MEng Design Special Topic ENG5026

Self Driving Cars: How They Reason And Drive

Bailey Hodgson, Raphael Nekam

2025

Dr Bernd Porr, Ph.D.-Canidate Giulia Lafratta

Abstract

This report focuses on the development of features as part of a larger research project, concerned with studying behaviour-based path-finding algorithms for self-driving robots. The device is to operate autonomously, relying exclusively on its on-board sensors and processing capabilities. The two areas of work covered in this report are (1) the development of a laser-pointer-based target localisation system. The developed system achieved reliable target recognition within a 1.5-metre range and translated the image coordinates accurately into real-world positions using inverse perspective mapping; and (2) a system enhancing the accuracy of rotational and translational motions of the self-driving robot using a single LiDAR sensor. This resulted in a simulation that showed a method for achieving accurate 90 degree turns following Average Scan Similarity. Both developments can improve the system's autonomous behaviour and better showcase the capabilities of the path-planning algorithm developed.

Acknowledgments

We would like to thank Dr. Bernd Porr and Giulia Lafratta for their support over the course of the project. Further thank goes to Thomas O'Hara. All three helped significantly to complete this project successfully.

Contents

List of Figures

List of Tables

1 Introduction & Project Background

This project was carried out as port of the MEng Design Special Topic (ENG5026) course at the University of Glasgow. It forms a part of a broader research initiative lead by Dr. Bernd Porr and Ph.D. candidate Giulia Lafratta, focused on developing a behaviour-based path-finding algorithm for robots. The main aim is to enable autonomous navigation of a robot only knowing its target location to achieve based on a coordinate system based around itself. It is to only use sensors mounted directly onto its chassis, explicitly not relying on external tracking systems such as overhead cameras.

At the start of this part of the project, the team had already developed a reaction-based path-planning algorithm performing well in simulation. To demonstrate the system's capabilities in a real-world scenario, a physical robot relying on a LiDAR sensor is currently being implemented. Currently, the robot requires manual input of the target coordinates through either code or command-line statements. While working, this method is not ideal for demonstrations and showcases, as the numerical values of the coordinates are not directly intuitive for users to understand the location the robot is instructed to reach. Instead, a visual approach for target definition is to be implemented. It was proposed to allow navigation via a laser pointer shining a dot onto the floor, with the robot recognising the dot and understanding this as the required target location.

Further, while the movement control system is functional in simulation, it faces difficulties as part of the real-world implementation. As the robot is instructed to turn by a certain angle around its own axis, it often does not perform this motion accurately, but rather over- or under-turns by a few degrees. The robot itself does not know that it rotated to a wrong angle and can hence not adjust for it. While these errors are often small, they accumulate over time, leading the robot end up at the wrong location after following the entire path. The same applies to linear movements. It was to be explored, whether data from the already present LiDAR sensor, currently used for object detection, could be used to improve the accuracy of the motion control.

Therefore, the project could be split into two separate tasks, both together improving the showcasing capabilities of this autonomous robot system:

- **Laser Pointer Based Target Localisation:** Developing a vision system that can detect a laser pointer's presence and translate its position into a real-world coordinate system the robot can use to obtain its required target location.
- **Motion Accuracy Improvement Using LiDAR:** Using data collected by the LiDAR sensor to improve the accuracy of rotational and translational movements executed by the robot.

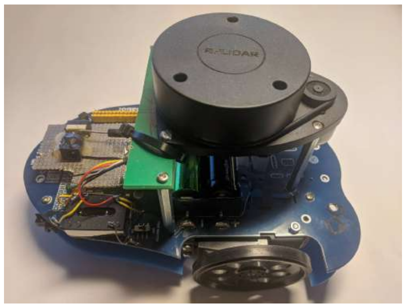


Figure 1: Alphabot with LiDAR sensor, as used in this project [1]

2 Target Localisation via a Laser Pointer

2.1 Problem Description

With the pathfinding algorithm already developed in simulation, the next step is to apply these methods in a practical scenario. The robot is expected to navigate toward a user-defined target location while avoiding obstacles along the way. Instead of specifying the target through abstract numerical input via code or the command line, a more intuitive and user-friendly approach is to be implemented: defining the target by shining a laser pointer on the ground in front of the robot. The red dot from the laser indicates the goal position, allowing the user to mark targets quickly and without modifying the physical environment (e.g., placing markers or objects).

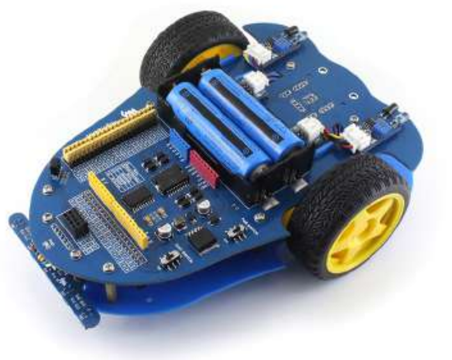
Furthermore, visual targeting mechanisms are relevant and common for real-world robotic applications, such as inspection, delivery, or transportation tasks, all commonly used in industrial use-cases. Using the capabilities of the robot in combination with visual guidance can help display the usefulness of the system.

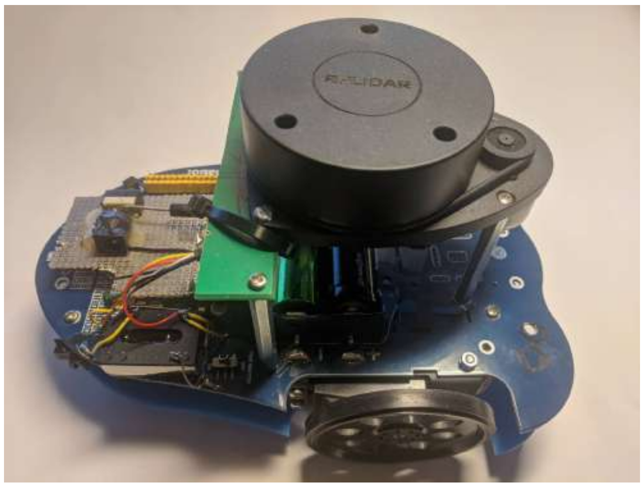
To enable laser-pointer-based navigation, the robot must detect the red dot in camera images, determine its position as pixel coordinates, and convert this into real-world distance and angle measurements from the robot.

2.2 Technical Requirements & Specifications

This project focuses on the control of Alphabot-type robots [2] (Figure 2) or similar, powered by a Raspberry Pi 3B+ [3] running Raspberry Pi OS (Bullseye). The Raspberry Pi is to handle all computations onboard, including path-finding, object detection, motor control, and the here developed laser target detection. All software is written in C++ to maintain compatibility with the existing codebase.

Environmental mapping and obstacle detection are based on a rotating LiDAR sensor mounted on the top of the robot (Figure 1). As this sensor requires an unobstructed 360° field of view, placements of additional sensors, such as the camera for the laser pointer detection, had to be considered carefully.


(a) Standard Alphabot [2]

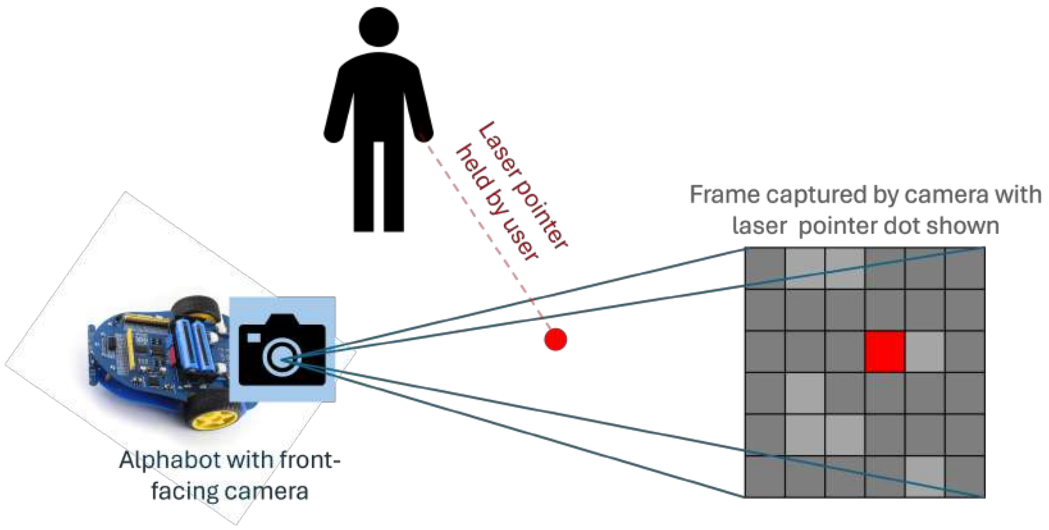


(b) Alphabot with LiDAR sensor, as used in this project [1]

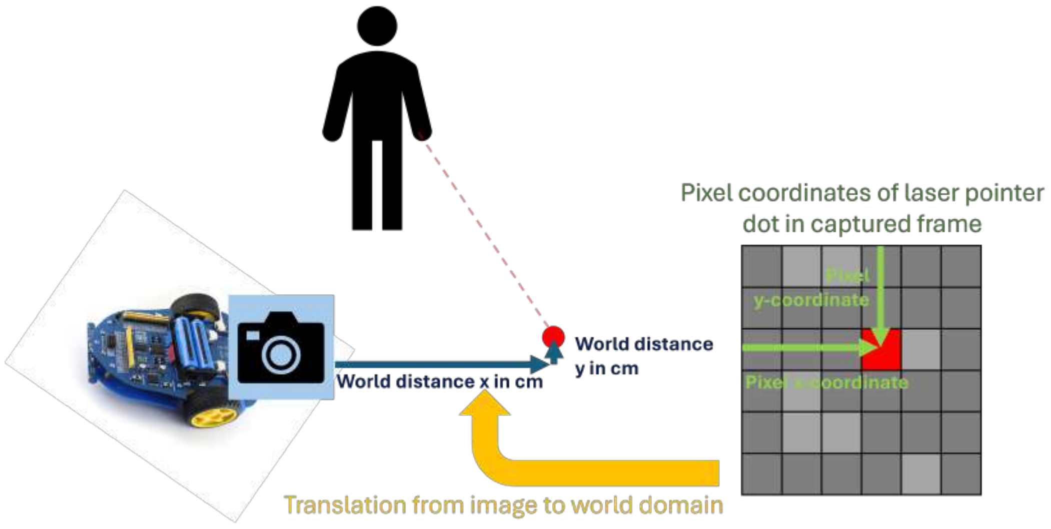
Figure 2: Images of Alphabot-Variants

2.3 Typical Usage Scenario

The robot remains stationary while witing for the user to define a new target position by pointing a laser dot on the floor. A front-facing camera captures the environment, and image processing techniques are used to detect the laser dot. The location of the recognised dot is then mapped to a real-world coordinate system relative to the robot. Once the target location is established, the robot uses its path-finding algorithms and object avoidance systems (both not covered as part of this report) and initiates movement towards it (Figure 3).


Laser pointer held by user
Frame captured by camera with laser pointer dot shown
Alphabot with front-facing camera

(a) Laser pointer detection in the image domain


Pixel coordinates of laser pointer dot in captured frame
Pixel y-coordinate
Pixel x-coordinate
World distance y in cm
World distance x in cm
Translation from image to world domain

(b) Translation of image domain coordinates to real world

Figure 3: Typical Laser Pointer Detection Usage Scenario

2.4 Related Work

Multiple laser pointer detection projects have been completed before, most of which are implemented in Python using OpenCV's built-in functionalities. One example of this is Brad Montgomery's implementation shared on GitHub [4]. In that project, a laser pointer is detected using HSV colour masking. A technique that was also explored during the development of this system. Another project worth mentioning is by Ankur, where a laser pointer is tracked using a laptop webcam and its pixel coordinates are used to control external devices, also shared on GitHub [5].

However, most projects available online, including both mentioned above, track laser pointers projected onto a surface that is parallel to the image plane. This simplifies both detection and coordinate mapping significantly. Nonetheless, they offered a useful foundation and point of comparison for the development of this system.

Another useful technique considered during development is the use of spatial moments to calculate the centre of a detected pixel cluster. By computing spatial moments of a cluster in an image (using OpenCV's `moments()` function), it is possible to obtain the centre of the features. OpenCV provides useful documentation to achieve this [6].

Another useful technique considered during development is the use of spatial moments to calculate the centre of a detected pixel cluster. By computing image moments (using OpenCV's `moments()` function), it is possible to obtain the centroid of a contour or blob by dividing the first-order moments by the zeroth-order moment (i.e.

In terms of perspective transformations, Inverse Perspective Mapping (IPM) is one of the most commonly discussed and applied methods online. Therefore, this was also explored during this project. OpenCV's documentation provides detailed information and examples on how to apply homography and IPM to map pixel coordinates to real-world coordinates [7]. These resources were particularly useful to implement the system described in this report.

While working on this project, many online examples, tutorials, and resources were consulted as needed. This process helped build a solid foundational understanding of image processing and computer vision, enabling the successful development of a laser pointer detection and real-world coordinate mapping system.

Key Takeaways from Related Projects

- **Laser pointer detection:** HSV colour masking is a common and effective method for distinguishing a laser pointer from its background in an image.
- **Centre of pixel clusters:** Calculation of spatial moments can be used to accurately obtain the centre of a given pixel cluster.
- **Real-world coordinate mapping:** Inverse Perspective Mapping (IPM) provides a reliable method for translating pixel coordinates into real-world coordinates. However, particular emphasis must be put on accurately defining the homography matrix required to perform IPM.

How this gained knowledge was applied to develop system of this report discussed in more detail in Section 2.6. Further references are provided in the following sections as needed.

2.5 Hardware Selection

2.5.0.1 Main Controlling Unit The controlling unit of the project was already predetermined by the wider scope of the project, and was set to be a Raspberry Pi 3B+ [3], in line with the previous development of the project. The operating system used was also required to be Raspberry Pi OS Bullseye.

2.5.0.2 Camera Module The camera module could be chosen freely, however, as described in 2.2 had to be mounted within the possibilities of avoiding obstructions of the LiDAR sensor used. First initial tests were completed with a Raspberry Pi Camera Module Revision 1.3. This was done simply for the reason of it being easily available. While working well, it quickly became evident that the field-of-view (FoV) was very limited (53.5/41.41 degrees of horizontal/vertical FoV [8]). Hence a switch to the latests Raspberry Pi Camera Module 3, Wide version, was made. The wide version was chosen for its increased field of view capabilities. Figure 4 shows a comparison of both cameras' viewing angles. Both cameras were mounted onto the same bracket. The captured images (Figures 4a and 4b) clearly show how camera module 3 has a much wider field of view (102/67 degrees of horizontal/vertical FoV [8]).

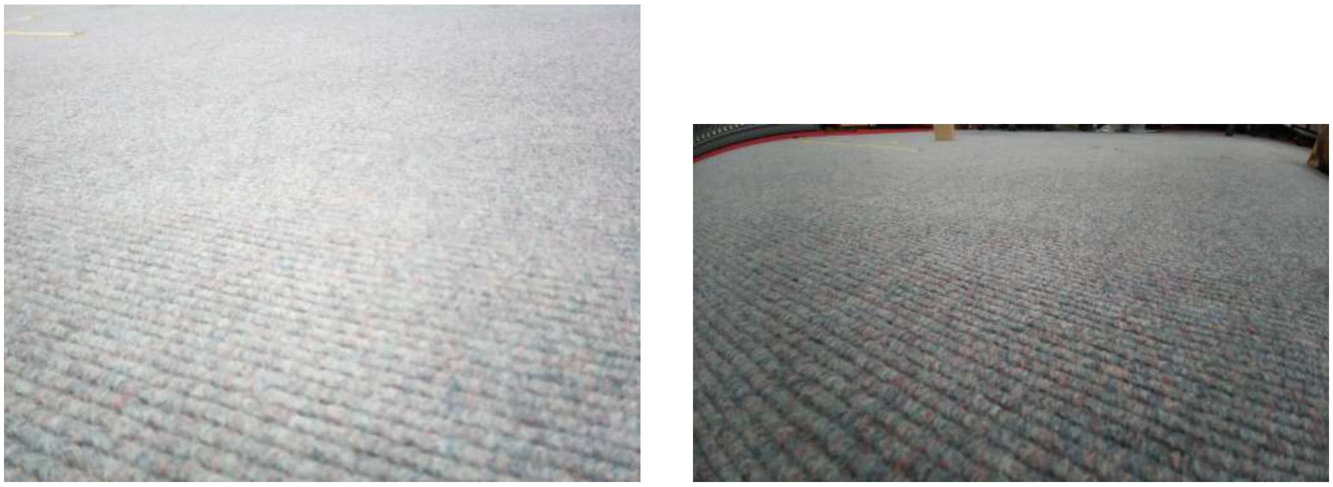

(a) Viewing angle of Raspberry Pi Camera Module Rev 1.3 (b) Viewing angle of Raspberry Pi Camera Module 3 Wide

Figure 4: Comparison of camera viewing angles

2.5.0.3 Camera Mount The camera was mounted beneath the LiDAR sensor, as required (Section 2.2). An already available metal bracket was used, allowing to fixate the camera at the front of the robot chassis at an adjustable angle. The angle was chosen through trial and error, with the camera set to capture mostly the ground. In the end, the camera was mounted at a height from the ground of approximately 7 centimetres at an angle of approximately 45 degrees, capturing a scene similar to the one shown in Figure 4b.

2.6 Methodology

Multiple options and methods were implemented and compared to identify the most effective combination of image transformations and detection strategies. The full mechanism involves two distinct steps: (1) Laser Pointer Detection in the Image (Section 2.6.1) and (2) Mapping of the Image Coordinates to a Real-World Coordinate System (Section 2.6.2). This section was structured in the same way, discussing each step and methods involved individually.

2.6.1 Laser Pointer Detection

2.6.1.1 Collection of sample images A dataset of 125 sample images was created (Figure 5). All different transformations and detection approaches were tested on this dataset, allowing for comparisons when applied to the same set of images. The photos were taken in the lab where the project was developed, allowing for parameter tuning specifically for this environment. This is important, as it can become very difficult to implement visual detection methods for a wide range of lighting requirements. The images were captured using a Raspberry Pi 3B+ and the Raspberry Pi Camera Module 3 Wide - the chosen microcontroller and camera module for this project (2.5). Objects were intentionally placed in the images to help assess how well each method performs. A red metal carabiner was used deliberately, since its shiny red surface closely resembles the laser dot in colour, providing a good way to evaluate limitations of the detection methods.

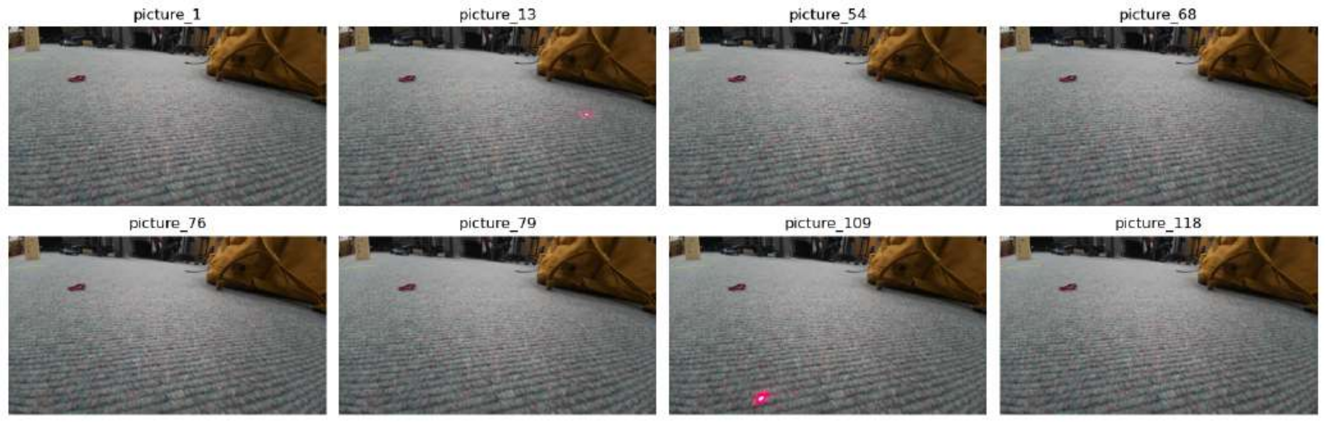


Figure 5: Excerpt of the sample image collection

2.6.1.2 Processing steps The laser pointer detection is split into 4 different steps (Figure 6): (1) Pre-Mask processing, (2) Masking, (3) Post-Mask processing, and (4) obtaining pixel coordinates. The first step, pre-mask processing, are processes applied to make the laser pointer stand out more clearly from the rest of the image. This includes capturing the frames with suitable camera parameters and optimises the images obtained for the next step: Masking. During masking, a binary mask marking potential pixel clusters of where the laser pointer is present in the image is to be obtained. Ideally, the only cluster present is the true position of the laser pointer. Next, post-mask processes are used to improve the binary mask obtained. Noise should be removed and, if multiple clusters are present, these should be further filtered to lead to only the true cluster representing the laser pointer's location remaining. The last step, obtaining pixel coordinates, does exactly what it's name implies. It should use the mask with a single cluster present and determine where this cluster is located in the image in terms of pixel coordinates.

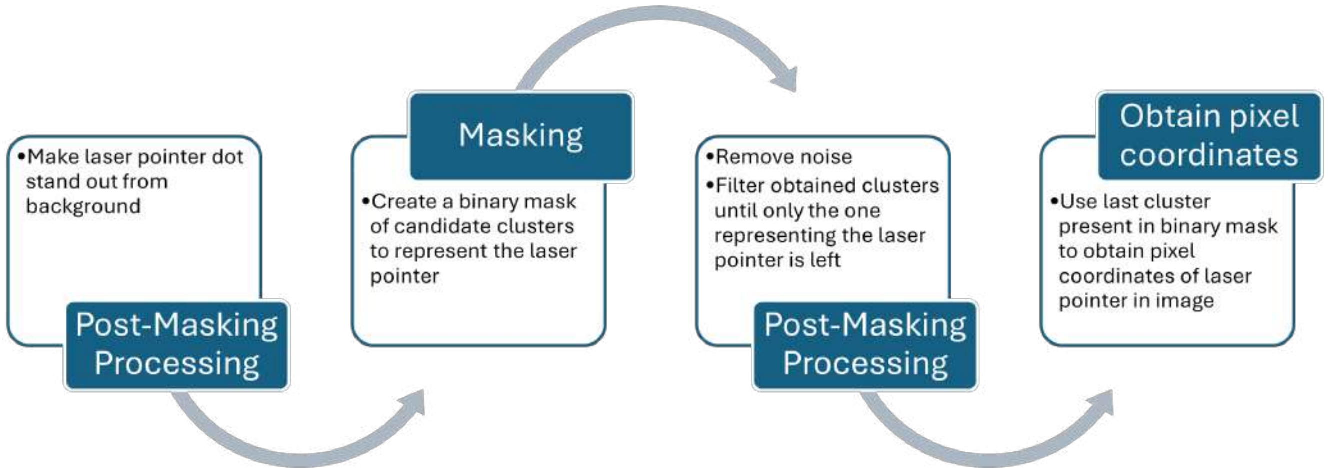

•Make laser pointer dot stand out from background
Post-Masking Processing
Masking
•Create a binary mask of candidate clusters to represent the laser pointer
•Remove noise
•Filter obtained clusters until only the one representing the laser pointer is left
Post-Masking Processing
Obtain pixel coordinates
•Use last cluster present in binary mask to obtain pixel coordinates of laser pointer in image

Figure 6: Processing steps of the laser pointer detection

2.6.1.3 Pre-Mask processing

Downsampling Reducing the resolution of images can help to improve processing speeds, as fewer pixels present require less calculations. However, depending on the way implemented, the downsampling itself may require a high amount of calculations. Further, it can reduce minor noise through the effects of binning and sampling.

The camera module used in this project (Section 2.5) offers an extremely high full-resolution of 4608 x 2592 pixels, while natively supporting hardware-based binning, reducing the resolution to 2304 x 1296 pixels while keeping the full field of view. This makes downsampling a potentially effective strategy, as benefits in performance and noise minimisation can be achieved, while avoiding computational down-sampling costs by relying on the module's hardware.

Dilation Dilation allows to expand bright regions of an image (Figure 7). The exact extent and pattern of this is defined by the applied kernel's size and shape. Dilation was initially considered as a potentially helpful pre-processing technique, allowing to increase the size of the laser pointer dot, as it appears as a cluster of bright pixels. However, as realised later, dilation is naturally only effective if the laser pointer is already visible as bright pixels in the image. At this point the detection is typically already pretty straightforward, making the addition of dilation redundant. At locations where the expansion of laser pointer pixel clusters would have been particularly useful, specifically when the laser pointer is positioned in the back of the image, dilation did not add any benefit, as the pixels of the laser were not expanded as the required brightness levels could not be reached. Hence, dilation was abandoned as a pre-processing step, but instead used for post-processing (Section 2.6.1.5).

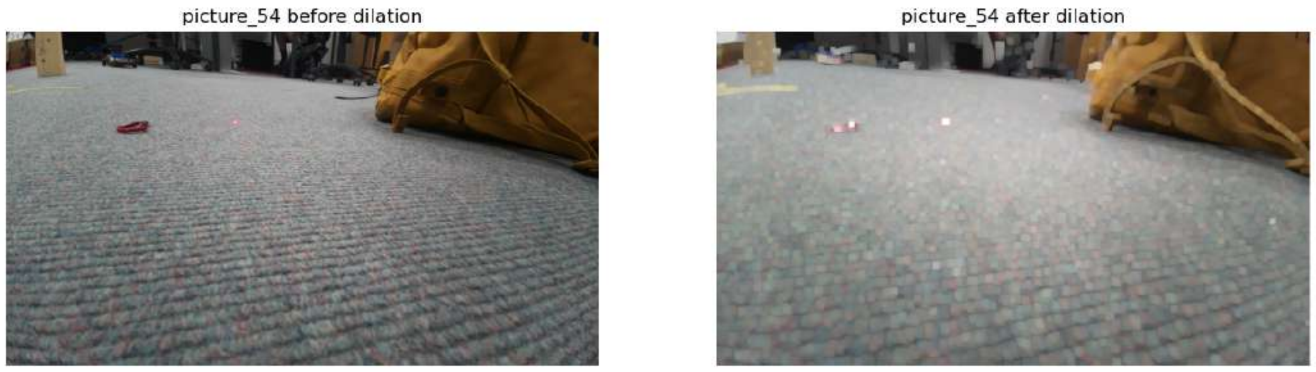


Figure 7: Example of applied dilation

Saturation Increasing saturation results in stronger colour intensity across all pixels. This can be done either programmatically, or, as in case of the camera module used for this project (Section 2.5), also by adjusting camera settings directly. Relying on internal camera settings can help improve systems performance, as this is typically more efficient than running custom code. Increasing saturation proved to be very suited for the task of detecting a laser pointer. In doing so, the laser's unique red colour can be further intensified to achieve a strong red cluster of pixels (Figure 8). This makes it easier for the later detection of the laser pointers position during masking.

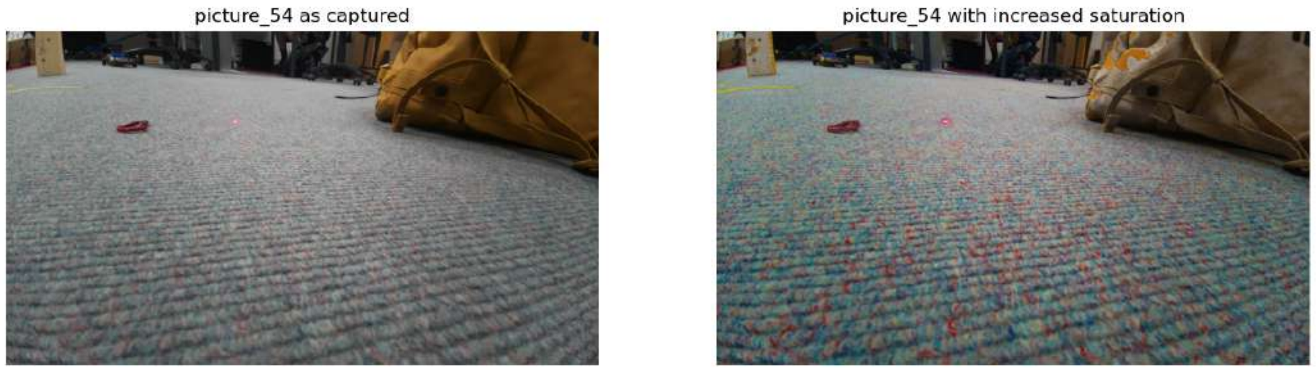


Figure 8: Example of applied increased saturation

Contrast By increasing the contrast of images, the difference between light and dark areas in the frame is intensified. As with saturation, this can be done either programmatically or by adjusting camera parameters, with the latter generally offering better performance. However, increasing contrast did not prove to be effective for improving the detection of the laser pointer's dot. While the laser spot became slightly brighter, the surrounding floor also became more illuminated, reducing the overall benefit. Lowering the contrast did not help either. Although neither adjustment made detection significantly harder, they also did not lead to meaningful improvements (Figure 9). As a result, contrast adjustments were not used further in this project.

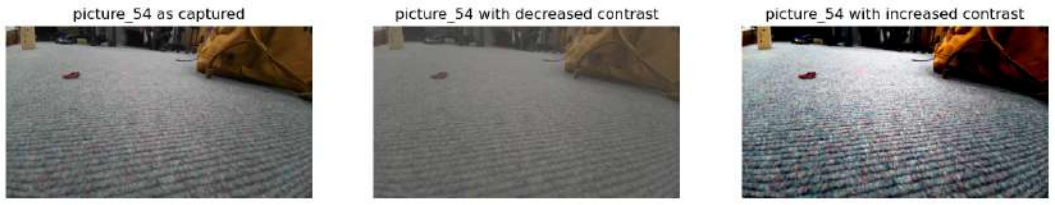

picture_54 as captured
picture_54 with decreased contrast
picture_54 with increased contrast

Figure 9: Example of applied decreased and increased contrast

Exposure Time / Exposure Value Exposure in photography describes the amount of light that reaches the camera sensor. The simplest way of adjusting this is by changing the shutter speed, the time a scene is exposed to light, which is also referred to as exposure time. A shorter exposure time results in a darker image and reduces motion blur. However, if the exposure time is too short, the image can quickly become underexposed, leading to losses of information as the sensor cannot pick up on these areas of the image.

Modern cameras, including the camera module used in this project, are typically capable of setting the exposure time to suitable values automatically. A parameter, the exposure value, can then be used to adjust the target brightness of the image the camera tries to reach. This is beneficial, as it allows for greater flexibility for different levels of lighting in scenes than setting the exposure time purely manually. Therefore, the focus for the tests described here laid on adapting the exposure value, rather than setting the exposure time manually.

The laser dot typically appears in images as a cluster of very bright red pixels. It was therefore intuitive to try lowering the exposure value as much as possible, to ideally isolate the laser dot from the background (Figure 10). However, testing showed that while the dot is very bright when close to the camera, it quickly looses its brightness as it moves further away. Appling exposure values too extreme made the laser dot too dark to detect reliably in such cases. Adjusting the exposure value was hence deemed to be a very effective method, however, must be applied with caution, as it can quickly lead to the laser pointer not being captured in images at all.

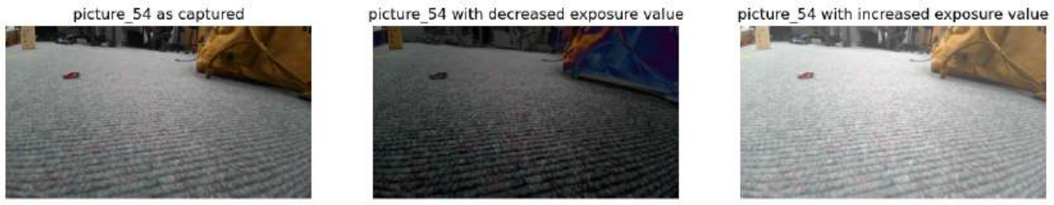

picture_54 as captured
picture_54 with decreased exposure value
picture_54 with increased exposure value

Figure 10: Example of applied decreased and increased exposure values

2.6.1.4 Masking Methods Evaluation

Background Subtraction (Pixel Change Detection) This method works by comparing the current frame with a previously captured ”background” frame, taken before the laser pointer enters the scene. The idea is that any changes between these two images should highlight the laser dot, without relying on its colour or brightness, which can vary drastically under different lightning conditions.

In practice, one image was taken at the start of the test-loop and set to be the background (Figure 11, *picture_1*), with further incoming frames than compared to it. Pixels that changed were marked white (255), while unchanged pixels set to black (0). The results (Figure 11), show the obtained binary mask put on top of the captured frame, with every pixel that changed marked in bright green. It shows laser pointer dots being highlighted correctly. However, it also picks up on a significant amount of noise, such as a moving person in the background, as well as changes of the bag (which did not change in location or shape, but seemed to have induced some sort of noise nonetheless). With this many falsely obtained positives in the mask, heavy post-processing would be required to narrow down which cluster of pixels truly is the laser pointer dot. Hence, this method was not explored further.

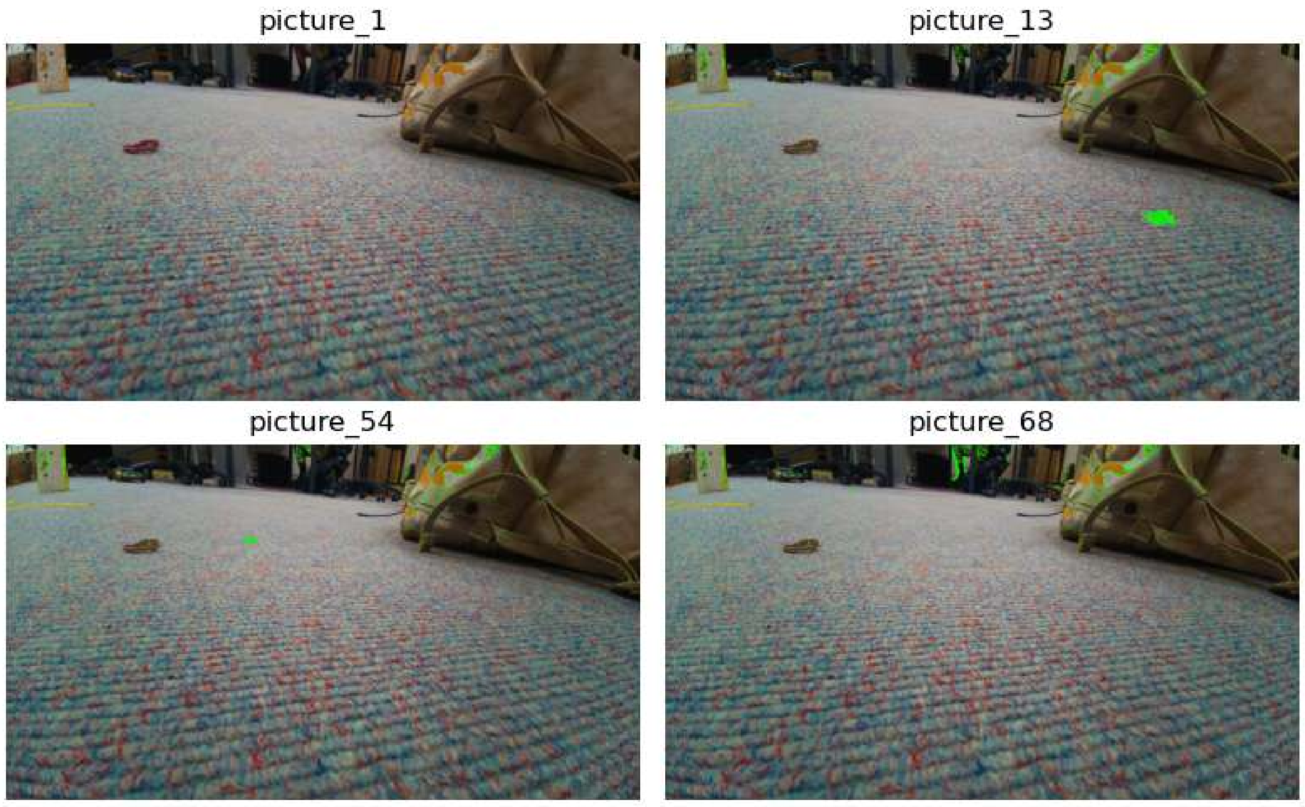


Figure 11: Evaluation of masking using background subtraction

White Top-Hat Transform White Top-hat transformations are typically used to highlight small, bright features in an image. To achieve this, the *Opening* of an input image *I* is calculated, by applying pixel erosion followed by dilation, with parameters defined by a kernel. The *Opening* is then subtracted from the input image *I* as follows:

$$TopHat(I) = I - Opening(I) \tag{1}$$

This was tested using OpenCV's built-in white top-hat function applied with different kernel sizes. However, the masks obtained (Figure 12) did not prove useful. Further, the laser dot size varies greatly depending on its distance from the camera. The camera mounted at a steep angle to the object plane, further complicates this issue, making it borderline impossible to choose a kernel size working well for all cases. Top-Hat transform was hence not seen as a useful binary masking process.

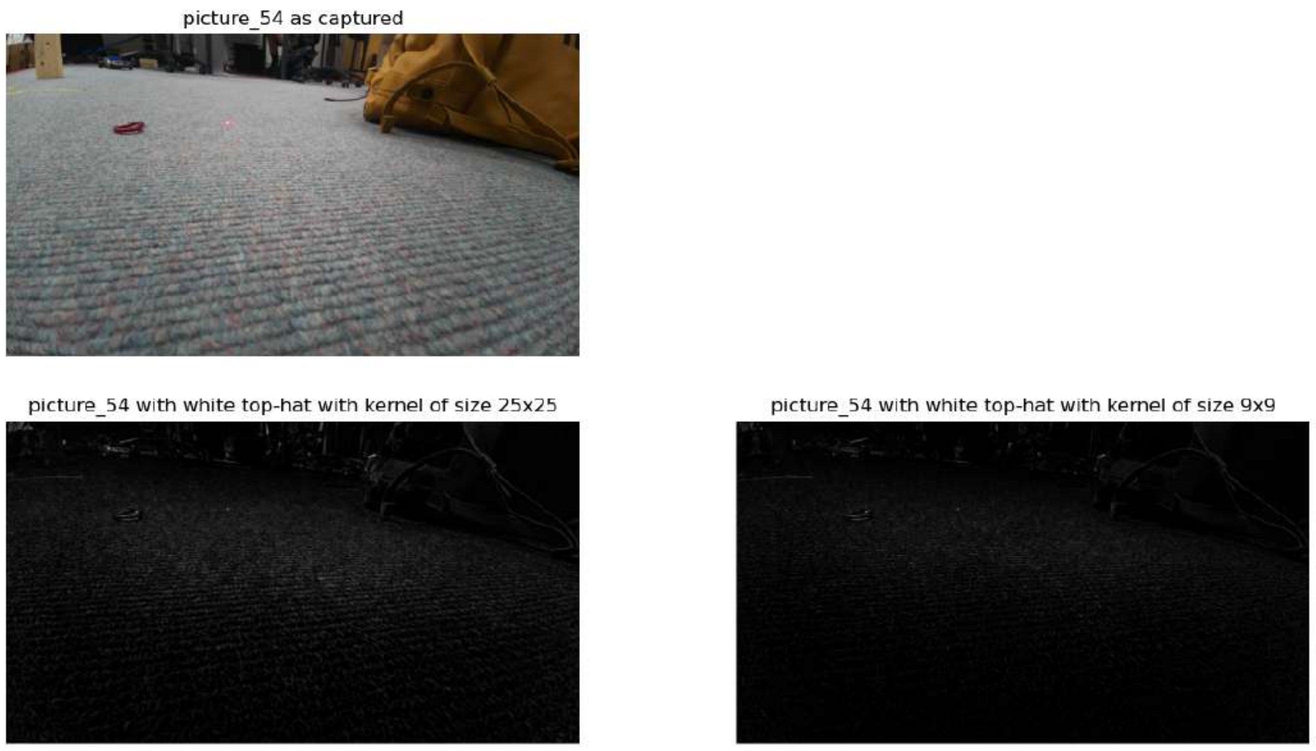

picture_54 as captured
picture_54 with white top-hat with kernel of size 25x25
picture_54 with white top-hat with kernel of size 9x9

Figure 12: Example of applied White Top-Hat transformations with different kernel sizes

Hue-Saturation-Colour (HSV) Colour Masking Every pixel in an image can be defined by its HSV (Hue, Saturation, Value) colour representation. As a laser pointer dot typically appears as a bright, red spot, it made sense to try isolating it by filtering for pixels that are both bright and red. HSV filtering was expected to give better precision compared to RGB filtering, especially with the laser often appearing over-exposed, making brightness-based filtering an effective approach. To do HSV masking, incoming frames were converted from the RGB (or in OpenCV's specific case BGR) colour domain into the HSV colour space. A lower and upper HSV bound were selected to isolate the laser dot, with pixels within the limits marked as white (255) and all other pixels set to be black (0).

This worked reasonably well during testing. However, finding suitable HSV bounds proved to be challenging. One reason for this, was that the appearance of the laser changes significantly depending on its distance from the camera. While being bright and red when close, it becomes quite dim further back. Another reason was, that red values wrap around the hue scale of HSV, meaning red values are present at values of both the lower (0), as well as the upper (255) limits of the scale. Because of this, using a single lower and upper bound was not sufficient and instead, two ranges had to be defined (one from hues of 0 onwards, the other one for hues up to 255), resulting in one binary mask each which were later combined. Nevertheless, HSV colour masking proved to be an effective approach, with obtained binary masks looking pretty much exactly to the ones obtained by RGB colour masking (Figure 13).

Red-Green-Blue (RGB) Colour Masking Individual pixels can not only be defined by their HSV value, but also by their prominence of Red, Green and Blue (RGB) channels. Similarly to HSV masking (see above), colour masking can also be applied in the RGB colour space. To achieve this, a base colour was defined in terms of values for each colour channel. Any pixels within a certain range (tolerance) of these values were marked white (255), all other black (0).

Applying this method to frames with increased saturation, lead to surprisingly reliable results (Figure 13). Most of the background noise is ignored, simplifying post-processing.

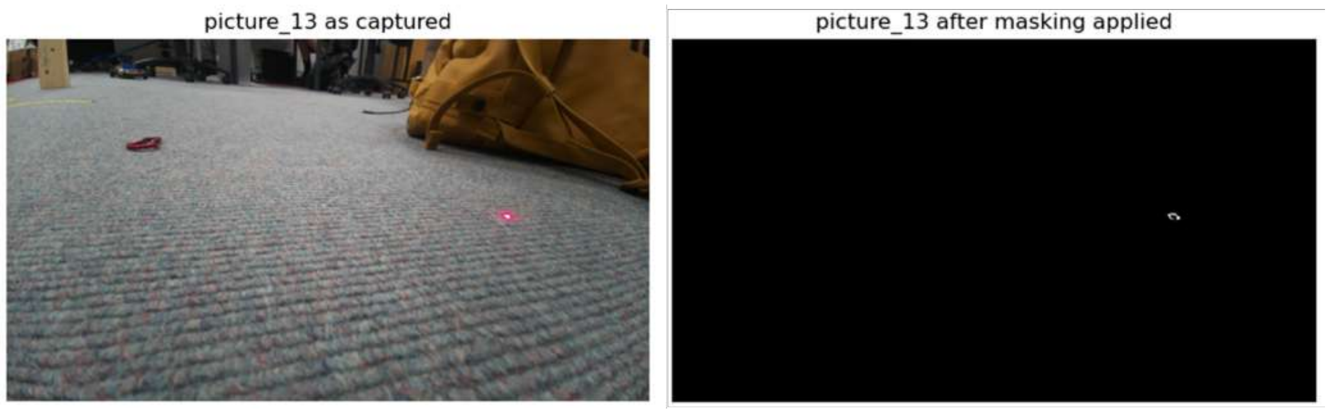


Figure 13: Evaluation of masking using RGB colour matching

2.6.1.5 Post-Mask processing

Closing gaps After masking, it may be useful to close small gaps between pixels that are classified as white (255). To achieve this, a combination of dilation and erosion is typically applied. This approach was tested.

First, a specific kernel size is defined, which is then used for both transformations. Dilation is applied to close small gaps between pixels. As this leads to clusters growing in size also into directions other than towards gaps, erosion is applied subsequently. Erosion shrinks the clusters back down to their initial size while keeping the closed gaps intact. This combination proved to work very well (Figure 14), with adjustments to the kernel size made as required.

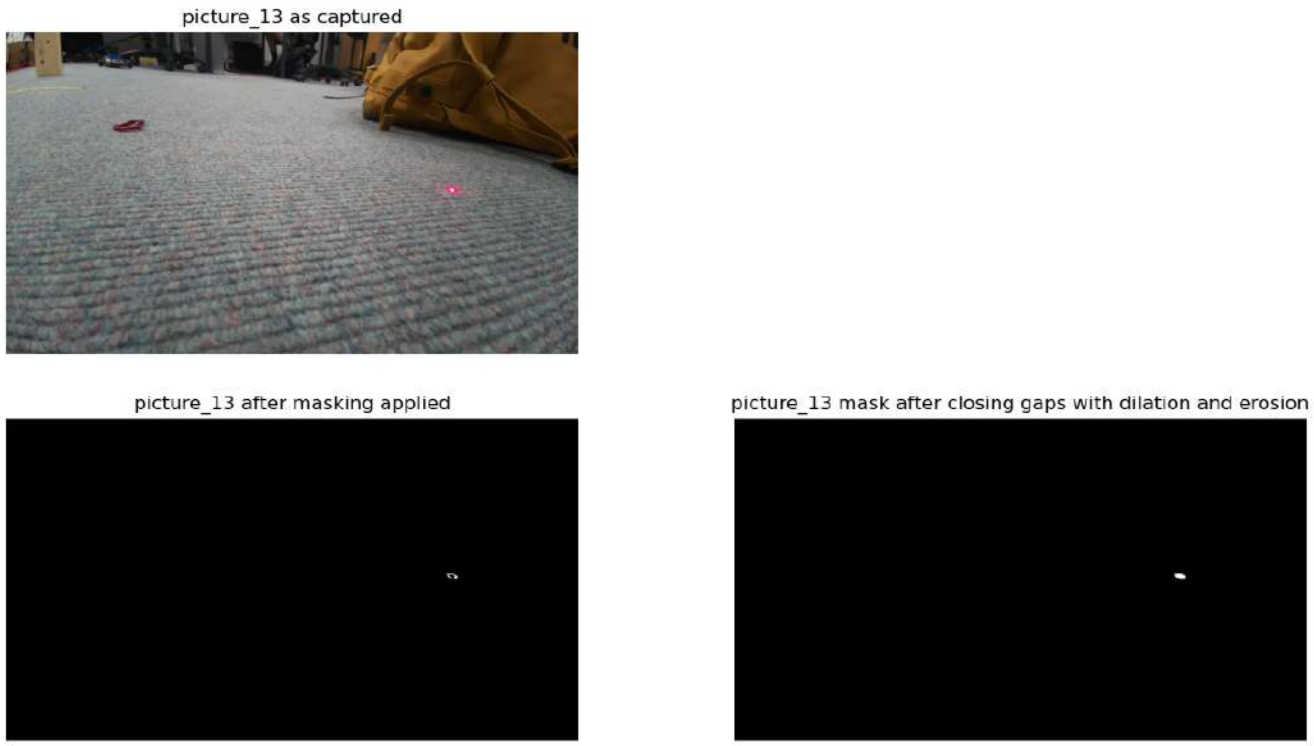

picture_13 as captured
picture_13 after masking applied
picture_13 mask after closing gaps with dilation and erosion

Figure 14: Example of gap closing using dilation and erosion

Aspect ratio/roundness Occasionally masking incorrectly marks clusters that are not the laser pointer dot. However, these clusters do commonly not follow the roundness that can be expected from the dot. Hence, these can easily be further removed from the mask. Only clusters of a certain shape, certain aspect ratio, should be kept. To showcase this, noise was added to a picture in form of a manually drawn red line. As Figure 15 shows, this line is effectively removed after aspect ratio filtering is applied, while the laser pointer dot is kept. It is important to note that the system is not necessarily looking for perfect circles, but rather elliptical shapes. This is due to the distortion of the laser pointer dot as it is projected onto a plane (the floor) that is at a steep angle to the camera (approximately 45° depending on the camera setup).

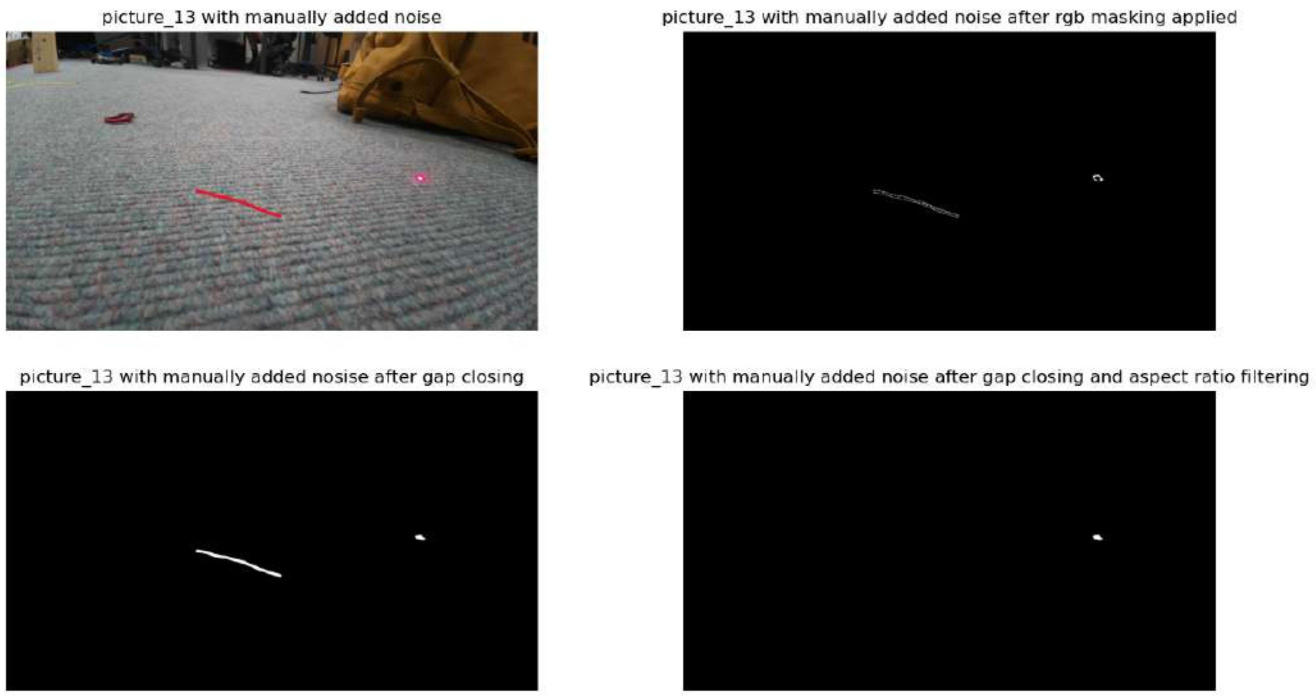

picture_13 with manually added noise
picture_13 with manually added noise after rgb masking applied
picture_13 with manually added nosise after gap closing
picture_13 with manually added noise after gap closing and aspect ratio filtering

Figure 15: Example of aspect ratio filtering

2.6.1.6 Obtaining pixel coordinates After applied filter and decision mechanisms that leave a single cluster in the binary mask that represents the laser pointer, the exact location of this cluster in the picture must be determined. To do this, the contour of the cluster is calculated. The spatial image moments m_{00}, m_{10} and m_{01} - the zeroth-order moment, first-order moment in x-direction and first-order moment in y-direction respectively - are further obtained to calculate the geometric centre (centroid) in terms of x and y coordinates of this cluster. The centroid coordinates (c_x and c_y, for x and y coordinate respectively) are computed as follows:

$$c_x = \frac{m_{10}}{m_{00}},\ c_y = \frac{m_{10}}{m_{00}} \tag{2}$$

These coordinates mark the geometric centre of the pixel cluster present in the image in terms of x and y pixel coordinates.

2.6.2 Pixel to Real-World Coordinate Mapping - Inverse Perspective Mapping

Inverse Perspective Mapping (IPM) is a technique of converting an image taken at an angle, a first-person-view, into a top-down/bird's-eye view of the object plane (Figure 16), in this project's case the ground plane. It can similarly be used for this project, where pixel coordinates, the laser pointer's location in the image, are translated into the real-world distances from the camera.

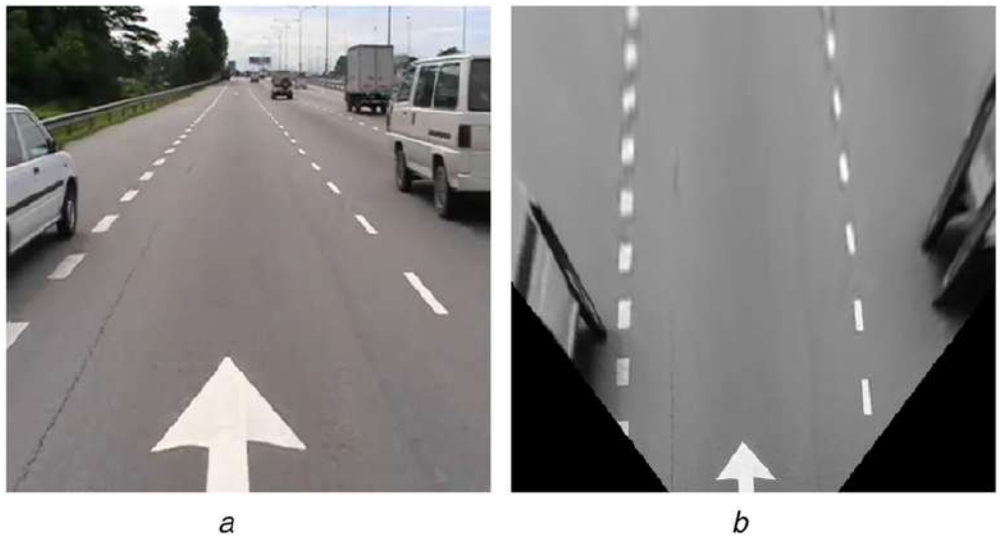


Figure 16: Example of an IPM converting an image from First-Person to Top-Down view [9]

A core requirement of IPM is the presence of a known 3×3 homography matrix H (shown how to be obtained in paragraphs below), which defines the projection between two plane: the image plane and the object plane. To convert a single pixel with coordinates (p_x, p_y) it must first be transformed into a 3-element vector $\mathbf{p}$. This is to allow matrix multiplication with the homography matrix H. The newly added third element is set to 1, resulting in the following vector $\mathbf{p}$ (As explained by OpenCV [7]):

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{3}$$

Next the vector $\mathbf{p}$ is multiplied with the homography matrix H to obtain the homogeneous coordinates $P_{homogeneous}$. Depending on what the homography matrix represents, either mapping from real-world coordinates to pixel coordinates or vice versa, it must be used differently. If it represents a real-world to pixel coordinate mapping, its inverse H^{-1} must be used, if it maps pixel to real-world coordinates the matrix H is used as is. The direction of mapping of H is typically known by the way it was obtained. Hence the next step is:

$$\mathbf{P_{homogeneous}} = \begin{cases} \mathbf{H} \cdot \mathbf{p}, & \text{if } \mathbf{H} \text{ maps pixels to world coordinates} \\ \mathbf{H^{-1}} \cdot \mathbf{p}, & \text{if } \mathbf{H} \text{ maps world coordinates to pixels} \end{cases} \tag{4}$$

To convert the know obtained homogeneous coordinates $P_{homogeneous}$ they must next be normalised by division through their third component to obtain the real-world cartesian coordinates $P_{world} = (X, Y)$. Therefore:

$$X = \frac{\mathbf{P_{world}[0]}}{\mathbf{P_{world}[2]}} \text{ and } Y = \frac{\mathbf{P_{world}[1]}}{\mathbf{P_{world}[2]}}, \text{ with } \mathbf{P} = (X, Y) \tag{5}$$

This gives the position of the laser pointer dot in real-world distances from the camera.

The homography matrix must be updated whenever the position of the camera (height or angle) changes in relation to the object plane. Hence, obtaining the homography matrix in an elegant and easy way was of importance, with positioning of the camera potential for changes as the laser pointer detection is used for other projects. Different ways of obtaining the homography matrix and performing IPM were explored.

2.6.2.1 Obtaining Homography Matrix using Automatic Calibration One of the most elegant methods of obtaining the homography matrix is using OpenCV's native calibration functions using images of checkerboard patterns. For this, pictures of a checkerboard with known parameters, particularly its number of rows and columns and square sizes, are taken. The board is printed onto a sheet of paper and placed on the floor at varying distances and angles, with the checkerboard always lying flat to the object plane (Figure 17). OpenCV's functionalities of `findChessboardCorners()` and `findHomography()` can then be used to compute the homography matrix based on the relation between image coordinates and real-world positions (the distances between the corners of the squares).

However, when attempting this method, the automatic calibration always failed for this project, as OpenCV was unable to detect the checkerboard's corners. Various checkerboard configurations were tested unsuccessfully, including 9x6 and 15x13 patterns printed on both A4 and A3 paper. It is likely, that the steep viewing angle of the camera, its low height above the ground and the high distortion of its wide angle lens prevented the correct detection of corners.

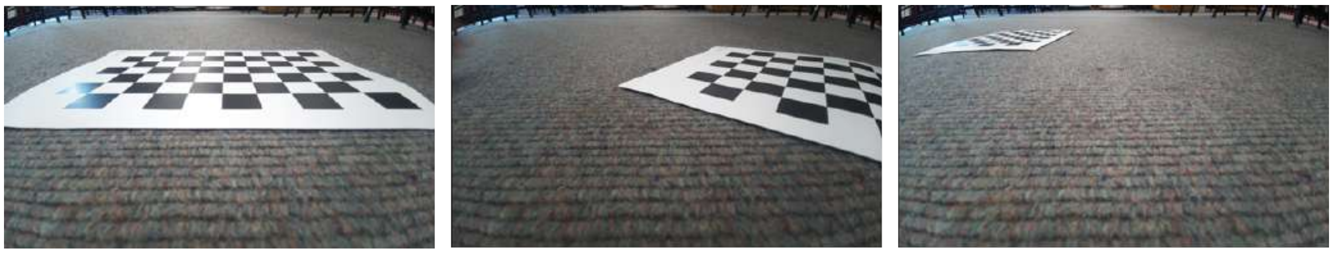


Figure 17: Images of the checkerboard used for automatic calibration using OpenCV

2.6.2.2 Performing IPM using Internal Camera Parameters As the automatic calibration of using checkerboard images failed to produce useful results, instead a different approach was explored. Here, rather than obtaining the homography matrix via images, it is obtained using information about the camera's position and its internal parameters. While the camera pose is known, its internal parameters must first be obtained.

Obtaining Internal Camera Parameters Similar to the automatic homography calibration, camera parameters such as its intrinsic matrix **K** and lens distortion coefficients can be obtained by using native functionalities of OpenCV. Again, images of checkerboards at different positions and angles are taken. However, this time the checkerboard is not placed on the object plane, but rather held directly in front of the camera lens 18. To safe programming efforts, the parameter calculation was done using video2calibration a project shared on GitHub [10]. The parameters obtained where then tested for their suitability by removing distortion from an image taken. This proved that the parameters were quite accurate, while the undistorted result was not perfect, it definitely decreased its amount, especially in the corners of the image (Figure 19).

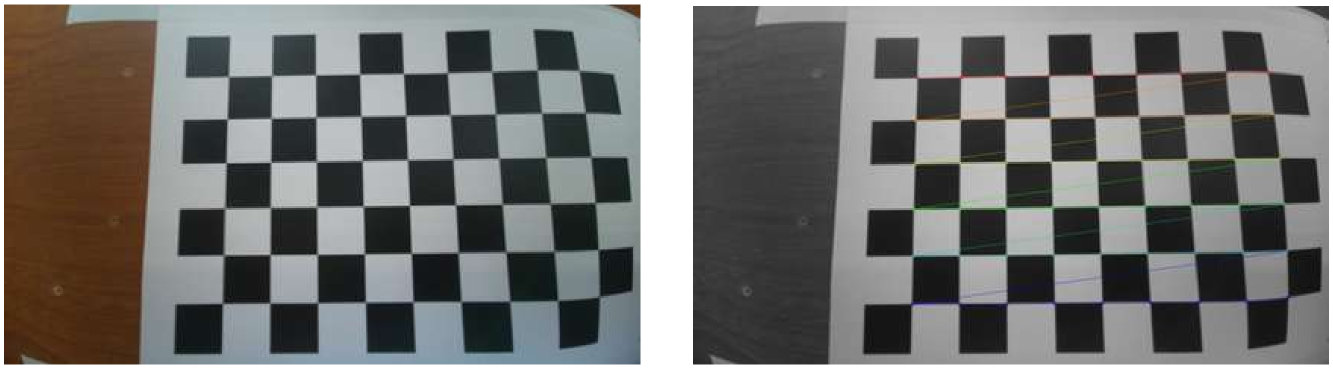

(a) Example image as captured by the camera (b) Results of OpenCVs checkerboard detection algorithm

Figure 18: Process of obtaining internal camera parameters

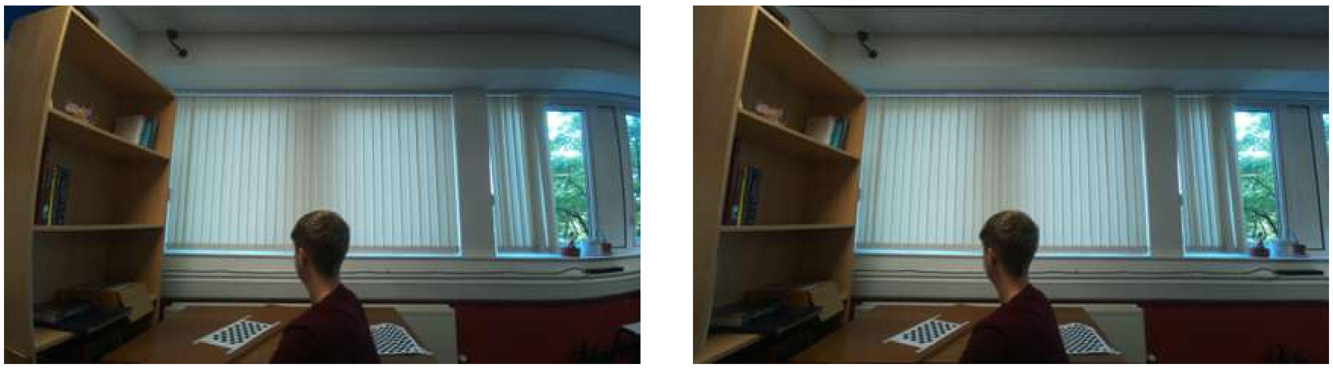

(a) Image as captured by the camera (b) Image after applying undistorted

Figure 19: Visual inspection of obtained internal camera parameters by removing distortion from an image

Using the obtained parameters to perform IPM With these intrinsic camera parameters obtained and the camera pose is known, IPM can be performed. This is done as follows (from the OpenCV Homogeny Tutorial, Demo 3: Homography from the camera displacement [7]):

The camera's intrinsic matrix K is known to be (focal length and principal point):

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

So are the camera's extrinsic parameters (rotation R and translation t, with θ as the tilt angle and h as the height from the ground):

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix} \text{ and } t = \begin{bmatrix} 0 \\ 0 \\ h \end{bmatrix} \tag{7}$$

The homography matrix H is defined as:

$$H = K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \text{ where } r_1 = \begin{bmatrix} r_{11} \\ r_{21} \\ r_{31} \end{bmatrix} \text{ and } r_2 = \begin{bmatrix} r_{12} \\ r_{22} \\ r_{32} \end{bmatrix} \tag{8}$$

The IPM is otherwise calculated as usual to translate pixel values to the real world domain (see above). However, when trying this during testing, the accuracy of the domain translation was unacceptably inaccurate. Therefore this approach was not further considered.

2.6.2.3 Obtaining Homography Matrix using Manual Calibration Since the other IPM approaches did not produce satisfying results, a different method was tested. This time, the homography matrix was calculated by relying on manual definitions of certain pixel to real-world coordinate relationships. This was done by showing a live video feed to the user and asking them to measure specific distances from the camera. The user then clicked on the image to mark points on objects with known real-world positions. This allowed a direct mapping between image coordinates and real-world coordinates to be established, forming the basis for the homography matrix.

This approach turned out to be very accurate. Once calculated, the homography matrix could be saved and reused, meaning that calibration only had to be repeated if the camera's position or angle changed. To make the process easier, a simple and user-friendly GUI was developed. This manual calibration method was ultimately the one used for the remainder of the project.

2.7 Integrated System Overview

A system of a complete laser-pointer-based target localisation was implemented. It detects a laser dot in a video feed and maps its pixel coordinates to real-world distances. It is designed to run on a Raspberry Pi 3B+ [3] with a Raspberry Pi camera module 3 Wide [11]. The implementation relies on classes and callbacks, making it modular to be easily implemented into a larger project. The full codebase can be accessed on GitHub [12]. An outline of the implementation is provided below.

2.7.1 System Architecture

The system consists of two main stage: (1) Calibration, which establishes the pixel-to-world coordinate transformation parameters using user-selected reference points; and (2) Tracking, which continuously detects a laser dot in a video feed and converts its image coordinates into real-world values using the results from the calibration.

2.7.1.1 Calibration Stage Calibration is required to be re-run whenever the camera position or angle changes. A user-friendly Graphical User Interface (GUI) (Figure 20) based around a live video-feed asks the user to click on reference points of known real-world coordinates. The specific instructions are shared in the top-left corner. For all inputs, users are supported by guidance lines added to the images. Previous selected pixels are marked by bright green dots. These inputs are used to compute a homography matrix, which is saved to `build/homography.yaml`. An image with added homography grid is stored at `build/calibration_grid.jpg` for verification of the obtained results (Figure 21). The full calibration process is shown in Figure 22.

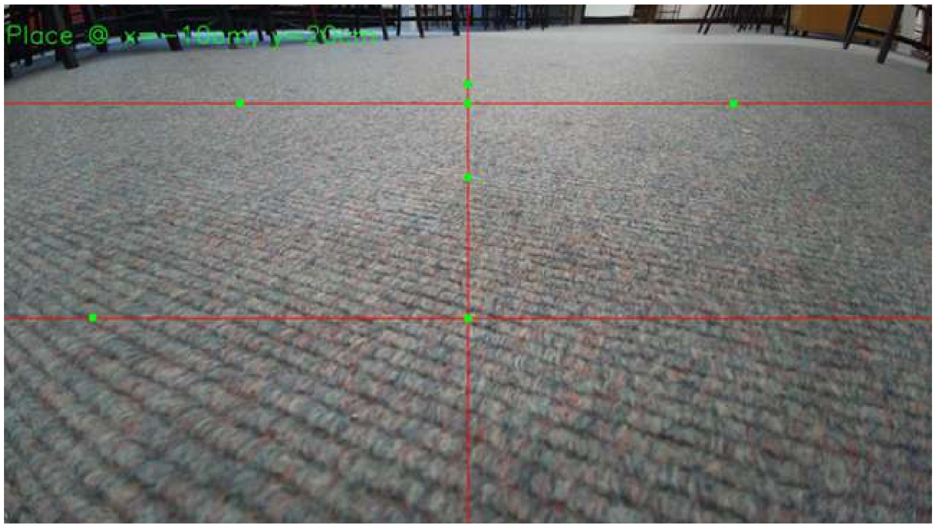


Figure 20: GUI for manual homography calibration

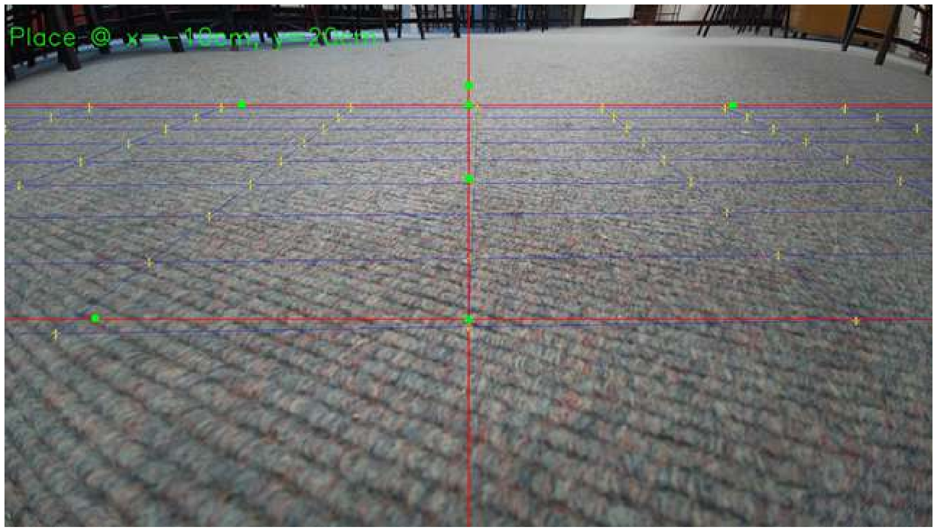


Figure 21: Image with homography grid for visually inspect the calibration for correctness

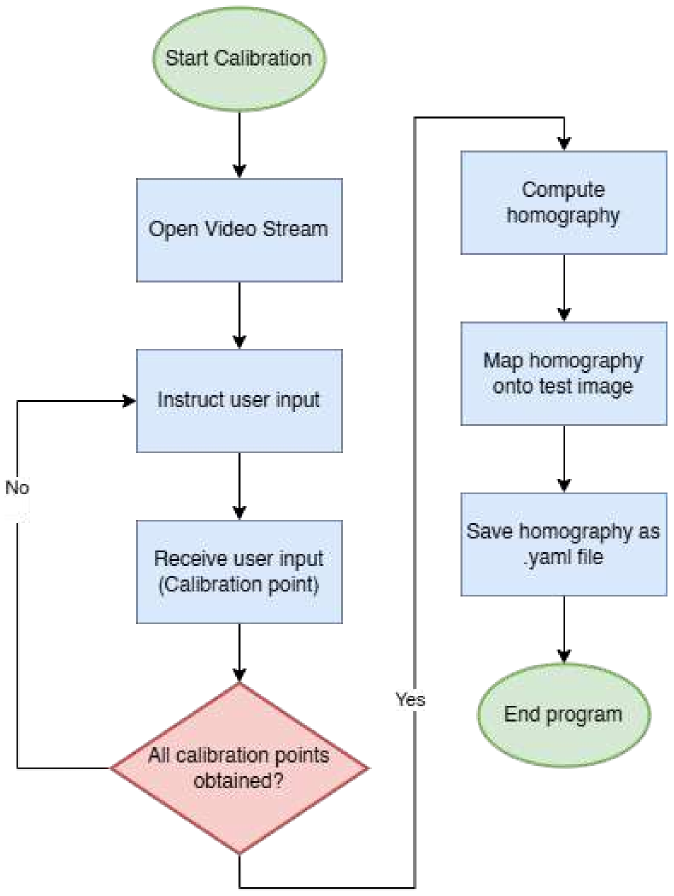

Start Calibration
Open Video Stream
Instruct user input
Receive user input (Calibration point)
All calibration points obtained?
No
Yes
Compute homography
Map homography onto test image
Save homography as .yaml file
End program

Figure 22: Flowchart of the calibration process

2.7.1.2 Tracking Stage Tracking (Figure 22) starts a video stream with specific camera settings (low framerate, increased saturation, lowered brightness) using `libcamera2opencv` [13]. A callback function processes each frame to detect the laser pointer and obtain its real-world coordinates.

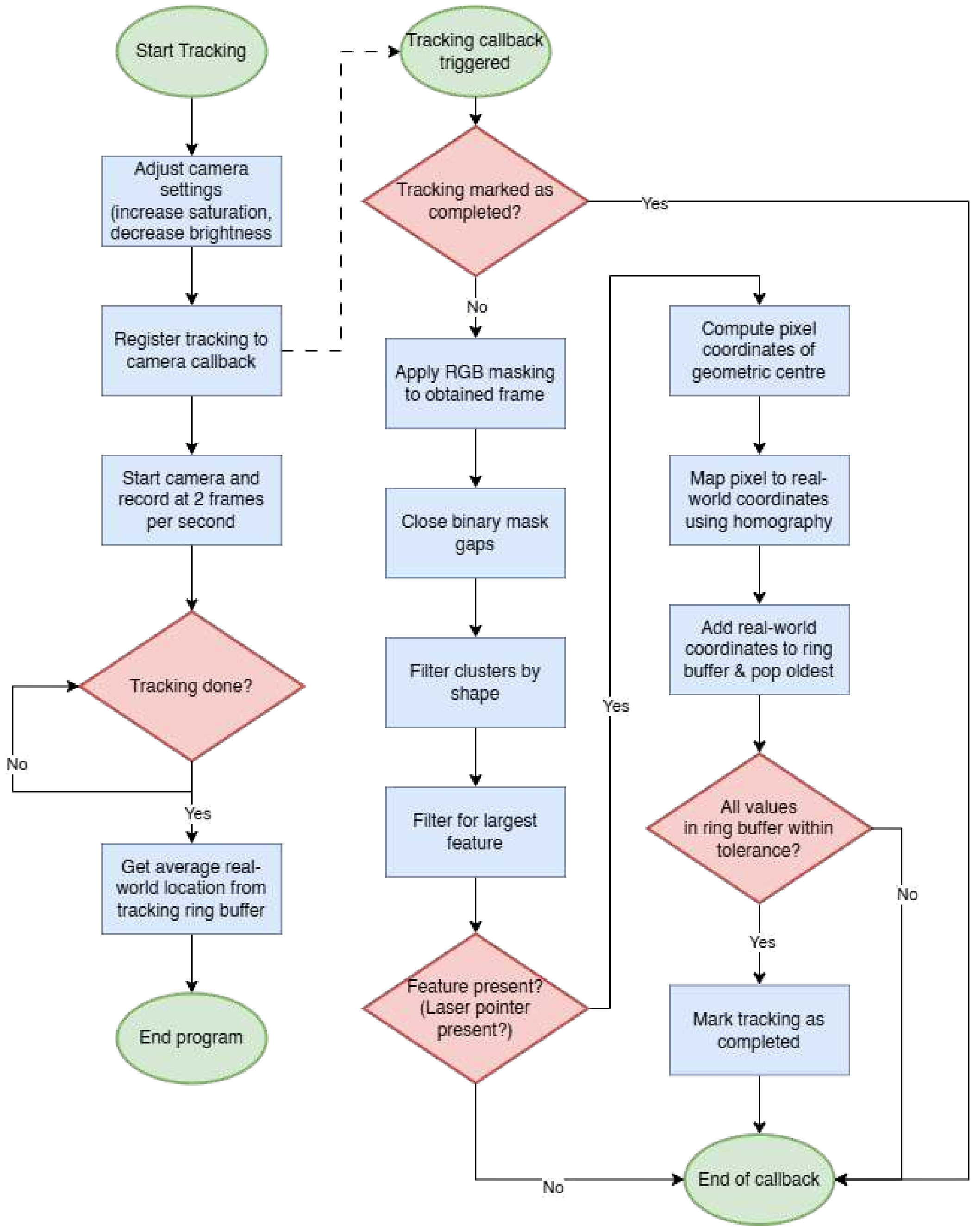

Start Tracking
Adjust camera settings (increase saturation, decrease brightness
Register tracking to camera callback
Start camera and record at 2 frames per second
Tracking done?
No
Yes
Get average real-world location from tracking ring buffer
End program
Tracking callback triggered
Tracking marked as completed?
Yes
No
Apply RGB masking to obtained frame
Close binary mask gaps
Filter clusters by shape
Filter for largest feature
Feature present? (Laser pointer present?)
Yes
No
Compute pixel coordinates of geometric centre
Map pixel to real-world coordinates using homography
Add real-world coordinates to ring buffer & pop oldest
All values in ring buffer within tolerance?
Yes
No
Mark tracking as completed
End of callback

Figure 23: Flowchart of the tracking process

The callback pipeline consists of the following steps (Figure 24):

1. Apply a binary mask based on a target RGB colour and tolerance.
2. Close gaps in the mask.
3. Filter detected pixel clusters based on shape.
4. Keep only the largest cluster.
5. Compute the centroid of the last remaining feature as pixel coordinates.
6. Map the pixel coordinates into real-world coordinates.
7. Store the result in a ring buffer.
8. Complete tracking if ring buffer values are all within tolerance.

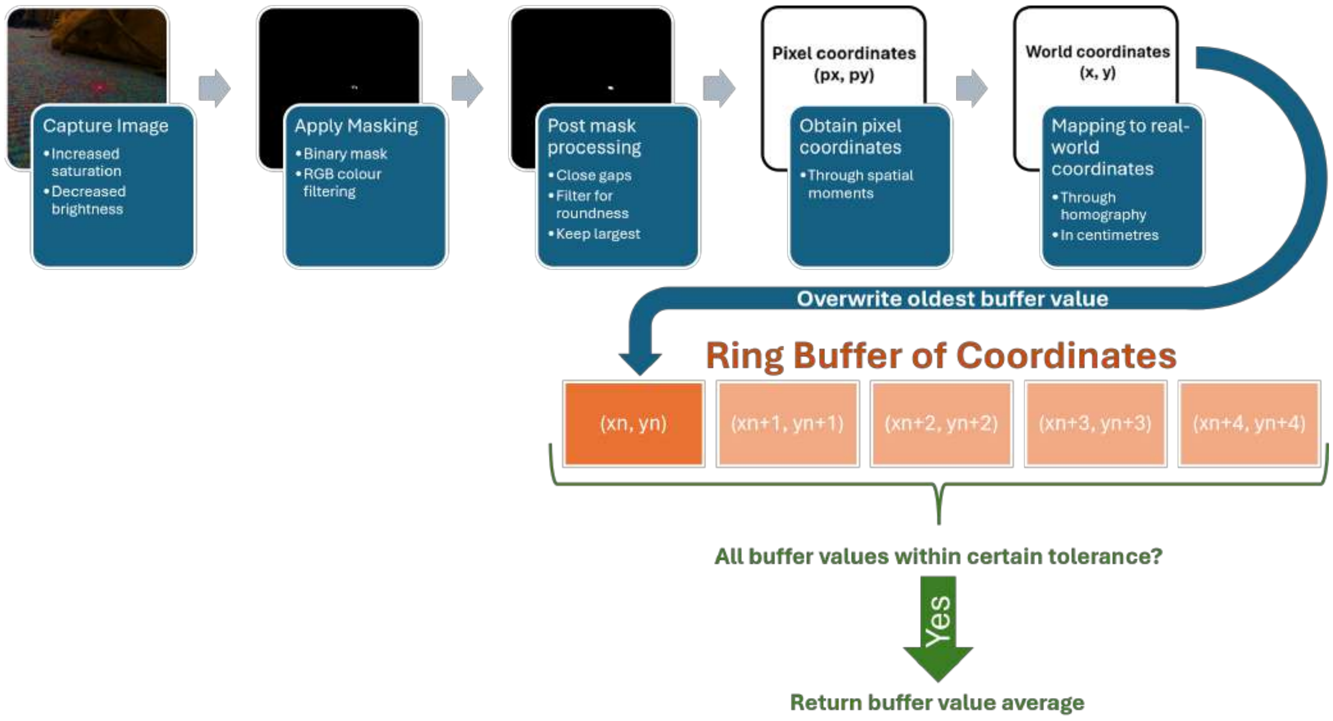


Figure 24: Process of tracking callback handling

2.7.1.3 Tracking Stage Design Rationale Each step in the tracking stage was tested before choosing the most suitable ones (Section 2.6). The following explains the benefits of each design choice (in the order of appearance in the pipeline):

Increased saturation, decreased brightness Saturation is increased and brightness decreased at the points of capturing the image using `libcamera2opencv`'s API [13]. This enhances the visibility of the laser before applying the binary masking transformation. Other processing methods, such as contrast or exposure adjustments were not used as they did not significant benefits.

Framerate of 2 frames per second (FPS) The image processing of tracking takes about 200-300 milliseconds per frame. A framerate of 2 FPS provides sufficient speed, while allowing the algorithm to run in real-time without overloading the processor. Faster framerates did not lead to any direct gains.

RGB Colour masking RGB masking was chosen over HSV masking, moving pixel masking and top hat transforms due to its robustness and ease of tuning. Although HSV is believed to perform comparably, potentially even better when tuned perfectly, achieving optimal HSV ranges was found to be tedious and difficult. The ease of determining RGB ranges using colour picker tools was deemed more advantageous for deploying the algorithm quicker for different environments.

Closing gaps in binary mask Pixel closing using dilation and erosion is applied to remove small holes and noise from the binary mask. It was generally found to improve laser detection when used.

Shape-based feature filtering (Checking for roundness) Occasionally, non-laser features pass the colour mask and must be further filtered. Filtering for expected elliptic shapes of the laser dot proved to be a great way of achieving this.

Filter binary mask for largest feature Rarely, multiple clusters remain in the binary mask. The largest one is typically the correct laser position.

Pixel to Real-World Coordinate Mapping using IPM manual homography calibration The chosen pixel to real-world coordinate mapping is done by IPM in combination with a manually obtained homography matrix. The homography matrix, rather than being obtained automatically by capturing checkerboard images, is calibrated via a GUI with live video-feed to the user, with the user asked to choose pixel coordinates of known real-world locations. This approach was chosen, as it proved to be the most accurate and reliable one of the methods explored.

Ring Buffer A ring buffer is used to store recent detections of laser pointer coordinates. This allows, to only consider positions that are kept steady across multiple frames to be seen as true detections. Steadiness is important to make sure that the obtained location does truly represent the requested target location, rather than capturing a laser dot while simply passing through the image. A small tolerance is applied, to account for shaking hands as the user points with the laser pointer.

2.7.2 Codebase Overview

The full code is shared on a publicly available GitHub [12].

2.7.2.1 Prerequisites

- **Hardware** - Raspberry Pi 3B+ [3] running Raspberry Pi OS Bookworm and a compatible Raspberry Pi camera module (large FoV recommended, such as the Camera Module 3 Wide [11]).
- **libcamera2opencv** - The project uses libcamera2opencv [13] by Bernd Porr [14] for accessing video frames. It relies on modifying the saturation parameter, which is not yet available in the main branch (as of 20/05/2025). Please use the version from the pending pull request [15] or this repository fork [16]. Follow the installation instructions provided in the respective `README.md`
- **OpenCV** - Tested with version 4.9 [17]
- **Compiler** - A C++ compiler is required to build the files, if the pre-built ones from the repository are not used. C++ Version 17 is preferred [18]
- **CMake** - If building the project is required, rather than using the prebuilt executables present in the repository.

2.7.2.2 File Structure

- **calibration.h/calibration.cpp** - Handles manual camera calibration and homography computation.
- **racking.h/tracking.cpp** - Detects the laser pointer and applies the homography to compute real-world coordinates.
- **main_calibration.cpp** - Example program for calibration.
- **main_tracking.cpp** - Example program for laser tracking.
- **build/** - Contains compiled executables and output files such as `homography.yaml`.

2.7.2.3 Classes

- **Calibration** – Guides the user through a live video-stream to the homography.
- **Tracking** – Handles image processing, laser detection, and coordinate transformation.
- **RingBuffer** – Internally used by `Tracking` to store laser dot detections.

2.7.2.4 Example Executables The GitHub repository [12] includes to example executables: `build/Calibrate` and `build/Tracking`.

- `Calibrate` allows user to manually perform the homography calibration by selecting pixel locations of corresponding known real world coordinates.
- `Tracking` performs real-time laser pointer tracking, return the obtained real-world coordinates after detecting a laser pointer at a stable location
- `Tracking --debug` shares a live video feed where the detected laser pointer location is marked with a blue cross. Further, both pixel coordinates and real-world coordinates are printed to the terminal for inspection.

2.7.2.5 Integration Into Other Projects The modular design and use of classes allows for easy integration of the tracking functionalities into further projects (Listing 1). Below is an example of how this can be achieved using the `Tracking`-class and `libcamera2opencv`. It must be ensured, that a valid `homography.yaml` file is present in the project's folder, as required by `Tracking`. Further, laser detection parameters (target RGB colour and tolerance, camera settings) may need to be adjusted to the environment's light conditions as needed. It is recommended, to spawn a separate thread for the `while(!tracking.getTrackindDone())`-loop to avoid blocking and starving other I/Os and threads.

```
struct TrackingCameraCallback : Libcam2OpenCV::Callback {
    Tracking* tracking = nullptr;
    void hasFrame(const cv::Mat &frame, const libcamera::ControlList &)
   override {
        tracking->handleFrame(frame);
    }
};

Tracking tracking(cv::Scalar(255, 0, 0), 70);  // target RGB and tolerance
TrackingCameraCallback trackingCameraCallback{&tracking};

Libcam2OpenCV camera;
camera.registerCallback(&trackingCameraCallback);
camera.start(getTrackingCameraSettings());

while (!tracking.getTrackingDone()) {}

std::cout << "Target at: " << tracking.getTargetLocation() << std::endl;
camera.stop();
```

Listing 1: Example Integration of Tracking

2.8 Results & Evaluation

The developed system successfully detects laser pointers from an image feed of a camera. It manages to transform the obtained pixel coordinates into real-world distances from the camera using a homography matrix and Inverse Perspective Mapping (IPM). Figure 25 shows how laser pointer dots are correctly detected in different locations. Several key performance characteristics were observed (see below). Section 2.10 covers possible steps that could further improve these results.

2.8.1 Processing Speed

The system manages to process every frame captured in approximately 300 milliseconds while running entirely on a Raspberry Pi 3B+. This enables the camera to capture at speeds of 2 frames per seconds while still successfully obtaining real-world coordinates of the laser pointer present in the image.

2.8.2 Detection Range and Reliability

When calibrated accurately, the system can detect laser pointer dots up to a distance of approximately 1.5 metres from the camera reliably in almost every frame. However, for distances beyond this limit, not every laser pointer present is picked up on, with some frames ignoring, others reacting to its presence.

2.8.3 Accuracy of Real-World Mapping

The real-world coordinates calculated from pixel positions are accurate within a few centimetres of distances up to approximately 1.5 metres. This precision is suitable for the required usecase for this project. However, accuracy becomes worse for distances beyond this point, mainly due to the steep angle of the camera.

2.8.4 Confirming Stability of Detected Target

Laser pointers are only reacted to if they are determined to be in a stable state. This is needed, as a laser pointer just entering the image should not immediately be picked up on. Only once the pointer dot is in approximately the same position for a few seconds will the system accept it as the target location.

2.8.5 Intuitive Calibration

The system allows for an intuitive calibration technique using a graphical user interface with users manually selecting known real-world locations. This approach makes it easier to re-calibrate the system as parameters regarding the relationship between camera and object plane change (e.g. if the camera height or angle is changed). Calibration must only be run once, with information stored for every next restart of the system.

2.8.6 Modularity of the Software

The software is designed in a modular fashion, fully relying on classes which can be easily incorporated by wrapping software calling class-functions. This makes it easy for integration of other parts of the project.

2.8.7 Comparison of Various Processing Techniques

The project explored various techniques for laser pointer coordinate detection and mapping the information obtained into real-world coordinate systems. Knowledge gained from unused methods can still potentially become useful as the system is developed further.

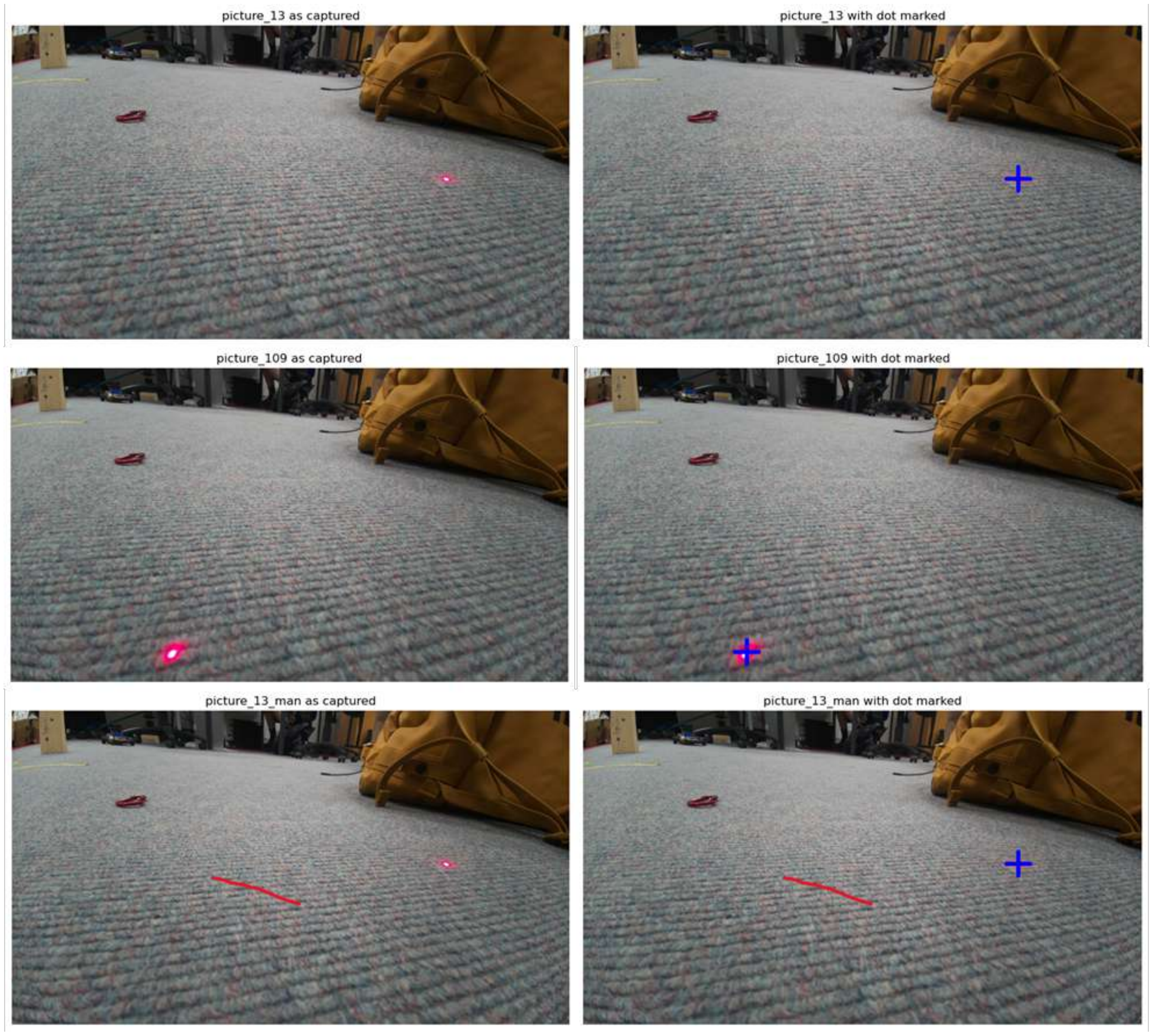

picture_13 as captured
picture_13 with dot marked
picture_109 as captured
picture_109 with dot marked
picture_13_man as captured
picture_13_man with dot marked

Figure 25: Showcase of laser pointer detections. The left side shows images as captured. The right side shows the images with an additional blue cross marking the position of the detected laser pointer location.

2.9 Conclusion

As part of the project a vision-based system capable of detecting a laser pointer in a video stream and mapping its position to a real-world coordinate system using inverse perspective mapping (IPM) was successfully developed. The system operates fully on a single Raspberry Pi 3B+, showcasing that it can run without requiring too much hardware power.

The relationship between camera and object plane must be calibrated manually through an intuitive graphical user interface. Other target values for the laser detection may also require tweaking depending on the light conditions in the surrounding the system is used in. The system generally performs quite well for lasers within a range of 1.5 metres from the camera.

As part of a larger project, the final system was introduced to leading project members as part of a showcase. The general feedback was good. It lays a solid foundation for laser pointer target localisation, with likely sufficient performance. Further improvements are to be implemented as required.

2.10 Suggestions for Future Work

2.10.1 Implementation into wider Context of the Project

As part of a larger project to showcase a newer path-finding algorithm in practice, the laser pointer target localisation sub-system may now be implemented into the wider scope. Through its modular design, it should be possible to do so quite easily. However, migration errors must always be accounted for.

2.10.2 Improving Processing Speed

While the code fulfils its main tasks, the processing time of 300 ms per frame might still cause issues. Further improvements are believed to be possible through code optimisations. Potentially, a switch from a Raspberry Pi 3B+ to a newer generation could also be considered, with latest versions offering much faster processing speeds. Specifically their improved graphical processing units could allow for big leaps in terms of processing time required per frame.

2.10.3 Handling Further Distances

The binary masking to detect laser pointers present in images does not reliably detect them when positioned further than 1.5 metres from the camera. The are present in only about half of the frames. This could be probably be improved a bit by improving the masking parameters. However, a fully reliable solution is yet to be found and could be further explored.

2.10.4 Improving Accuracy

While the system can detect laser pointers at distances further than 1.5 meters from the camera to some extent, accuracy becomes quite problematic. This is mainly due to the steep angle of the camera, which imposes perspective problems on the frames captured. Hence, distances that are quite far apart in the real-world, appear quite close together in terms of pixels (Figure 26). This is inherent to the setup used and cannot easily be resolved without changing the camera's position. However, one improvement could be to use a higher resolution video feed. Currently frames are only captured at half the resolution the camera offers (2304x1296 px) as processing times would grow exponentially. Using a stronger processing unit could allow to process higher resolution images and potentially increase the accuracy of targets at further locations.

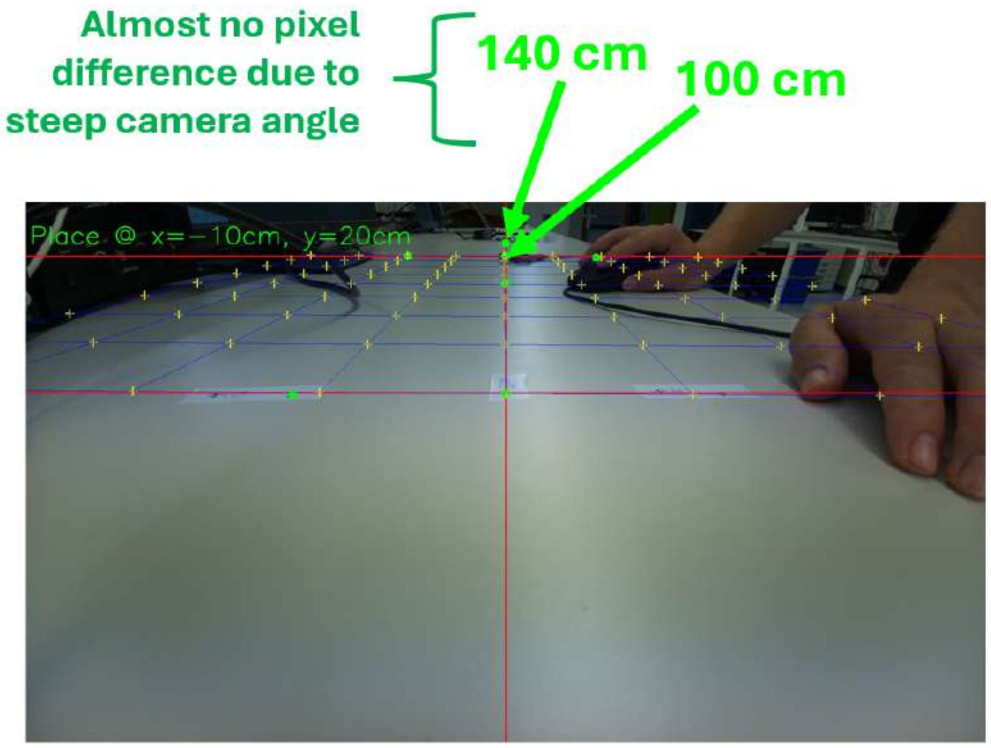


Figure 26: Almost no pixel difference for further distances

References

[1] B. Porr, *Image of the alphabot robot*, `https://github.com/berndporr/alphabot/blob/main/robot.jpg`, Accessed: 19-May-2025, 2023.

[2] Waveshare, *Alphabot-ar bluetooth - mobile robot platform for raspberry pi*, `http://www.waveshare.com/alphabot-ar-bluetooth.htm`, Accessed: 19-May-2025, 2023.

[3] Raspberry Pi Foundation, *Raspberry Pi 3 Model B+*, `https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/`, Accessed: 2025-05-22, 2018.

[4] B. Montgomery, *python-laser-tracker*, `https://github.com/bradmontgomery/python-laser-tracker`, Accessed: 23 May 2025, 2024.

[5] Ankur, *OpenCV-Laser-Tracker*, `https://github.com/Ankur/OpenCV-Laser-Tracker/blob/master/TrackLaser/track_laser.py`, Accessed: 23 May 2025, 2024.

[6] OpenCV, *Opencv: Moments*, `https://docs.opencv.org/master/dd/d49/tutorial_py_contour_features.html`, Accessed: 2025-05-22.

[7] Unknown, *Homography — opencv documentation*, `https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html`, Accessed: 2025-05-22, 2025.

[8] Raspberry Pi Foundation, *Raspberry pi camera documentation*, `https://www.raspberrypi.com/documentation/accessories/camera.html`, Accessed: 20-May-2025, 2024.

[9] M. A. El-Sayed, M. M. El-Sayed, and A. A. El-Sayed, "A Real-Time Inverse Perspective Mapping for Lane Detection," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 10, no. 2, 2019, Figure 2: Inverse perspective mapping. Accessed: 2025-05-22. [Online]. Available: `https://www.researchgate.net/publication/331203551`.

[10] Smidm, *video2calibration: A Python tool for camera calibration from video*, `https://github.com/smidm/video2calibration`, Accessed: 2025-05-22, 2023.

[11] Raspberry Pi Foundation, *Raspberry Pi Camera Module 3*, `https://www.raspberrypi.com/products/camera-module-3/`, Accessed: 2025-05-22, 2023.

[12] R. Nekam, *Laser2world: Ros2 node for laser projection from 3d model to 2d image*, `https://github.com/RaphaelNekam/laser2world/tree/main`, GitHub repository. Accessed: 20-May-2025, 2024.

[13] B. Porr, *Libcamera2opencv: C++ code to pipe from libcamera into opencv*, `https://github.com/berndporr/libcamera2opencv/`, GitHub repository. Accessed: 20-May-2025, 2023.

[14] B. Porr, *Bernd porr's github profile*, `https://github.com/berndporr`, GitHub profile. Accessed: 20-May-2025, 2025.

[15] B. Porr, *Refactoring to use the new picamera2 library: Pull request #1*, `https://github.com/berndporr/libcamera2opencv/pull/1`, GitHub Pull Request. Accessed: 20-May-2025, 2023.

[16] R. Nekam, *Libcamera2opencv_fork: Fork of bernd porr's libcamera2opencv*, `https://github.com/RaphaelNekam/libcamera2opencv_fork`, GitHub repository. Accessed: 20-May-2025, 2024.

[17] OpenCV, *OpenCV: Open Source Computer Vision Library (Version 4.9.0)*, `https://github.com/opencv/opencv/tree/4.9.0`, Accessed: 2025-05-22, 2024.

[18] cppreference.com, *C++17 - cppreference.com*, `https://en.cppreference.com/w/cpp/17`, Accessed: 2025-05-22, 2017.

[19] Y. Liu, C. Wang, H. Wu, Y. Wei, M. Ren, and C. Zhao, *Improved lidar localization method for mobile robots based on multi-sensing*, `https://www.mdpi.com/2072-4292/14/23/6133`, Accessed: 19-May-2025, 2022.

[20] G. F. Steven Alsalamy Ben Foo, *Autonomous navigation and mapping using lidar*, `https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1302&context=cpesp`, Accessed: 19-May-2025, 2018.

[21] H. M., K. O., and T. M., *Lidar positioning for indoor precision navigation*, `https://openaccess.thecvf.com/content/CVPR2022W/PBVS/papers/Holmberg_Lidar_Positioning_for_Indoor_Precision_Navigation_CVPRW_2022_paper.pdf`, Accessed: 19-May-2025, 2022.

[22] H. Yang, L. Wang, J. Zhang, Y. Cheng, and A. Xiang, *Research on edge detection of lidar images based on artificial intelligence technology*, `https://arxiv.org/abs/2406.09773`, Accessed: 19-May-2025, 2024.

[23] B. Hodgson, *Lidar turning repo*, `https://github.com/BaileyHodgson/lidar-based-turningr`, GitHub profile. Accessed: 20-May-2025, 2025.