (100%)	Yes (97.09%)	Only different colors	Yes
Medianflow	No (66.67%)	No (11.11%)	Yes	No
MIL	No (0%)	No (22.76%)	Yes	No
Mosse	No (0%)	No (12.08%)	Yes	No
Pattern matching	No (0%)	Yes (92.99%)	Only different patterns	Yes
Simple background	Yes (100%)	Yes (100%)	No	No
TLD	No (36.56%)	Yes (98.71%)	Yes	No

Table 4.3: Summary of the trackers abilities

5. Localization

In the previous chapters, we have covered the steps to obtain a position of an object in 2D images. In this chapter, we will take a closer look at obtaining a position of an object in the world coordinates by combining information from multiple cameras.

At this point we have computed not only camera parameters for each camera but also the rotation matrix and the translation vector describing their relative position. Our goal is to get a position of the object in world coordinates from a tuple of coordinates from the images taken by cameras.

5.1 Projection matrices

Projection matrices provide a way to transform world coordinates to image coordinates. The first step is to find projection matrices for both cameras, and then we will use them for solving the triangulation problem.

We define a projection matrix as a transformation matrix P, such that: $x = PX$, where X denotes a vector of size 4×1 – homogenous world coordinates of the object and x denotes homogenous object coordinates in the image plane of the camera – a vector 3×1.

5.1.1 World coordinate system

We define the world coordinate system as orthogonal, with the origin in the center of projection of the first (usually left) camera. The positive part of the z-axis is pointing in front of the camera and below the camera is positive y-axis and to the right is positive x-axis. Layout and the coordinate system is displayed in the Figure 5.1.

5.1.2 Computing projection matrices

After defining coordinate systems, we can compute the projection matrices for both cameras. We use projection matrix composition to get the projection matrix from the calibration results.

We compose the projection matrix as $P = K(R|T)$, where K is intristic camera matrix and $(R|T)$ is extrinsic matrix. R is a rotation matrix and T is a translation vector. We use this composition to compute the projection matrices.

First camera – Since we set the origin of the world coordinate system in the first camera, the camera has no rotation nor translation to the coordinate system. Therefore we compute the projection matrix as:

$$P = K_1 \cdot \begin{pmatrix} I_{3,3} & | & 0_3 \end{pmatrix}$$

Where K_1 denotes intrinsic parameters matrix of the first camera, $I_{3,3}$ identity matrix 3×3 and 0_3 zero vector. Only intristic parameters matrix take effect on given coordinates, computing from world coordinates in the image plane of the camera.

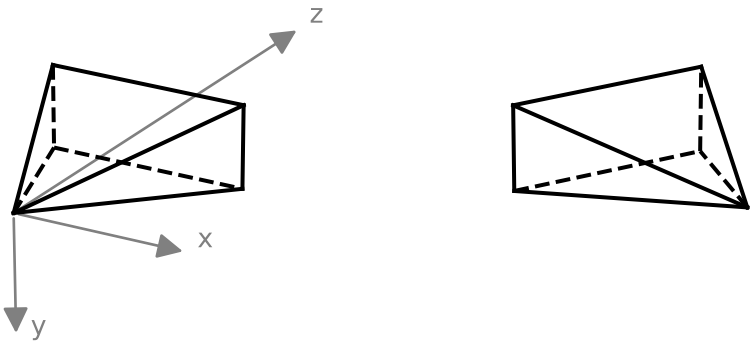


Figure 5.1: Cameras layout and the coordinate system

Second camera – For the second camera's projection matrix, stereo calibration results will be used. We know the rotation matrix and the translation vector between the cameras, being able to get coordinates of the second camera relative to the first one.

We now use this information in construction of the projection matrix, $Q = K_2(R|T)$, where K_2 is intristic parameters matrix of the second camera, R rotation matrix and T translation vector.

More about the decomposition itself can be found in an article by Simek [2012].

5.2 Triangulation

Now when we know the projection matrices we can formalize our problem as

$$x = PX, y = QX \tag{5.1}$$

with the goal to find X. However, errors may occure during measurement of x, y and calibration. In further steps we consider that calibration results are provided with high accurancy compared to measurement of x and y (that is the reason to have longer calibration time with more images at once).

We describe simple linear triangulation, which rewrites the equations 5.1 into format $AX = 0$. A solution for this equation for unknown X will be the position of the object in world coodinates. This approach provides a way to determine four unknowns – coordinates of the point X (homogenous).

5.2.1 Simple linear triangulation

We will now shortly describe how the triangulation works under the hood.

We start from the single equation $x = PX$ and try to rearrange it. Since we do have two such equations (for each camera, see 5.1), we will create enough equations for finding the unknown position (X) of our object in world coordinates.

We denote a point $x = (a, b, c)$ where x represents homogenous coordinates of the object in the image. We also choose a, b, c in a way to have $c = 1$. We can do that, since the coordinates are only up to scale factor.

Then we use the fact, that the cross product of the vector itself is equal to zero vector. Therefore $x \times PX = 0$ from the previous fact. We rewrite it to three equations using crossproduct rules:

$$c(p^{1T}X) - a(p^{3T}X) = 0$$
$$b(p^{3T}X) - c(p^{2T}X) = 0$$
$$a(p^{2T}X) - b(p^{1T}X) = 0$$

Where p^{iT} denotes ith row of P. Since $c = 1$ we can equally write:

$$a(p^{3T}X) - (p^{1T}X) = 0$$
$$b(p^{3T}X) - (p^{2T}X) = 0$$
$$a(p^{2T}X) - b(p^{1T}X) = 0$$

These equations are linear in the components of X. Only two equations are linearly independent since the third one could be obtained as the sum b times the first row and $-a$ times the second row. Therefore, we use only the first two equations

In this way we obtained two equations from $x = PX$. In our case, we have two such equations – one for each camera (5.1). We denote a point $x = (a, b, 1)$ and $y = (m, n, 1)$. Then using the steps described in previous paragraphs we obtain these equations:

$$a(p^{3T}X) - (p^{1T}X) = 0$$
$$b(p^{3T}X) - (p^{2T}X) = 0$$
$$m(q^{3T}X) - (q^{1T}X) = 0$$
$$n(q^{3T}X) - (q^{2T}X) = 0$$

We now can obtain an equation in the form of $AX = 0$. The solution of this equation is our solution for the position of the object in the worlds coordinates.

The matrix A has the following format:

$$A = \begin{pmatrix} a(p^{3T}) - (p^{1T}) \\ b(p^{3T}) - (p^{2T}) \\ m(q^{3T}) - (q^{1T}) \\ n(q^{3T}) - (q^{2T}) \end{pmatrix}$$

We can easily check, that multipling the matrix AX gives us the same equation desribed earlier. Now we analyze what we obtained.

For each image two equations were included, giving a total of four equations in four homogeneous unknowns.

Without an error during measurements a point X satisfying $AX = 0$ would exist. However, due to the errors it might not exists. Therefore, a solution to this equation cannot be find easily. This problem can be solved by direct linear transformation algorithm (DLT), which will estimate X. More about the method could be found in Hartley and Zisserman [2003].

Implementation note

For the triangulation we used OpenCV function triangulatePoints(P, Q, x, y), which is based on simple triangulation method with use of DLT method for solving equations.

6. Experiments

For the designed system we haven chosen multiple experiments to verify its accuracy. Experiments in this part focus on the localization process and overall experiments. The tracker experiments are located in the chapter 4.

6.1 Calibration and localization

In order to measure a quality of calibration and localization we exclude a tracking component. Therefore, we propose an experiment with static points in a camera views. Then we estimate a position in 3D for these points.

It is complicated to measure distance from the origin of the coordinate system, i.e. point inside the first camera to any given point. Therefore, we measure distances between the static points in the real world and compute the distance between their estimated position in 3D coordinates. Then we compare the results.

To skip the tracking part, we created a new tracker. The new tracker always returns the same position. Doing this, we excluded tracker from our process.

We have chosen the grid as in the Figure 6.1 as our pattern for experiments. The vertical lines are circa 400 mm long and the distance between them is circa 200 mm. We measured distances between the crossings. We numbered the crossing, the left column is from top to the bottom from 1 to 7 and the right column from 8 to 14 (see the Figure 6.2).

The setup of the cameras was nearly parallel, which means, the center rays of their views were parallel, looking in same direction. The distance between them was circa 16 cm. Selected points are displayed in the Figure 6.4.

We repeated the experiment ten times, with exactly same setup in order to get more reliable results. The only difference was the frames chosen for the calibration (frames for the calibration were randomly chosen from the same video). The number of the frames was same in each run. Therefore, we obtained slightly different calibration data on the same setup.

With each set of calibration data we estimated position of the static points. Then we computed distances between them. From such results from ten runs, we computed average errors.

The results are listed in the Table 6.1. First two columns define the pair of point between which are the results in corresponding row. Third column denote

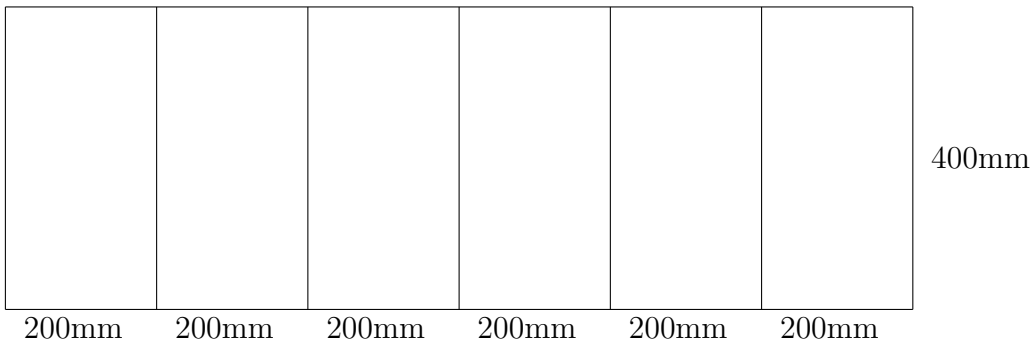


Figure 6.1: Pattern used for experiments

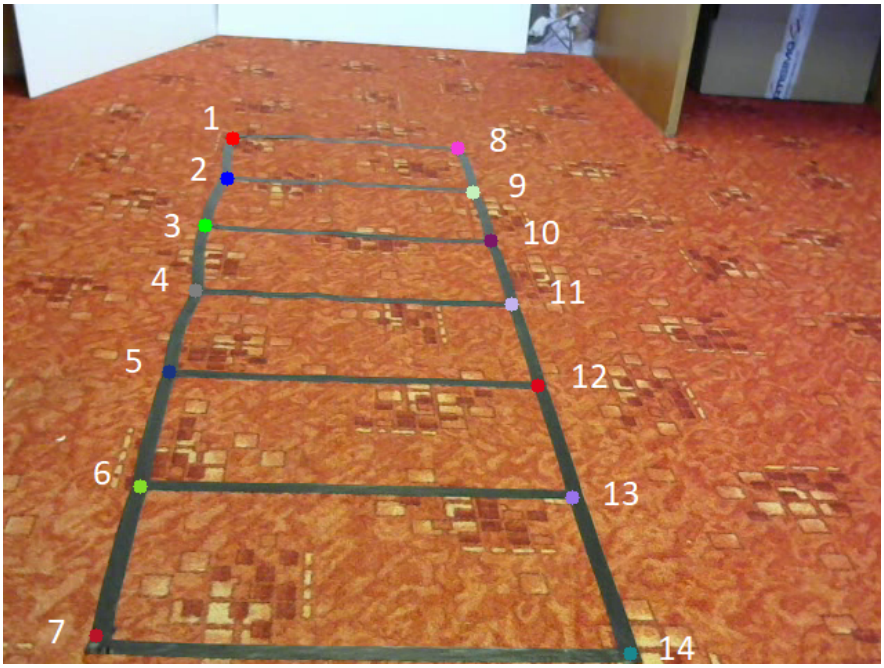


Figure 6.2: Numbering the points

the real length. The first column of the rests display the average absolute error. Next columns display its relative value. The last column displays the sample standard deviation for absolute error.

To observe the results better, we have used also box plot displayed in the Figure 6.3. This box plot represents same results from the same experiment. The orange line for each pair of points represent median of the results. The 50% of the results lies within the box. We denote height of the this box as interquartile range (IQR). The whiskers represents all other measurements, which lies at most one and half of the IQR away from the box. All measurement which do not belong in previously mentioned groups are considered as outliers and will be displayed as circles.

Based on the results displayed in the Table 6.1 and the box plot 6.3 we can see that estimated results are least accurate for the points furthest from the cameras. Movement by few pixels so far away from the camera means great real movement. On the other hand, the movement by same amount pixels closer to the camera means far smaller real distance. Because of that, the error is expected to increase for further points.

On the contrary, we can see that results too close to the camera are actually a bit less accurate than the results close to the center of the view. The points close to the camera are at the edge of the view. These less accurate results may be caused by wrongly estimated distortion coefficients, as distortion coefficients have major effect at the edges of the image.

Other box plots for the same setup with displayed results for the left and right column, and for the perpendicular plane are displayed in the Appendix D.1.

We were interested if the results are good not only on the one plane – in our case the ground – but also in the others. Now we have tested the precision in two axes, which were perpendicular to each other (vertical and horizontal lines). We have noticed, that the results for the vertical lines are less precise. Therefore, we

Points		Real	Error		
From	To	length [mm]	Average [mm]	Average [%]	Std. dev. [mm]
1	2	200	32.25	16.13	23.96
2	3	200	69.08	34.54	15.72
3	4	200	23.36	11.68	14.21
4	5	200	19.06	9.53	11.52
5	6	200	21.86	10.93	13.55
6	7	200	25.84	12.92	11.53
8	9	200	39.11	19.55	11.99
9	10	200	47.49	23.75	25.33
10	11	200	15.09	7.55	8.48
11	12	200	9.74	4.87	8.80
12	13	200	13.99	7.00	5.49
13	14	200	35.93	17.97	11.10
1	8	400	19.55	4.89	12.64
2	9	400	25.31	6.33	21.40
3	10	400	11.77	2.94	5.10
4	11	400	14.15	3.54	6.98
5	12	400	10.58	2.65	4.26
6	13	400	6.43	1.61	2.78
7	14	400	9.32	2.33	4.48
15	16	200	14.68	7.34	9.12
15	17	400	17.82	4.46	8.03
17	18	200	3.43	1.71	2.99
18	16	400	13.32	3.33	7.91

Table 6.1: Results of the experiment focused on the distances

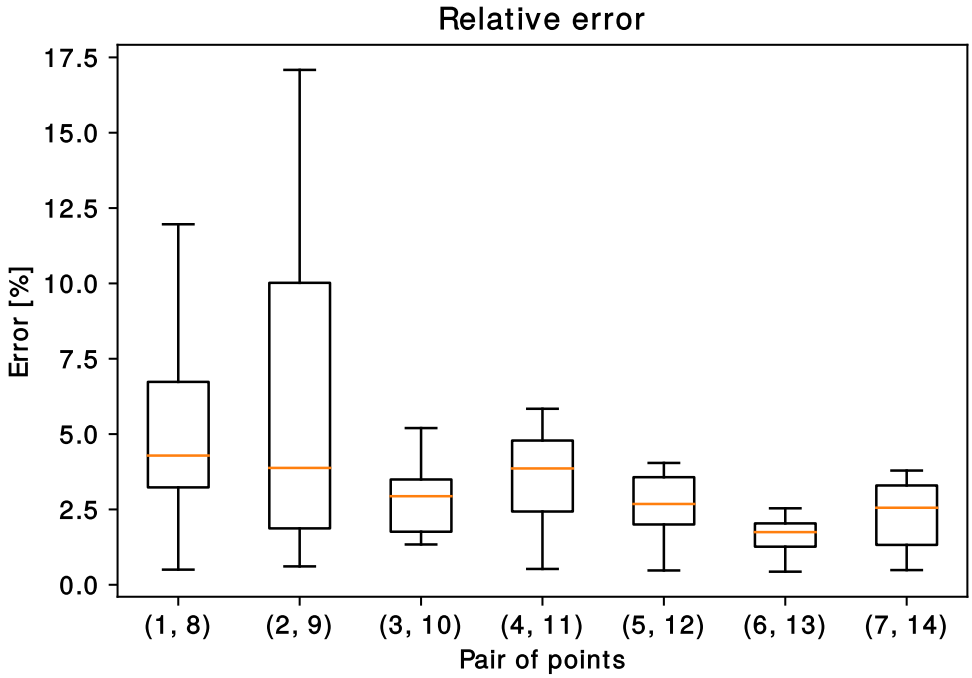


Figure 6.3: Box plot of relative errors of estimated horizontal distances between the columns (parallel setup)

decided to test the last axis and its precision. We place a wall with the marks, which is perpendicular to the ground and measured the distance between the marks. The setup can be seen in the Figure 6.5. The marks are numbered as displayed in the image 6.5a. The results are included in the Table 6.1.

We did the same experiment with another setup, when the cameras were 63 cm apart from each other. We were interested, if the results are better in parallel setup, or not. Based on the results, the parallel setup performed better when the lines were parallel to the camera X-axis (i.e. horizontal). On the other hand, the setup with cameras far from each other and not having parallel view performed better in measuring non-horizontal lines.

Even though we explored this fact, we were not able to find out the cause. In general, the mathematical background for parallel setup is more straight-forward and thus provide more stable results. It also provides almost full correspondence of the views, therefore it is preferable to use for depth maps. On the other hand, the non-parallel setup should provide better precision for closer objects. Similar results (parallel setup performing better in some cases and non-parallel setup in others) were observed also for example by Cardenas-Garcia, Yao, and Zheng [1995].

The results for this second experiment are listed in the appendix in the Section D.1. Same numbering of the marks was used.

6.2 Complex experiments

As the last step, we decided to test the whole system in a real environment. This time we included all parts of the application – calibration, tracking and also

Figure 6.4: Selecting the points from the videos

(a) Left view of the camera with marks (b) Right view of the camera

Figure 6.5: The views of the camera on the perpendicular plane to the ground

Figure 6.6: Experiment with the robot

localization. In the previous experiments, we tested the liability of the trackers and the precision of the results. Now we test the usability of the system as the whole.

6.2.1 Experiment with the one small object

The most important experiment – the experiment which decides if the system is usable – is experiment under real conditions. We created a square to follow by autonomous robot. This square have the length of the side equal 14.5 cm (see Figure 6.6). The robot did four laps around the square. Its speed was approximately 0.2 ms^{-1}. For tracking we used HSV tracker. The results are displayed in the Figure 6.7.

We can see in the left picture, that the drawn line is similar to our shape. Not having sharp edges, since our robot turns in the bends over the inner wheel. On the other hand, the results from the another view are not so amazing. The coordinates have a lot of noise over Z-axis. This noise is under 5 cm, on average 1.4 cm. The noise is only present, if the robot is following horizontal lines of the square. This corresponds to our results mentioned in the Section 6.1 – the horizontal lines are less precise, because the camera views are not parralel.

As the last step we computed maximum of the distances between any two points, which were located. In this case, the expected value the length of the diagonal, which is equal to 20.5061 cm. Our results are between 18.54 cm to 19.38 cm, depending on the run. That means, that the error is only usually between 1 and 2 cm. This corresponds to previously measured inaccuracy in non-parralel alignment.

6.2.2 Experiment with two objects

This experiment will test the ability of the system to track multiple objects. This effectively exclude Simple background tracker as it is not able to track multiple objects. Furthermore, we track objects with multicolored surface, making HSVtracker also unusable. The rest of the trackers are mostly slower, more problematic.

We can see the setup for the experiment in the Figure 6.8. We can see two boxes, both in shades of blue. During testing many trackers failed to track the

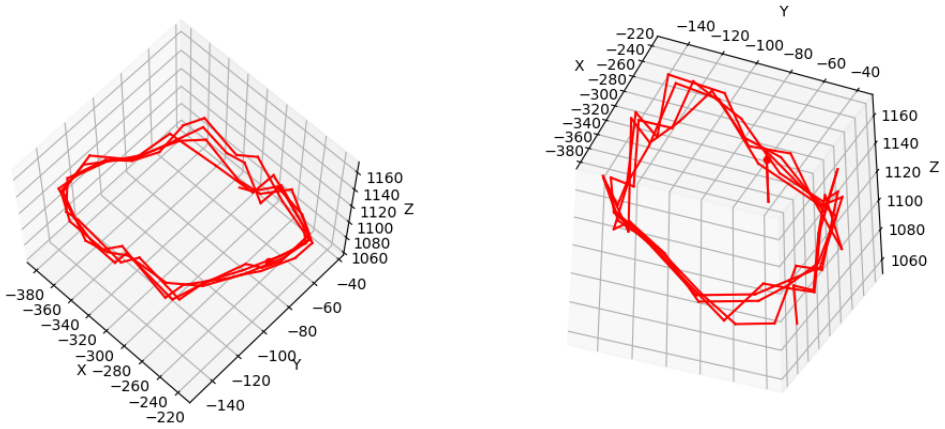


Figure 6.7: Results of the localization – four laps by the robot

object and lost it. We chose the Correlation tracker as it provided best results. It is medium fast tracker. The tracker sometimes lost the images, but most of the time it was able to track. Because of the object lost, sometimes an interaction of the user is needed, to reinitialize a tracker.

Sample results of the trajectories for two objects are displayed in the Figure 6.9. During the experiment, the boxes were moving between the yellow marks and swapped places. We can see these four marks easily as the points, where the trajectories significantly changed directions.

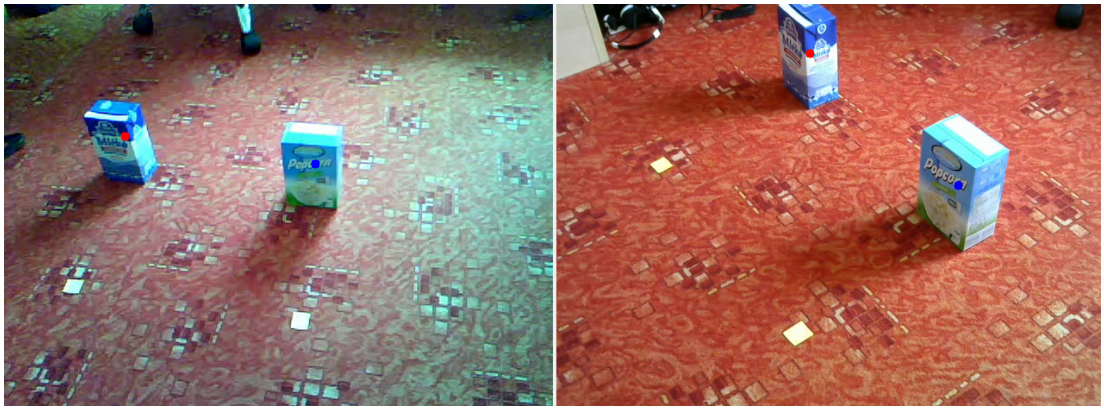


Figure 6.8: Initialization of the two objects

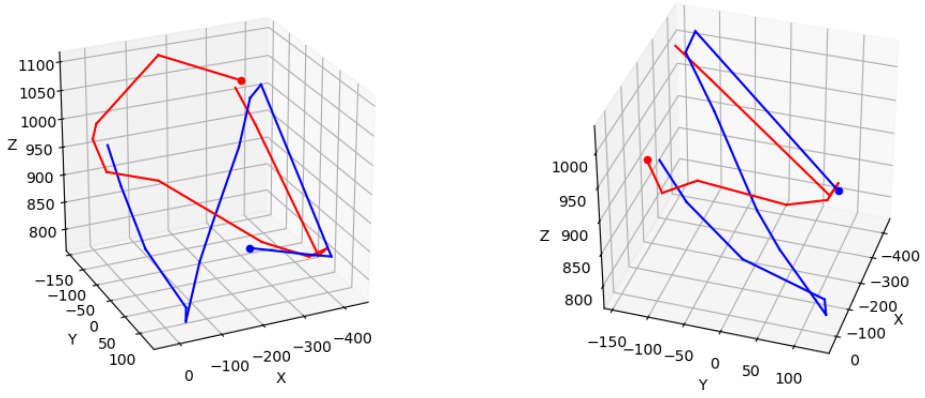


Figure 6.9: Sample trajectories from tracking two objects

Conclusion

This thesis proposed a system for visual object localization in the 3D space.

The primary goal of this thesis was to provide a step by step guide for the task of the localization of an object in 3D. We achieved the goal by describing the whole problem and solving it step by step.

The thesis started with an overview of the related works in the area, continued with calibration process, explaining the essential elements of the calibration. We presented a short introduction to calibration routines used by OpenCV, and we also described the results obtained by this process.

We implemented detection-based trackers such as Simple background, HSV and Pattern matching. We investigated available trackers in the open-source libraries. These trackers from the OpenCV and Dlib are sequence-based. We proposed several statistics for comparison of the trackers. We tested all trackers on several setups, measuring their speed and accuracy. We also explored their abilities to track multiple objects and to recover from occlusion. We presented the results in tables and provided images for a better understanding of the concepts.

Based on the results, for environment with only one moving object the most suitable tracker is Simple background. If the object has one-colored area, of the color which is not present in the background, then the HSV tracker is the best option. Both these trackers are accurate and fast. These trackers also recover from an occlusion in contrary to the most of the others. On the other hand, if the environment does not satisfy conditions for using Simple background or HSV tracker, it is possible to use Medianflow, Correlation, Mosse or TLD tracker. These trackers usually perform well. If none of the trackers performed well enough, the last step is to try the rest of the trackers (i.e., MIL, Boosting or Pattern matching).

As the next step, we explained how projection matrices are computed. We explained their importance in the projection of a point from 3D space to a 2D view of each camera. The chapter also contains a description of simple triangulation method. This method shows how to obtain a position of the object in 3D space if the projection matrices and the positions in the images are available.

When all previous steps were covered, we tested the proposed system in several environments and settings. We studied the accuracy of the system by static experiments, and by computing the estimated distances between the points and comparing them to the real values. In the end, we also provided experiments focusing on overall experience when using the application.

The results show that the precision over specific axes depends on the alignment of the cameras. Parallel camera setup outperformed non-parallel camera in the accuracy of horizontal lines (i.e., parallel to the X-axis). On the other hand, non-parallel setup outperformed parallel setup in most of the situation, when the measured lines were not parallel to X-axis.

The measured error for the distances was usually under 10%. We noticed and also explored, that the accuracy is lower when the points are further away from the camera. Also, the points on the edge of camera view tend to have worse accuracy.

We also consider our technical goals as fulfilled. The application can calibrate automatically using a chessboard pattern, track one or more objects with a chosen tracker, display the results of the localization live. Furthermore, it can work with the recordings instead of the live camera views. The data from calibration and localization are automatically saved. Also adding a new tracker is possible.

Several areas can be explored in further work. While testing the application, we noticed a noise in some specific axis. Additional work could explore the cause of the noise and possible methods to eliminate it. We also noticed that the precision for the objects further away from the cameras is lower compared to the precision of the closer ones. The question remains, which parameters of our setup influence this precision and how much. A suitable extension of this project would be the use of more cameras and explore if this setup improves the precision.

As a conclusion, we consider the application usable in practice. The results are perfectly usable for many purposes, although not fully reliable up to millimeters. The systems provide easy access to a trajectory of the object. The project is suitable for all situation, where the trajectory log is needed with only limited precision.

Bibliography

Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. Visual tracking with online multiple instance learning. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 983–990. IEEE, 2009.

Roman Barták, Michal Koutný, and David Obdrzálek. Practical 3d tracking using low-cost cameras. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 4236–4237, 2016.

James Black, Tim Ellis, and Paul Rosin. Multi view image surveillance and tracking. In *Motion and Video Computing, 2002. Proceedings. Workshop on*, pages 169–174. IEEE, 2002.

D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui. Visual object tracking using adaptive correlation filters. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2544–2550, June 2010. doi: 10.1109/CVPR.2010.5539960.

Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

JF Cardenas-Garcia, HG Yao, and S Zheng. 3d reconstruction of objects using stereo imaging. *Optics and Lasers in Engineering*, 22(3):193–213, 1995.

Martin Danelljan, Gustav Häger, Fahad Khan, and Michael Felsberg. Accurate scale estimation for robust visual tracking. In *British Machine Vision Conference, Nottingham, September 1-5, 2014*. BMVA Press, 2014.

Helmut Grabner, Michael Grabner, and Horst Bischof. Real-time tracking via on-line boosting. In *Bmvc*, volume 1, page 6, 2006.

Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. Exploiting the circulant structure of tracking-by-detection with kernels. In *European conference on computer vision*, pages 702–715. Springer, 2012.

André Ibisch, Sebastian Houben, Matthias Michael, Robert Kesten, Florian Schuller, and Ingolstadt AUDI AG. Arbitrary object localization and tracking via multiple-camera surveillance system embedded in a parking garage. In *SPIE/IS&T Electronic Imaging*, pages 94070G–94070G. International Society for Optics and Photonics, 2015.

Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Forward-backward error: Automatic detection of tracking failures. In *Pattern recognition (ICPR), 2010 20th international conference on*, pages 2756–2759. IEEE, 2010.

Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1409–1422, 2012.

Uwe-Philipp Käppeler, Markus Höferlin, and Paul Levi. 3D object localization via stereo vision using an omnidirectional and a perspective camera. In *Proceedings of the 2nd Workshop on Omnidirectional Robot Vision, Anchorage, Alaska*, pages 7–12, 2010.

Satya Mallick. Object tracking using opencv (c++/python), 2017. URL `https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/`.

N Owens, C Harris, and C Stennett. Hawk-eye tennis system. In *Visual Information Engineering, 2003. VIE 2003. International Conference on*, pages 182–185. IET, 2003.

Kyle Simek. Dissecting the camera matrix, 2012. URL `http://ksimek.github.io/2012/08/14/decompose/`.

Satoshi Yonemoto, Naoyuki Tsuruta, and Rin-ichiro Taniguchi. Tracking of 3D multi-part objects using multiple viewpoint time-varying sequences. In *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*, volume 1, pages 490–494. IEEE, 1998.

Li-Wei Zheng, Yuan-Hsiang Chang, and Zhen-Zhong Li. A study of 3D feature tracking and localization using a stereo vision system. In *Computer Symposium (ICS), 2010 International*, pages 402–407. IEEE, 2010.

List of Figures

List of Tables

A. User documentation

The purpose of this project is to propose and implement a system for object localization using a stereo vision – two cameras. The system computes relative position of the cameras to each other using a calibration pattern. Then the user selects the object to track. Different algorithms can be used for tracking. The tracking algorithms available are detection-based and also sequence-based. When the object is found in the view of the both cameras, a position in three dimensional space is estimated. This part of the documentation is focused on the end user. We introduce installation details and manual for program usage.

A.1 Installation guide

This section documents the process of downloading until running the program.

A.1.1 Downloading the code

The code is available at `https://github.com/JankaSvK/thesis`.

A.1.2 Hardware requirements

The software was tested on a system with Intel(R) Core(TM) i5-7300HQ CPU (2.50GHz, 2496 MHz, 4Core), 16GB RAM running Microsoft Windows 10 Enterprise. Minimal requirements are lower, but the computation power reflects on frequency of getting localization results. Also, we tested the program on the Ubuntu 16.04.

Two cameras are needed. We tested using a Logitech V-U0018 and Genius Slim 1322AF. A laptop camera may be used too. Requirements for the cameras are at least 640×320 px resolution and 20 FPS. We advise to turn off the autofocus, as it changes the focal length.

A.1.3 Dependencies

The following packages are required to run the application. We also provide versions of packages used to create and test our implementation.

package	version
Python	3.4.0
NumPy	1.13.3
OpenCV-contrib	3.4.0
Matplotlib	2.1.1
Tkinter	8.6
PIL (with ImageTk module)	1.1.7
dlib	19.10.0

You can easily check the installed versions by running `checkVersions.py` in the directory `helpers/`, which is located in the root of the repository.

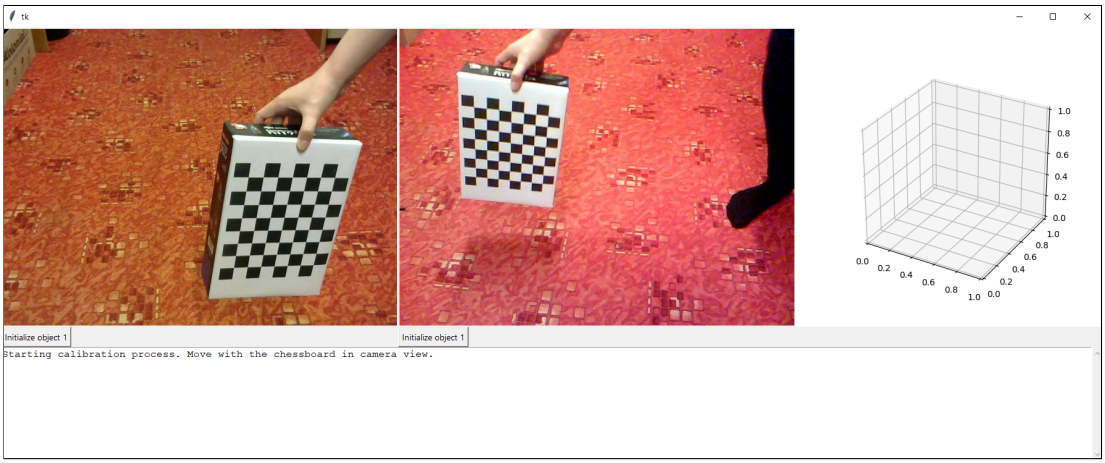


Figure A.1: Application window

A.2 First run

In the folder `program/` we find an entry point for our application `Main.py`. The text in this chapter is written with assumption that current directory is the directory `program/`. After starting the application a window will show up (displayed in Figure A.1). With no options provided, the program will run on the first two cameras available.

A.2.1 Calibration

As a first step, calibration is expected. We need a calibration pattern – chessboard (example in the Figure 3.2), which could be printed. For given chessboard, edit values in `program/Config.py`. Change the value of `chessboard_inner_corners` to a number of *inner* corners of your chessboard. For example, classic chessboard 8×8 squares has only 7×7 inner corners, so we enter a tuple `(7, 7)` as a number of inner corners. Also change `chessboard_square_size` to the size of your square in millimeters. It is important to check, if printed chessboard has squares, not rectangles, since the printer can slightly scale the image while preprocessing for printing. Moreover, calibration assumes it is a planar object, so glue it to the box or another solid object.

For calibration we have to provide a rich set of views of the chessboard. It is important to move with it and to capture it from various angles, distances and in different parts of the image. A richer set of views increases robustness of the calibration.

After each successful calibration step you will be notified in the console. After successful stereo calibration, estimated distance between the cameras will be printed. If it does not correspond to the reality, consider a recalibration. If it still did not helped, the user can increase the number of images needed for calibration in the configuration file (be careful, the computation time may increase).

If the calibration finished successfully, the calibration results will be automatically saved in `program/calib_results/`. The files will be saved into three directories, regarding if it is stereo or mono calibration. In case of mono calibration, it will be stored in the folder of corresponding camera. The hierarchy is displayed in the Figure A.2. The naming convention for these files is

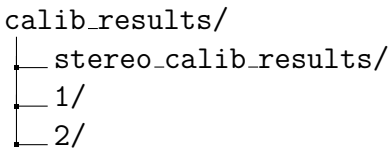


Figure A.2: Hierarchy of the directory with saved calibration results

`{year}-{month}-{day}-at-{hour}-{minute}.json`, where the date and time specify the moment when the calibration finished.

A.2.2 Selecting the objects

After successful calibration (calibrated with chessboard, or loaded from a file), the user can initialize the trackers. Under each view of the camera, a button for each object is located.

For tracker initialization, click on the button. The next two clicks in the view of the camera, will specify the bounding box for the object. After initializing trackers in both cameras, localization will automatically start.

If the tracker lost the object, the message is displayed in the camera view. Note that, not all trackers are able to recognise losing the object.

A.2.3 Localization

After initialization of the trackers, localization will start automatically. The results are displayed in the graph on the right. We can rotate the graph by grabbing it by mouse in its window. The dot represents the current position and the line represent the trajectory.

Results from localization are automatically saved at the end of the program in `program/localization_data/`. The naming convention for the files includes the date and time, when was the program closed, i.e it has following form: `{year}-{month}-{day}-at-{hour}-{minute}-{object_id}.json`.

Saved localization data consists of the four columns separated by tabs. Each line represents successful localization. In the first column is the time. In the rest three columns coordinates are stored (x, y, z).

A.3 Extras

Different options may be passed to the program (A.1). In case no option is passed, the program runs on first two available cameras. Firstly, calibration for each camera is done and then stereo calibration. As a tracker `KCF` is used by default.

A.3.1 Notes for options

- *Videos* – only AVI formats are accepted.
- *Trackers* – as `TRACKER` may be used a name of implemented tracker. Allowed tracker names are: `BOOSTING`, `CORRELATION`, `HSV`, `KCF`, `MEDIANFLOW`, `MIL`, `MOSSE` `PATTERNMATCHING`, `SIMPLEBACKGROUND`, `TLD`.

Listing A.1: Available options

```
Usage: Main.py [options]

Options:
  -h, --help            show this help message and exit
  --camera_input1=NUMBER
                        Index of camera to be run as left camera
  --camera_input2=NUMBER
                        Index of camera to be run as left camera
  -o NUMBER, --number_of_objects=NUMBER
                        Number of objects to be tracked.
  --calibration_results1=FILE
                        Calibration results for the first camera
  --calibration_results2=FILE
                        Calibration results for the second camera
  --stereo_calibration_results=FILE
                        Stereo calibration results
  --video1=FILE         Video recording for the first camera
  --video2=FILE         Video recording for the second camera
  -t TRACKER, --tracker=TRACKER
                        The algorithm used for tracking
  --chessboard=TRIPLE   Calibration chessboard parameters
                        inner_cornersX,inner_cornersY,size
  --bbox=BBOX           Bounding boxes
```

- *Calibration results* – calibration results from previous runs may be used by specifying path to the file.
- *Chessboard* – the expected format is for example `--chesboard=7,8,22`, where the first two number specify number of inner corners and third is the length of the square side in millimeters.

A.3.2 Capturing the videos

We provide additional script to capture and save videos. The script will automatically capture video from all available cameras and save it into `captured_videos`. The capturing can be exited by pressing the key "q". The script is available in `helpers/video_capture.py` from the root directory of the repository.

A.3.3 Sample scenarios

`python3 Main.py` – runs the application on first two available cameras
`python3 Main.py --video1=path/file.avi --video2=path/file.avi` – runs the application on the videos instead of the cameras
`python3 Main.py --tracker=CORRELATION` – use a correlation tracker
`python3 Main.py --chessboard=7,8,22` – using a chessboard pattern with 7×8 inner corners, each has 22 millimeters long side
`python3 Main.py -o2` – setting to track two objects

B. Developer documentation

In this chapter, we provide an overview of the whole application code base. The motivation is to help to get an idea how the program is working and to get oriented in the code.

B.1 Application parts

The program consists of several components, namely Application Process, Graphical User Interface, Cameras Provider, Calibration Provider, Trackers Provider and Localization. Each of this component has own file. In the following list, we explain the purpose in each of them.

ApplicationProcess.py

This part of the program is responsible for the running all the other parts. It runs them step by step, starting with the initialization of the cameras and ending with the localization process. For some of these tasks, it creates a new separate thread to run.

Running other parts is provided in this order:

- initialization of the cameras – CamerasProvider.py,
- initilaization of the GUI – GUI.py,
- performing mono and stereocalibration – CalibrationsProvider.py,
- initilization trackers – TrackersProvider.py,
- computing projection matrices and localization – Localization.py.

CamerasProvider.py

Cameras provider runs in a separate thread. Based on the options it will capture the images from the cameras or the video files. Capturing is stopped, when the application closes, as a final step it releases the cameras or the video files.

In case that the input is provided by video files, waiting loops are added, to not to play video accelerated. In case that computation of the other parts slows down the application, the provider will capture an image from the video, which is behind (that means, videos with different FPS may be used).

GUI.py

GUI is responsible for everything that user can see. GUI takes care of printing messages to console, updating images in the camera view, displaying results of tracking and also localization.

If the window is closed, the GUI set an event, for all other threads to signalize to end their work. Then the application exits.

CalibrationsProvider.py

We divide calibration process into two separate parts. Mono calibration of each camera and stereo calibration. Both calibrations find pictures with the chessboard

and then run OpenCV functions to calibrate. To set a number of images to calibrate on, change the value in the Config.py

TrackersProvider.py

Trackers provider takes care of all trackers – their initialization and also tracking. If the initialization of the tracker is requested, then it initializes the tracker with the specified bounding box (provided by mouse clicks) on the most recent image.

If the tracker was already initialized, the tracker's provider asks the tracker to return the position of the object in given image.

Because the trackers are usually not able to reinitialize, we encapsulate tracking algorithm into Tracker class. On this encapsulated class initialization may be called multiple times. It that case, it creates a new tracker in the background and replaces the current one by the new.

Since we call trackers by name, the class TrackersFactory provide a mapping between the names and the classes to create the trackers. Each tracker has to be added to this class.

Localization.py

Localization class is a static class. It can compute projection matrices on given calibration results, which the Localization class use later when computing the object position in the 3D space.

Then for receiving a position of the object in 3D point we need to provide coordinates from both images. Firstly it corrects the distortion from the cameras and then uses triangulation method to compute the position.

B.2 Threads

The application runs in several threads. The *Main* thread is responsible for exiting the whole application. It creates a thread for the *Application*. Also the *GUI* and the *TrackersProvider* runs in own thread. All other tasks do *Application* thread.

In Main.py the stop_event is created. This event is passed to every thread. After setting a stop event, every thread should finish its work. The stop event is set by closing application windows – so it is set inside the GUI thread.

B.3 Queues

To share information between different parts and threads, we provide several queues for the data. In this section, we provide a short description to each queue, what information it contains and which components of the program use it. All queues are created in `QueuesProvider.py`.

Image entries Queue

Images entries queue contains captured images with some additional information. We store only the last 500 pictures for each camera. The oldest pictures are

```
class TrackerSample(object):
    def init(self, image, bbox):
        # Should return True or False
        pass

    def update(self, image):
        # Should return a tuple (True/False, bbox)
        pass
```

Figure B.1: Template for the new tracker class

automatically thrown away by using `dequeue`. Using common web camera with 30 FPS, it stores the last 16 seconds per camera. The image entries are then used by the calibration process, trackers and finally displayed by GUI.

Image entry

Each image entry in Images queue contains this information:

- timestamp – the time when the image was captured. The time in seconds since the epoch,
- image – contains three channel two dimensional arrays containing the captured image,
- chessboard – information about the chessboard in the image. Access to chessboard information is provided by functions.

Tracked points

TrackedPoints2D is a queue for the results of the trackers. It is a list of the queues, separate one for each tracker's results. These results are then used by GUI to display tracker information.

Localized points

Localized points contain a list of estimated coordinates for each object. The lists are together stored in the list LocalizedPoints3D. At the end of the program, all data are automatically saved.

The Localization class adds data to this queue and GUI is adding the estimated positions to the 3D live view.

Point

The class Point is used to store tracking results and also localization results. Point consists of the time stamp (similar to the Image entry) and coordinates.

Console output

To provide a console with the information in the application window we use a list for logging information. Everything appended to this queue will be displayed in

the text box. Only strings can be appended.

To the console output can add messages any part of the program. The GUI will then display them.

Mouse clicks

Each mouse click with the left mouse button in the camera view is recorded and saved to this queue. For each camera view, we create a separate list, into which we append coordinates of the mouse click.

Mouse clicks are used only for trackers initialization. The GUI records them by calling a callback.

B.4 Adding a new tracker

In this application you can test different trackers and their ability to track an object in this task. It is possible to add a new tracker and use it.

We recommend to include a tracker into a directory with all other trackers (`program/trackers`). We keep a naming convention, so we start the name of the tracker with a prefix "Tracker".

It is important to remember, that for each object in each camera view a new instance of the tracker will be initialized.

Tracker is expected to be a class which implements two methods. The skeleton for the new tracker is displayed in the Figure B.1.

The image will be provided as a three channel two dimensional NumPy array. The bounding box is represented by four numbers, in the following order: `(x, y, width, height)`. Coordinates `(x, y)` represents top left corner of the bounding box and `width`, `height` its dimension.

The function `init` should return `True` or `False`, depending if the initialization was finished successfully.

The function `update` should return tuple `(state, bouding\_box)`. If the tracker can locate the object, it should return state `True` and corresponding bounding box in the same format as used during initialization. If the object was not found, a tuple `(False, None)` should be returned.

As the last step, we add a new tracker to `program/TrackersFactory.py`. We associate its name as a string to the tracker's class.

B.5 Configuration

Additional configuration setting are available in `program/Config.py`.

C. How to run experiments

In thesis, we mentioned many different experiments. In this chapter we provide a description how to run them. Please, firstly check the dependencies and if the application is running correctly (follow instructions in the Chapter A).

All experiments are located in the directory `program/experiments`. We will use this directory as our root in the following text. Also, every script should be executed by Python3.

C.1 Sample calibration

We provide several calibration scenarios to try out. A script named `calibrate.py` is available in the directory `localization/`. The script sets options to calibrate from different pairs of the video. To run calibration on given set of videos, run `calibrate.py {id}`, where `{id}` can be 16, 38, 43, 63. This `{id}` represents the true distance between the cameras. The estimated distance will be outputted in the program.

If the calibration cannot succeed, check the number of images needed for calibration in the `Config.py` (located in the `program/program`) and set it to smaller number.

C.2 Sample localization

In the same directory, `localization/` a script to run localization on different scenarios is available. Run `track.py {distance}{id}`, where `{distance}` represents the distance between the cameras (available 16, 38, 43) and the `{id}` represents the id of the pair for the videos (for all values of distance available scenarios 1 or 2).

The script also outputs on the first line the command which was called to run the application.

C.3 Static localization experiments

Experiments providing the results for localization without tracking are only executable under Linux based systems.

In the directory `localization_static/` a script is available. Run the script `get_data_for_ladder.py id`. Allowed `id` is 1, 2, 3 or 4.

This script create results used to compute distances between the dots. To compute the distances, a script `compute_distances.py id` is available. As `id` may be used again 1, 2, 3 or 4.

C.4 Tracker experiments

All tracker's experiments are located in the directory `trackers/`. The experiments can be started by `trackers\_experiments.py`. First argument of the

script represents the scenario for the experiment.

Speed and accuracy

`trackers_experiments.py 1`

This experiments runs all trackers at the each frame of the video. To compute an inaccuracy, we use Simple background tracker as the representative tracker (displayed with the red bounding box).

At the end, results for all trackers are displayed. The first columns specifies the tracker, the second represents FPS (more described in the chapter Tracker) and the third the inaccuracy.

`trackers_experiments.py 2`

The second scenario consists of the same experiment, with the orange cap on the top of the robot, to provide results also for HSV tracker.

Under occlusion

`trackers_experiments.py 3 id`

To test a behavior of the trackers under occlusion, we provide an experiment 3.

Admissible `{id}` is from `0` to `8`.

`trackers_experiments.py 6`

Tests all trackers under occlusion – same as experiment 3, but it use all trackers.

`trackers_experiments.py 7`

Test with thin tunnel to test trackers under partial occlusion.

`trackers_experiments.py 8`

Test with wide tunnel to test tracker under full occlusion.

Tracking multiple objects

`trackers_experiments.py 4 id`

The fourth experiment focuses on tracking multiple objects. Again, the results were evaluated by the human. Admissible `{id}` is from `0` to `8`.

`trackers_experiments.py 5`

The same experiment as previous, now with changed objects to be onecolor for the HSV tracker.

D. Additional experiments

D.1 Experiment with static point

16 cm setup

Here we provide additional results for the experiment mentioned in the Section 6.1. The box plots for distances from the left column (D.1), from the right column (D.2) and for the perpendicular plane (D.3) are available.

63 cm setup

Same experiment described in the Section 6.1 we tried with another camera setup. This time the cameras were approximately 63 cm apart from each other and their views were non-parallel. Same numbering of the grid points is used in this setup as in the parallel setup.
Results include:

- table summarizing an average error and its standart deviation for all distances (D.1),
- box plot for distances of the horizontal lines (D.4),
- box plot for distances of the vertical lines in the left column (D.5),
- box plot for distances of the vertical lines in the right column (D.6),
- box plot for distances of the lines in perpendicular plane (D.7).

D.2 Experiment with partial occlusion

In the experiment we tested trackers under partial occlusion. The results are displayed in the Table D.2 and the setup is shown in the Figure D.8a. We remind that the X means the inacuraccy between the representative tracker and tested tracker. We compute its sample mean and sample standart deviation. In this experiment the HSV tracker performed best so we choose it as representative tracker. The Simple background background had many problems wiht changing light.

D.3 Experiment under full occlusion

Similarly to the experiment mentioned in the Chapter 4 we did another experiment with full occlusion. This time, the direction of the robot is preserved. The results are listed in the Table D.8b and the setup is displayed in the image D.8b. The HSV tracker is choosed as representative tracker.

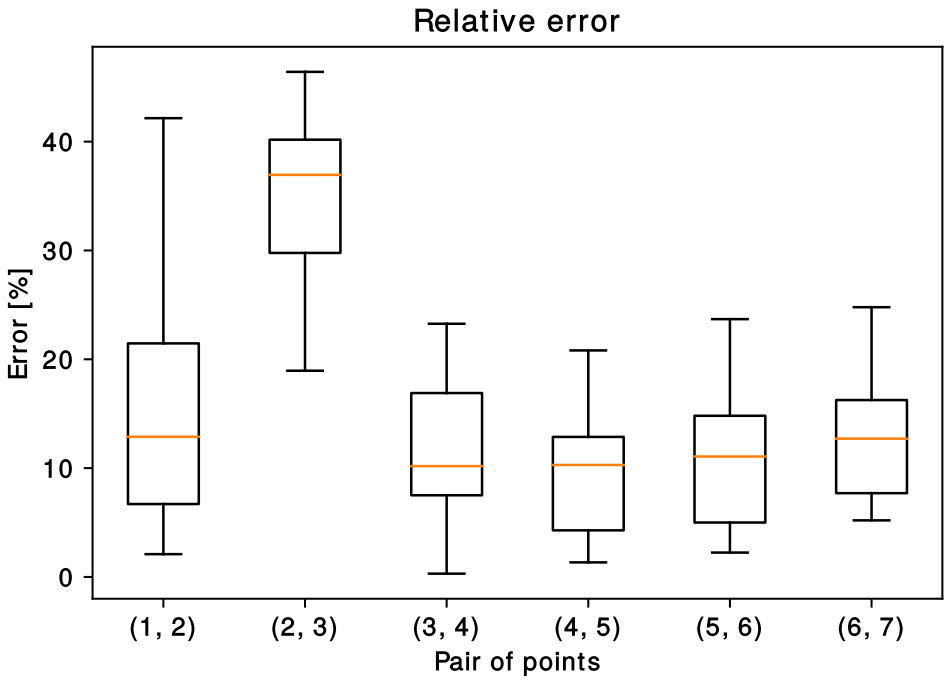


Figure D.1: Boxplot of relative errors of estimated vertical distances in the left column (experiment with parallel setup)

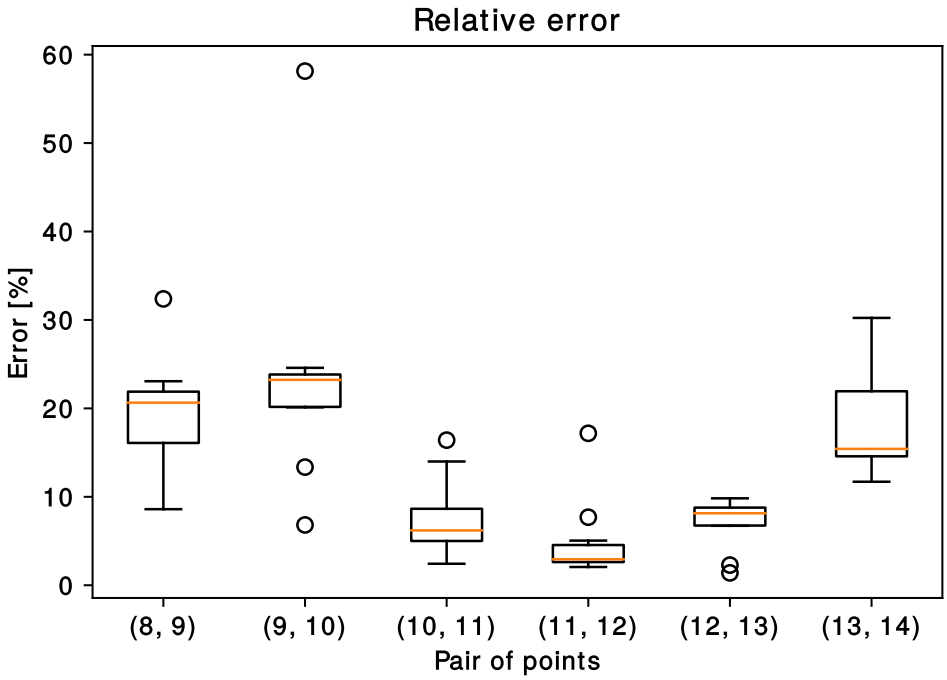


Figure D.2: Boxplot of relative errors of estimated vertical distances in the right column (experiment with parallel setup)

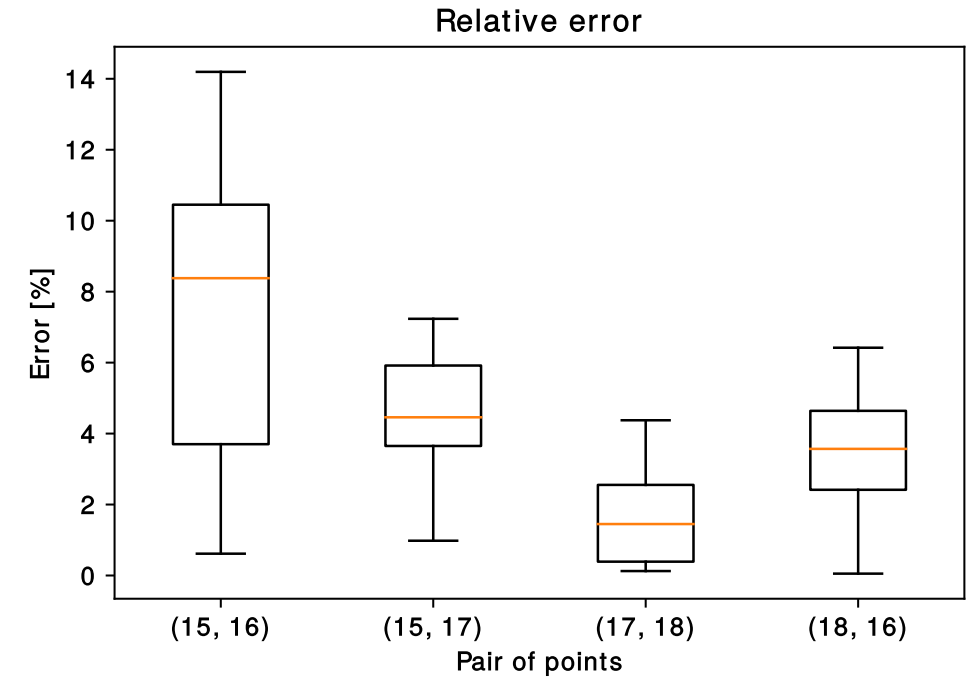


Figure D.3: Boxplot of relative errors of estimated distances for perpendicular plane (experiment with parallel setup)

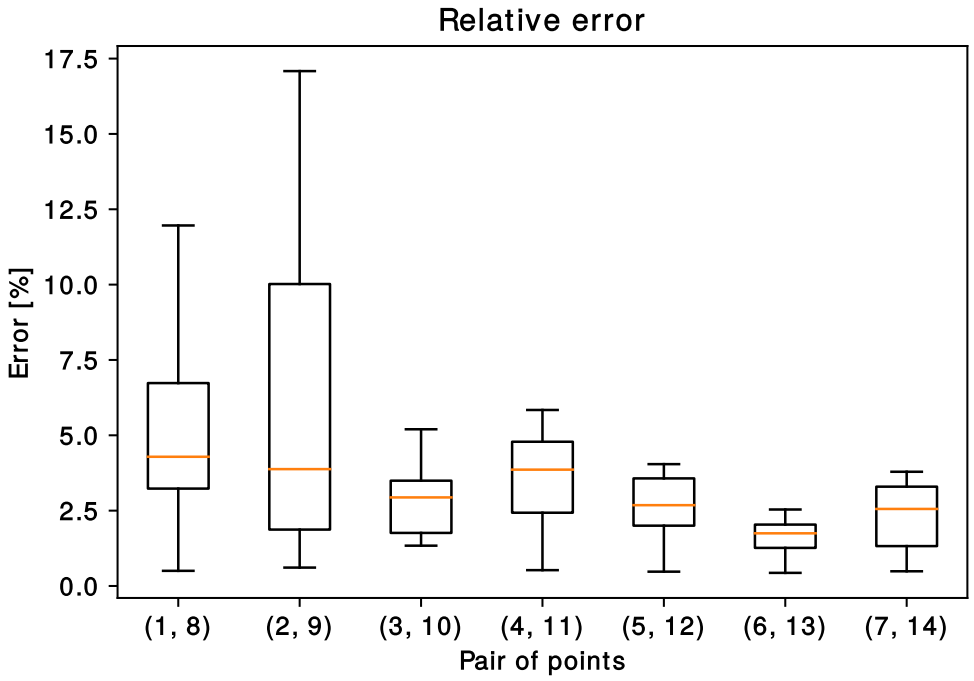


Figure D.4: Box plot of relative errors of estimated horizontal distances between the columns (non-parallel setup)

Points		Real	Error		
From	To	length [mm]	Average [mm]	Average [%]	Std. dev. [mm]
1	2	200	13.57	6.79	4.52
2	3	200	32.61	16.31	6.22
3	4	200	5.41	2.70	3.84
4	5	200	5.43	2.72	2.69
5	6	200	4.79	2.40	2.80
6	7	200	8.38	4.19	3.90
8	9	200	28.06	14.03	8.64
9	10	200	25.86	12.93	7.57
10	11	200	7.73	3.86	3.30
11	12	200	5.94	2.97	2.75
12	13	200	5.81	2.90	2.12
13	14	200	2.53	1.27	2.24
1	8	400	51.94	12.98	7.69
2	9	400	49.67	12.42	6.24
3	10	400	37.91	9.48	5.62
4	11	400	40.44	10.11	4.51
5	12	400	33.74	8.43	3.84
6	13	400	32.49	8.12	3.26
7	14	400	31.86	7.96	3.25
15	16	200	17.50	8.75	2.07
15	17	400	31.42	7.86	3.36
17	18	200	14.79	7.40	1.77
18	16	400	39.49	9.87	3.09

Table D.1: Results of the experiment focused on the distances

Tracker	Speed [FPS]	$\bar{X}$ [px]	σ_X [px]	Recovers from occlusion (ratio of successful tracking)
Boosting	19.65	208.14	153.63	No (18.60%)
Correlation	110.34	168.71	94.59	No (23.75%)
HSV	507.25	0.00	0.00	Yes (100.00%)
Medianflow	339.23	139.48	102.12	No (40.34%)
MIL	12.67	164.94	89.61	No (21.75%)
Mosse	522.00	157.59	144.83	No (32.55%)
Pattern matching	151.22	47.72	20.77	Yes (91.56%)
Simple background	120.96	42.15	12.29	Yes (100%)
TLD	16.72	43.46	18.08	Yes (96.92%)

Table D.2: Comparison of trackers under partial occlusion

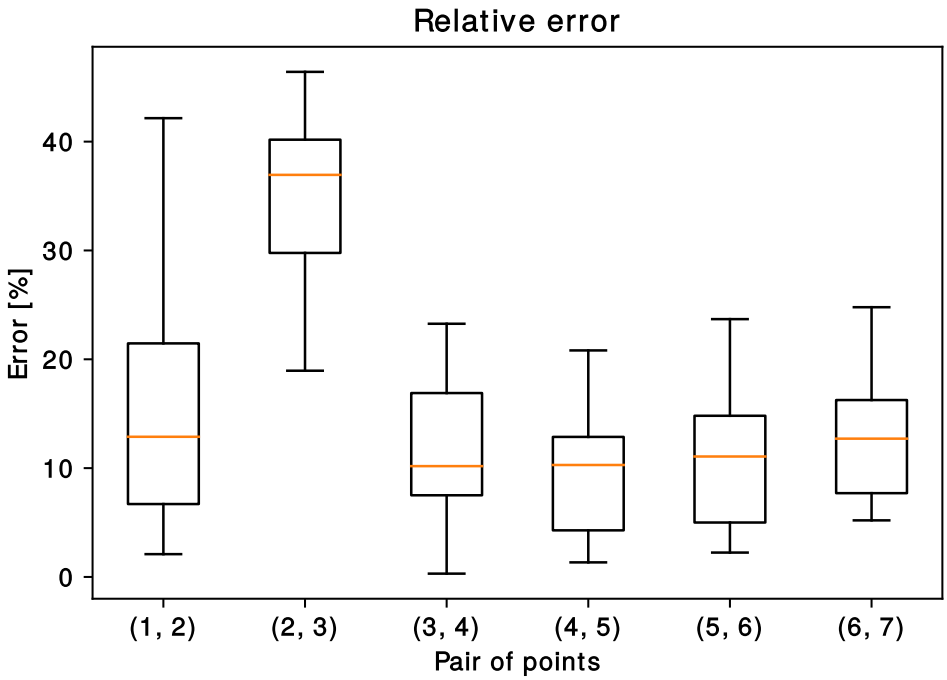


Figure D.5: Boxplot of relative errors of estimated vertical distances in the left column (non-parallel setup)

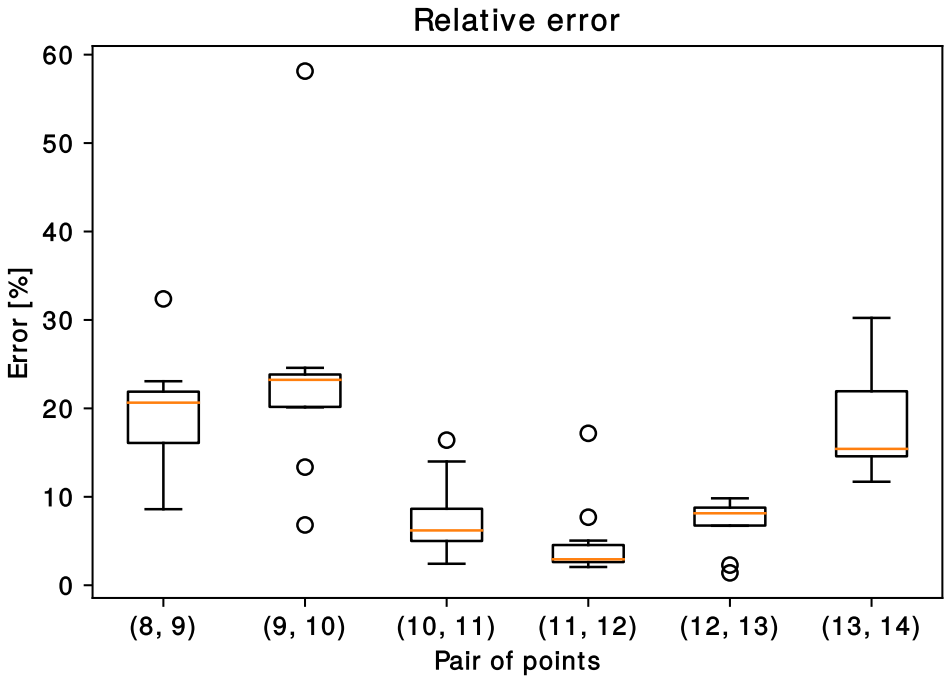


Figure D.6: Boxplot of relative errors of estimated vertical distances in the right column (non-parallel setup)

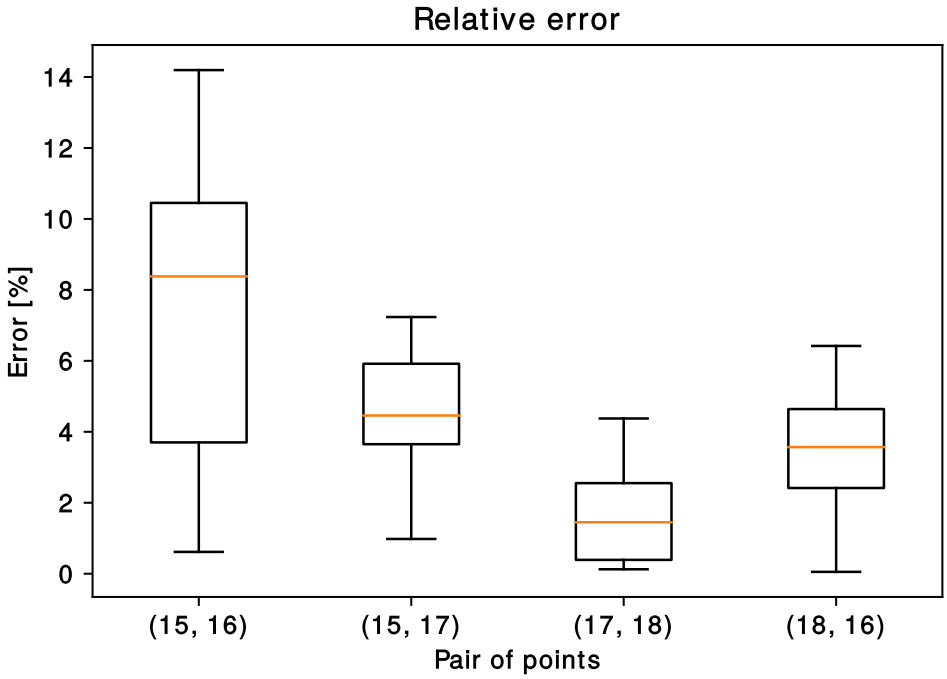
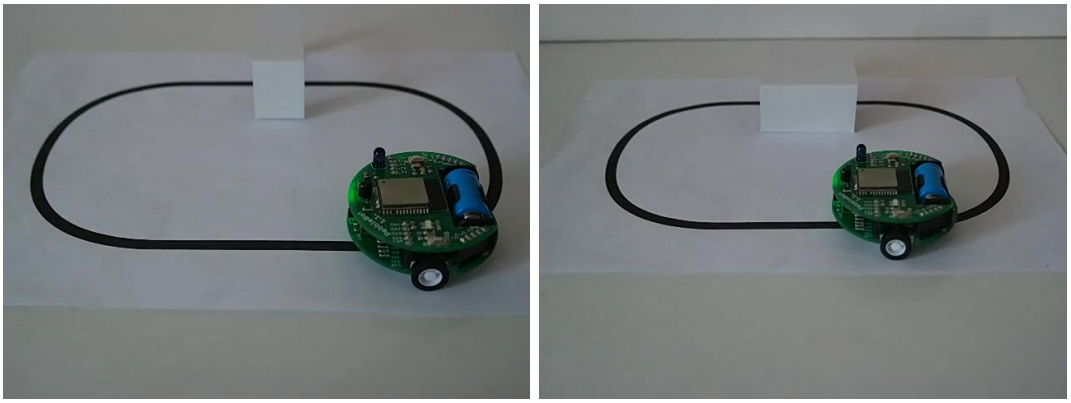


Figure D.7: Boxplot of relative errors of estimated distances in the perpendicular plane (non-parallel setup)

Tracker	Speed [FPS]	$\bar{X}$ [px]	σ_X [px]	Report occlusion (ratio of reported)	Recovers from full occlusion (ratio of successful tracking)
Boosting	15.61	137.07	91.90	No (0%)	No (17.46%)
Correlation	108.89	123.82	75.17	No (0%)	No (16.42%)
HSV	498.31	0.00	0.00	Yes (100%)	Yes (100%)
Medianflow	318.07	121.08	74.70	No (2.90%)	No (16.22%)
MIL	10.50	122.69	87.34	No (0%)	No (16.84%)
Mosse	789.96	106.57	89.99	No (0%)	No (33.78%)
Pattern m.	149.14	26.53	7.81	No (0%)	Yes (100%)
Simple b.	118.94	84.59	67.21	No (0%)	No (41.68%)
TLD	15.66	30.39	10.94	No (0%)	Yes (100%)

Table D.3: Comparison of trackers under full occlusion



(a) Setup for partial occlusion (b) Setup for full occlusion

Figure D.8: Setups used for testing trackers performance under occlusion