POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF CONTROL, ROBOTICS
AND ELECTRICAL ENGINEERING

INSTITUTE OF ROBOTICS AND MACHINE INTELLIGENCE

DIVISION OF CONTROL AND INDUSTRIAL ELECTRONICS

USING A TWO-DIMENSIONAL BINARY MATRIX TO DETERMINE POSITION BY MARKING WITH A LIGHT POINT

BACHELOR'S ENGINEERING THESIS

ROLLAND THEODORE, 151250
ROLLAND.THEODORE@STUDENT.PUT.POZNAN.PL
ROBEL MASSEBO, 153922
ROBEL.MASSEBO @STUDENT.PUT.POZNAN.PL

INSTRUCTOR:
DR HAB. INZ. KONRAD URBAŃSKI
KONRAD.URBANSKI@PUT.POZNAN.PL

POZNAN 2025

ABSTRACT

This thesis which we have done shows and describes the design and implementation of a real-time laser tracking system using image processing techniques to interact with a binary matrix. The project uses Python program and the main scientific library OpenCV to process video input from a camera, detect grid corners, and apply perspective transformation to align the matrix for accurate analysis. A laser detection mechanism isolates the laser pointer in varying lighting conditions and tracks its position with high precision, mapping it dynamically to corresponding grid cells. The system incorporates morphological transformations to reduce noise and improve detection accuracy while addressing challenges such as lighting variability, matrix alignment, and laser stability. Performance testing demonstrates the system's capability to process frames efficiently and deliver responsive results in real-world scenarios. This work highlights a practical application of computer vision, offering a foundation for interactive systems in education, entertainment, and beyond.

ACKNOWLEDGEMENTS

First and foremost, we give all glory and honor to God Almighty for His love, grace, and unwavering guidance in our lives. Through the blessings of Jesus Christ and the comfort of the Holy Spirit, we have found strength and purpose to accomplish this work.

We would like to express our deepest gratitude to our supervisor, **Dr. hab. inż. Konrad Urbański,** for his invaluable guidance, constructive feedback, and continuous support throughout this project. His expertise and encouragement have been instrumental in shaping the development and completion of this thesis.

We also extend our gratitude to our colleague **Mateusz Korzeniewski** who has helped us a lot during our project phase, our sincere appreciation. Your kindness and encouragement will forever be cherished.

We are thankful to our colleagues, instructors, and everyone at the **Faculty of Control, Robotics, and Electrical Engineering, Poznan University of Technology**, for their support and inspiration during my academic journey.

Robel Massebo:

I extend my heartfelt thanks to my family for their unconditional love, prayers, and encouragement. Their support has been my foundation, enabling me to persevere even in the most challenging times.

Rolland Theodore:

I also extend my heartfelt thanks to my mother, **Mrs. Akpaka Felicia Ijeoma**, whose unwavering love, sacrifices, and constant motivation have been my driving force. Her belief in my abilities has given me the strength to persevere and achieve this milestone.

CONTENTS

FIGURES

1. INTRODUCTION

1.1 THESIS ORGANIZATION

In this thesis, the tasks were divided through a collaborative effort to ensure a balanced and efficient workflow.

Robel Massebo contributed to the following chapters:

- Chapter 1
- Chapter 2
- Chapter 4 (collaboratively written)
- Chapter 5

Rolland Theodore contributed to the following chapters:

- Chapter 3
- Chapter 4 (collaboratively written)
- Chapter 6
- Chapter 7

We have written chapter 4 together in collaborative effort due to the broad scope of the chapter.

1.2 PROJECT OVERVIEW

We use the term position tracking in various technological fields, like robotics, gaming, interactive systems and augmented reality. By using OpenCV for image processing, the system can identify and track the laser dot in real time, matching its position to the corresponding grid in the matrix. In this thesis, we introduced a new method for position tracking that combines a two-dimensional binary matrix with laser point detection.

This approach uses a readily available tool which is a laser pointer with a specially designed binary matrix to create an efficient and adaptable tracking system. The binary matrix allows for precise segmentation for positional identification, while the laser point acts as a clear and dependable marker within the system.

The proposed solution addresses key challenges like lighting variations, perspective distortion, and noise interference by ensuring reliable and accurate performance. It is designed to be user-friendly, flexible, and scalable, making it applicable in diverse areas like robotic navigation, interactive technologies and augmented reality.

1.3 BACKGROUND AND MOTIVATION

The project gets its inspiration from the efficiency and simplicity of barcode scanners, which works with large amounts of information through serial codes. By looking at how such systems works in shops, inspired the idea of creating a similar accessible solution for position tracking.

Binary matrices are particularly suitable for these applications due to their high contrast and structured design. Previously, traditional position tracking systems mostly rely on costly hardware, like infrared sensors or GPS in order to have a higher accuracy but advancements in computer vision have enabled cost-effective alternatives that can work with simple tools like webcams and solutions based on software.

By combining a binary matrix with a laser pointer and getting the help of OpenCV for real-time processing, this project explores how structured patterns and simple tools makes robust and affordable tracking systems. This approach helps by solving the issues to have a cost-effective, scalable technologies for interactive applications in fields like education, robotics, and augmented reality.

We frequently use a laser pointer for presentations and to simply highlight objects. Now we use that simple and affordable tool to accurately track movement. This project makes this possible by combining a laser pointer with a specially designed grid.

***Figure 1.1**: Example of a barcode scanner using a laser pointer on a binary matrix for position detection --------- Ref (9)*

1.4 PROBLEM STATEMENT

The project works in order to solve the challenge of creating an affordable and scalable tracking solution. We combine the binary matrix with a laser pointer, the project then uses structured patterns and real-time image processing to get a versatile system for precise position tracking in diverse environments.

Modern position tracking systems that are present today like GPS and barcode scanners, are mostly used due to their efficiency and accuracy. Although, these systems rely on expensive and a bit sophisticated hardware, making them less accessible for small-scale applications or educational purposes.

***Figure 1.2:** Close-up of a laser pointer scanning a barcode with the laser focused on a serial line. --------- Ref (10)*

1.5 Objective and Scope

The system works to provide precise positional data by addressing challenges like noise interference, lighting variability and perspective distortion.

The main objective of this project is to design and implement a cost-effective system that can detect and track a laser pointer on a binary matrix in real time. Develop a real-time system to detect a laser pointer on a video feed and map its position onto a 5x5 binary matrix grid, and then flattened using Python and OpenCV.

Scope:

- To develop a binary matrix with high contrast and structured segmentation to get an accurate position mapping.
- Using OpenCV for real-time image processing which includes perspective correction and laser detection.
- Extracting and visualizing the grid as a binary 5x5 matrix, and then flattening it to get the kernel code.
- Real-time processing of video frames to ensure smooth tracking and stability.
- Testing the system in various diverse environmental conditions to ensure its scalability and robustness.

Figure 1.3: *Laser pointer emitting a red dot, illustrating precise position marking --------Ref (11)*

This project uses traditional computer vision techniques instead of advanced machine learning methods. This keeps the balance between efficiency and simplicity which also makes it accessible for other applications like gesture-based controls, interactive displays and navigation aids.

1.6 Relevance of Binary Matrix and Laser Point Detection

Laser point detection provides an easily identifiable and reliable marker that can be tracked in real-time using simple image processing technique. By combining binary matrix to it, makes the system an efficient and cost-effective alternative to traditional tracking methods, providing scalability for various applications like interactive learning environments, robotics and augmented reality.

Binary matrices are effective tool for position tracking due to their predictable structure and high contrast. This feature makes it easy to differentiate between distinct locations. The cells found in the matrix has a corresponding specific position which enables precise mapping of a laser pointer's position.

2 LITERATURE REVIEW

This chapter lists challenges in laser-based detection such as environmental noise, variable light, and binary grids alignment. The aim is to provide a comprehensive understanding of the methods and technologies used in this project. It also talks about relevant theories, techniques, and existing research that form the foundation for this project. We explore position tracking systems, binary matrices, and laser point detection in optical tracking.

2.1 OVERVIEW OF POSITION TRACKING SYSTEMS

This project focuses on camera based optical tracking by using a simple and cost-effective system. By combining these components with real-time image processing using OpenCV, it can offer an accessible and scalable solution for precise position tracking. This is suitable for interactive applications in education, robotics and AR.

We use position tracking systems for various applications like gaming, robotics, navigation and augmented reality. They utilize algorithms and sensors in order to track markers and objects in real time. There are common types of position tracking systems such as:

- **Camera-Based Optical Tracking Systems**: Optical tracking systems uses camera to detect visual markers, objects and patterns. They offer precise and a cost-effective way for tracking, mainly in indoor environments. The challenges addressed by these systems includes:
  - **Environmental Variability:** Lighting changes and reflections can impact detection accuracy.
  - **Computational Delays:** Processing video frames in real time needs robust computational resources and an efficient algorithm.
  - **Alignment Issues:** Position and camera orientation can affect the quality of captured images.
- **Infrared and Ultrasonic Systems**: This can offer higher precision but requires a specialized hardware. It is vulnerable to interference from environmental factors such as reflective surfaces.
- **GPS-Based Systems**: In outdoor areas GPS system gives us an accurate position data, but they lack the precision required for indoor applications. This makes GPS system unsuitable for robotics, AR and other related streams.

2.2 BINARY MATRICES IN OPTICAL TRACKING

Binary matrices can be easily identified and processed by image detection algorithms due to high contrast and their predictable structure. Binary matrices consist of cells and all the cells are assigned to a value of either `0` (black) or `1` (white) which provides a clear distinction of individual locations. This makes that a highly effective tool for optical tracking.

Two common types of binary matrices used in optical tracking are:

- **Checkerboard Patterns**: Checkerboards are used for camera calibration. It has a simple geometric structure and can be easily detected due to their black and white squares. It is nice tool for estimating camera parameters like distortion coefficients, focal length, and the optical center.

- **QR Codes**: QR codes are used for position tracking due to their reliability and robustness. They encode information in a two-dimensional grid. It also has an error-correction capabilities to handle partial data loss.

This thesis uses a **two-dimensional binary matrix** as a grid for position detection. The grid provides structured segmentation which makes the laser dot's position to be mapped accurately to specific cells. The binary matrix design's simplicity makes it reliable for interaction with the laser pointer and easy to process in real-time.

Relevance to Laser Detection:

We combine the features of a high-contrast binary matrix with the laser pointer. This helps us to get an effective way to track position.

- The laser pointer serves as an identifiable marker which is clear to be detected in the camera feed using simple image processing techniques.
- The binary matrix ensures reliable detection, even under variable environmental conditions, which makes it preferable over other tracking methods.

OpenCV facilitates binary matrix-based tracking by offering functions for thresholding `cv2.threshold` and adaptive processing `cv2.adaptiveThreshold`, enabling dynamic adjustments to environmental changes.

The image illustrates a variety of barcodes and QR codes, each designed for specific purposes.

Barcodes are simpler and scan linearly as shown below in **Figure 2.1** They are one-dimensional (1D) codes made up of parallel lines and spaces of different widths. It is used for storing alphanumeric and numerical data. We use them for product identification in different fields.

Figure 2.1*: Example of a barcode showcasing the utility of binary matrices in optical systems --------- Ref (12)*

The barcodes shown in **Figure 2.1** differ in structure and format, each catering to specific use cases. For instance:

- **Top Row (Far Left and Right)**: These kinds of barcodes are longer and structured densely which is helpful and used for storing more data.
- **Middle Row (Left)**: This is a compact barcode with fewer lines than the previous one. It is favorable for smaller things where space is limited.
- **Middle Row (Right)**: This is slightly longer than its counterpart. It balances readability and data capacity.
- **Bottom Row (Far Left)**: This kind of barcode is more stretched horizontally. It can be used where scanning from a distance or specific orientation is necessary.
- **Bottom Row (Far Right)**: This is a combination of compact design and moderate data density. It can be used in general purpose applications.

QR (Quick Response) codes store information in both horizontal and vertical dimensions as shown in **Figure 2.2** making them important tool for greater data capacity and flexibility. They are two-dimensional (2D) which can store more complex data, such as multimedia links, URLs and text. It is used in various applications like mobile payments and advertisements.

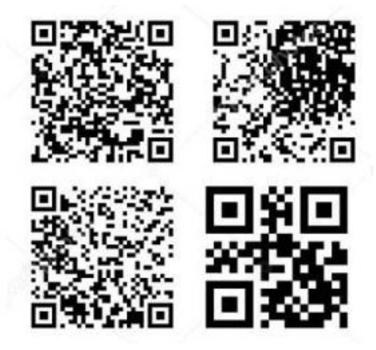


***Figure 2.2**: QR code used as an advanced binary matrix for data storage and position tracking, showcasing the utility of binary matrices in optical systems --------- Ref (12)*

The QR codes shown in **Figure 2.2** tells us the variations in design and complexity, reflecting different purposes:

- **Top Left**: This is a dense Quick Response code. It has a high data capacity that can store large amounts of information such as text and URL.
- **Top Right**: This one appears with errors or more distorted. It shows resilience to partial obstruction and damage with an error correction capability.
- **Bottom Left**: This one is a slightly less dense design. It can be used for quicker scanning.
- **Bottom Right**: This is a simpler structure. It can be used for storing minimal data like a short identifier or single command.

2.3 Challenges In Laser-Based Detection

There are several challenges that need to be tackled during the process of this project. This includes:

- **Environmental Noise:** Other red objects, reflective surfaces or scattered light can create false positives which interferes with accurate detection.
- **Lighting Conditions:** Variations in lighting, reflections or glare can obscure the laser point which makes it difficult to detect accurately.

- **Hand Movement Stabilization:** Hand movements while holding the laser pointer can cause the dot to fluctuate, leading to inconsistent tracking.
- **Matrix Alignment:** Ensure that the binary matrix appears straight and aligned in the webcam. It requires perspective correction techniques and advanced calibration.
- **Tracking the Laser Point:** Tracking the position of the red dot is difficult due to its movement and instability in the cells. The dot's position has to be detected accurately when stabilization occurs.
- **Dot-to-Cell Calibration:** Matching accurately the size of the laser dot to a matrix cell requires precise calibration to have a correct detection.
- **Detection of Black Cell as White Cell:** When the laser intensity is too high, the light can reflect off a black cell, making it appear white. This leads to misclassification in binary thresholding.
- **Detection Difficulty in White-Only Regions:** In areas of the matrix filled entirely with white cells, it becomes challenging to distinguish the laser point from the background, especially under bright lighting conditions.

Addressing these challenges involves implementing solutions such as:

- **Image Preprocessing for Noise Removal:** we apply morphological operations `cv2.morphologyEx` and Gaussian blurring `cv2.GaussianBlur` to eliminate interference.
- **Dynamic Thresholding:** we use `cv2.inRange()` to adapt colour segmentation to ambient lighting changes.
- **Stabilization Delays:** for minor hand movements before confirming a laser point's position, `time()` is applied. We monitor the red dot’s movement across the cells and capturing its position once it remains stable for a few seconds
- **Perspective Transformation:** We use the homography transformation `cv2.getPerspectiveTransform()` to ensure proper matrix alignment.
- **Bitwise Masking for Laser Detection:** Applying `cv2.bitwise_or()` to combine multiple binary masks for detecting red hues in the laser dot, ensuring comprehensive detection across its full color spectrum.
- **Adaptive Brightness Handling:** To prevent high-intensity laser reflections from altering binary classifications, we capture multiple frames before processing, establishing a stable reference to reduce misclassification of black cells as white.
- **Contrast Enhancement in White-Only Regions:** Adjusting image contrast or using a secondary detection method (such as edge detection) to distinguish the laser point in fully white areas.

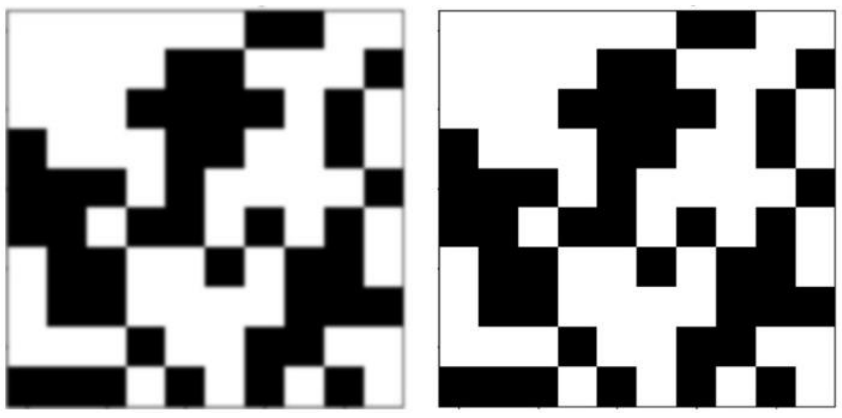


Figure 2.3: *Application of Morphological Operation (Gaussian Blurring) for Noise Reduction in Laser Detection --------- Ref (13)*

Perspective Alignment with OpenCV:

- We get the detected the matrix's corners by analyzing contours in the camera feed. Contours are approximated to polygons using `0.02 * cv2.arcLength()` and `cv2.approxPolyDP()` making an accurate corner detection even under noisy conditions.
- **`cv2.getStructuringElement()`**: This makes kernels for morphological operations that refine contours and improve corner detection.

These functions allow the system to dynamically adapt to environmental changes and ensure matrix alignment, mitigating the effects of noise and distortion. The transformation aligns the grid to a top-down view, mitigating distortions caused by camera angle.

2.4 Laser Point Detection Techniques

Laser point detection relies on isolating the laser dot from the camera feed and mapping it to the binary matrix. OpenCV simplifies this process through a series of operations. These techniques, implemented for detecting the laser pointer ensure robust tracking of the laser dot under dynamic conditions:

- **Contour-Based Localization**:
  - `cv2.findContours()`: detects all closed shapes in the filtered image
  - Contour Filtering: the largest contour, corresponding to the laser dot, is selected for analysis.
- **Centroid Marking:**
  - Centroid Calculation: the precise center of the laser dot is computed using `cv2.moments()`, which calculates the geometric center of the contour. After we have the centroid of the laser dot, it is highlighted for debugging or visualization.
  - `cv2.circle()`: draws a marker at the detected position to confirm successful detection.
- **Colour Filtering**:
  - The laser dot is isolated by filtering its colour range in the HSV colour space using `cv2.cvtColor()` and `cv2.inRange()`.
  - `cv2.bitwise_or()` combines masks for lower and upper red hues to ensure the laser is detected regardless of slight color variations.
- **Positional Mapping:** After detecting the laser dot, its position is mapped to the binary matrix for interpretation.
  - `cv2.perspectiveTransform()`: Ensures that the laser dot's coordinates are accurately aligned to the matrix, even if the camera view is at an angle or distorted.

2.5 Role of Camera Parameters in Optical Tracking

Camera parameters, including **aperture size, focal length**, and **sensor resolution** has an impact on the performance of optical tracking systems. The focal length, in particular, determines the scale and perspective of the captured image.

A. Focal Length and Image Perspective:

A longer focal length reduces distortion but limits the field of view, requiring precise positioning of the camera relative to the matrix. In the other hand, a shorter focal length provides a wider field of view but may distort the grid, making it harder to align the binary matrix.

The system employs camera calibration techniques to compute the camera's intrinsic parameters, as demonstrated in **Figure 2.4**. This ensures that the perspective transformation correctly aligns the matrix for laser point detection. This image illustrates how different focal lengths influence the perspective and alignment of the binary matrix as captured by the camera. A shorter focal length creates distortion, while a longer focal length ensures accurate alignment but reduces the field of view. Functions such as `cv2.getPerspectiveTransform()` and `cv2.warpPerspective()` rely on intrinsic camera parameters like the focal length. It corrects distortions caused by a shorter focal length or it will refine the alignment for a longer focal length.

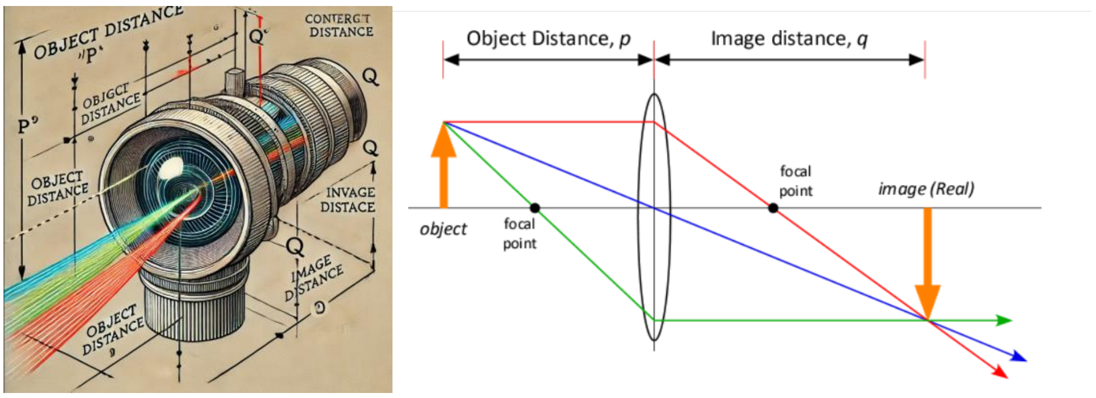


Figure 2.4: *Effect of focal length on binary matrix alignment and perspective --------- Ref (14)*

In the diagram above, the focused image is formed at the position marked by the inverted orange arrow, rather than at the focal point.

The focal length is defined as the distance between its focal point and the center of a convex lens. To measure the focal length of the lens experimentally, the position of the object is adjusted until a sharp and clear image is obtained. This clear image will appear at the position indicated by the yellow arrow.

In optical systems, focal length of a lens is a critical parameter that governs how light rays converge to form an image. We then use the lens formula:

$$\frac{1}{f} = \frac{1}{p} + \frac{1}{q}$$

where:

- f is the focal length,
- p is the object distance, and
- q is the image distance,

In this formula, p and q are distances. They help us to determine the lens's focal length. Now we can easily understand how light rays starting from a point converge after passing through a lens.
For this project, the lens analogy is applied conceptually in the perspective transformation process. The same way as how a lens adjusts the orientation and scaling of an image based on its focal length, the perspective transformation matrix implemented using OpenCV's `cv2.getPerspectiveTransform()`, warps the grid into a top-down view. It ensures that the laser dot's position and the grid's alignment can be mapped accurately to individual cells, even if the input image is captured from an angle.
The principles of ray optics are also understood when the light rays are bent through the optical axis of a converging lens, forming a real image at the focal plane. Similarly, in this project, rays of light from the laser pointer are traced and their positions mapped to the binary grid using mathematical transformations. This method makes the camera's ability to track light intensity and interpret the scene geometry in real time.
This connection between computer vision and optical theory tells us how classical concepts of optics helps in modern digital image processing techniques by making connection between theoretical and practical approaches.

Experimental Approach and Objectives
The experimental process integrates the following steps:

- Object and image distances are adjusted systematically.
- A laser pointer acts as the light source which produces a beam.
- A real-time grid mapping system is used to track the position of the focused laser dot.

The use of a laser ensures precision in alignment and focus detection.

The grid-based approach further refines the measurement by offering a high-resolution method for pinpointing the laser dot's position

B. Camera Calibration:

In this project, checkerboard patterns are used for calibration due to an easily detectable **grid corners**, which provide consistent and reliable geometric references.
The Camera calibration is helpful in optical tracking systems to correct lens distortions and achieve accurate 3D-to-2D point mappings

During calibration, the corners of the checkerboard pattern are used to compute the camera's intrinsic parameters like principal point, focal length, and distortion coefficients. The `cv2.calibrateCamera()` function in OpenCV is used to calculate the camera's intrinsic parameters and distortion coefficients. All these parameters are essential for compensating for lens distortions. This can cause straight lines in the scene to appear curved or distorted. Radial distortion and tangential distortion are worth to mention here mainly when we use wide-angle lenses.

These values are then applied to the camera feed during perspective transformations. Correcting for distortion ensures that the captured image aligns with the actual geometry of the scene which helps achieve an accurate position tracking.

Once the camera calibration is completed, the `cv2.undistort()` function is used to remove the distortions from the camera's feed. This correction ensures that the binary matrix, which serves as the reference grid, is accurately aligned with the real-world coordinates. It makes it possible to track the laser dot's position with high precision.

Figure 2.5 shows the checkerboard pattern with its detected corners. These corners are used as helpful points for calculating the camera's intrinsic parameters. The successful detection of these grid intersections helps to compute the key values like focal length and principal point. The parameters derived from this process are used to undistort the image and correctly transform the grid for position tracking.

By calibrating the camera with such a pattern, the system can more accurately map the binary matrix to its corresponding physical location, correcting any distortions that could otherwise misalign the matrix or interfere with laser point detection.

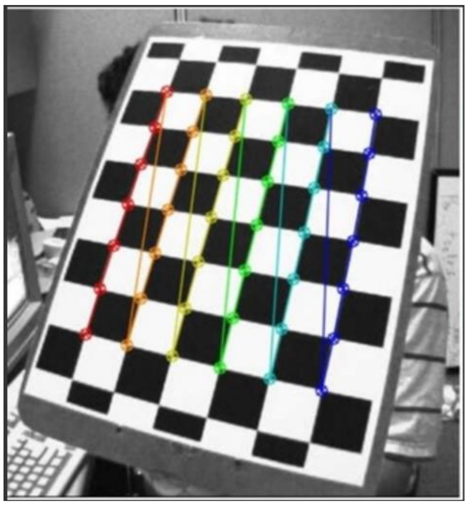


***Figure 2.5**: Checkerboard pattern with detected corners used for camera calibration --------- Ref (15)*

Figure 2.5 shows the corners detected on the checkerboard pattern, which are essential for calculating the camera's intrinsic parameters and distortion coefficients. These parameters are used for perspective corrections in subsequent transformations.

C. **Lighting Conditions:**

Variation in light intensity also affects the clarity of the captured image. Adjusting exposure and ensuring consistent lighting are essential for improving accuracy and reducing noise. `cv2.convertScaleAbs()` and `cv2.equalizeHist()` functions are helpful to enhance contrast and normalize lighting. We also apply thresholding methods such as `cv2.adaptiveThreshold()` to handle uneven lighting.

2.6 EXISTING OPTICAL TRACKING APPROACHES

The approach presented in this thesis builds upon marker-based and laser-guided systems, leveraging a binary matrix as the reference grid and a laser pointer for user input.

Several optical tracking approaches have been proposed in previous research:

1. **Laser-Guided Systems:**

   Laser dot detection is done using colour-based masking `cv2.inRange()`, followed by contour analysis `cv2.findContours()` in order to isolate the largest contour corresponding to the dot. The centroid of the dot is calculated using `cv2.moments()` for precise localization. We can apply the laser pointers in interactive systems. For example, gesture recognition interfaces and smart boards depends on accurate detection of the laser dot and its interaction with a reference grid.

2. **Marker-Based Systems:**

   The binary matrix in this project acts as a marker, and its detection is facilitated using contour-based techniques `cv2.findContours()`, grid alignment checks and thresholding. The project uses OpenCV functions like `cv2.matchTemplate()` to validate matrix positioning. It uses predefined visual markers such as binary patterns or ArUco markers which encode positional information. They are robust but require careful calibration and controlled environments.

2.7 OPENCV'S ROLE IN POSITION TRACKING

OpenCV is integral to implementing real-time position tracking, providing a robust tool. In this project, OpenCV is used for:

- **Laser Detection**: Tools like `cv2.inRange()`, `cv2.bitwise_or()`, and `cv2.findContours()` simplify the process of isolating and identifying the laser dot.
- **Contour Detection**: `cv2.findContours()` and `cv2.approxPolyDP()` are used to detect contours and refine the shape of the laser point.
- **Perspective Transformation**: OpenCV's `cv2.getPerspectiveTransform()` and `cv2.warpPerspective()` functions are used to correct distortions and align the detected laser point with the binary matrix.
- **Real-Time Visualization**: Functions like `cv2.polylines()` and `cv2.circle()` provide real-time visual feedback for debugging and accuracy assessment.
- **Image Preprocessing**: Functions like thresholding, color filtering, and morphological operations help isolate the laser pointer from the background.
- **Interactive Feedback**: OpenCV's real-time rendering helps for continuous visualization of the laser's position and grid alignment in live adjustments.

OpenCV's powerful geometric transformation and image processing capabilities enable reliable and adaptive features across various environments by allowing a robust alternative to traditional tracking methods. The combination of OpenCV with binary matrix tracking offers a cost-effective and scalable solution for real-time laser point detection and position mapping.

3 System Design and Methodology

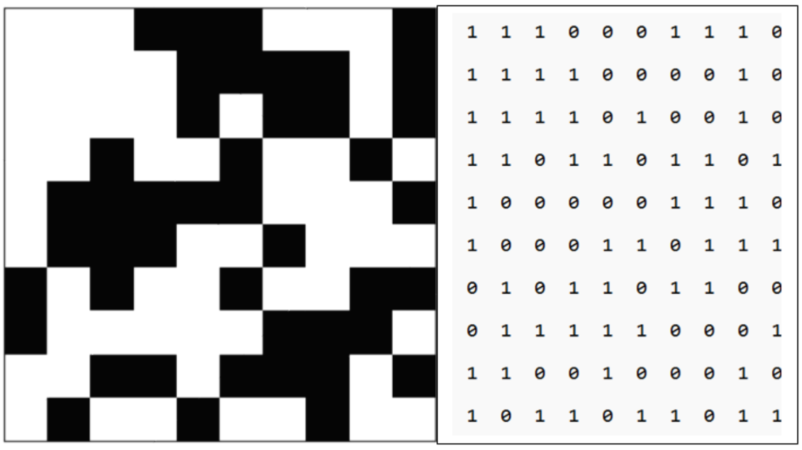
In this chapter, we discuss the use of a binary matrix combined with laser point detection, along with OpenCV to achieve real-time and accurate tracking. We addressed challenges like environmental variability, perspective distortion and explore the methodological techniques and design principles used to make the position tracking system.

3.1 Binary Matrix Design

The primary goal is designing the matrix with a grid that aligns with the camera's field of view. It ensures that the laser point could be mapped accurately to specific cells. All the cells in the matrix corresponds to a fixed uniform size, and it is designed in such a way that each 3x3 or 5x5 kernel is never the same. The laser point also has to cover one cell at a time.

The binary matrix is the main structure used to implement tracking the position and its design ensures simplicity and high contrast by allowing an easier way of detection and position identification. It is made up of cells that represent either `0` (`black`) or `1` (`white`). This allows us to clearly identify marked positions from those of the unmarked ones.

The binary matrix has distinguishable cells as shown in **Figure 3.1** with a potential position for the laser dot.



```
1 1 1 0 0 0 1 1 1 0
1 1 1 1 0 0 0 0 1 0
1 1 1 1 0 1 0 0 1 0
1 1 0 1 1 0 1 1 0 1
1 0 0 0 0 0 1 1 1 0
1 0 0 0 1 1 0 1 1 1
0 1 0 1 1 0 1 1 0 0
0 1 1 1 1 1 0 0 0 1
1 1 0 0 1 0 0 0 1 0
1 0 1 1 0 1 1 0 1 1
```

Figure 3.1: *10x10 binary matrix with (2.7mm x 2.7mm per cell) with cells marked as 1 (white) and 0 (black), providing a clear grid structure for laser point position tracking.*

3.1.1 Matrix Structure and Generation

The binary matrix's design allows us for an easy system to detect the laser point's location because each of the cells presented are identified and mapped according to coordinate.

The matrix ensures uniformity in clarity and cell size. We have made the following considerations:

- **Cell Size**: The cell in the matrix is designed to be small in order to match with the laser dot size. This approach ensures each laser is positioned to a specific matrix cell.
- **Grid Size**: We also scale the matrix to resolutions as we prefer based on the required precision.

As we have tried to show in **Figure 3.2**, a 10x10 grid offers a basic test setup for this project. On the other hand, **Figure 3.3** which is 589x589 grid allows us for higher resolution which will form the basis for our project.

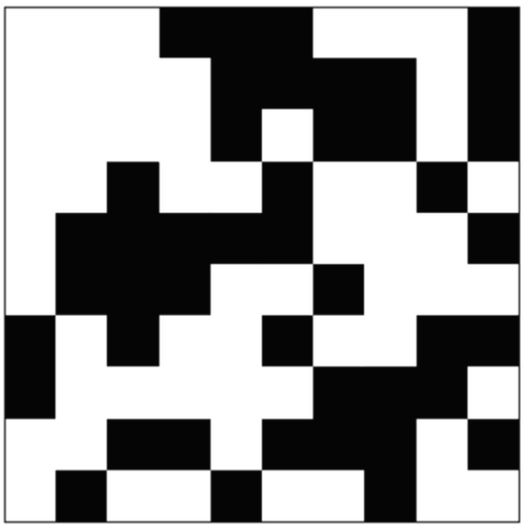


Figure 3.2: *Initial test matrix (10x10, 2.7 mm per cell) demonstrating the basic layout for laser position tracking*.

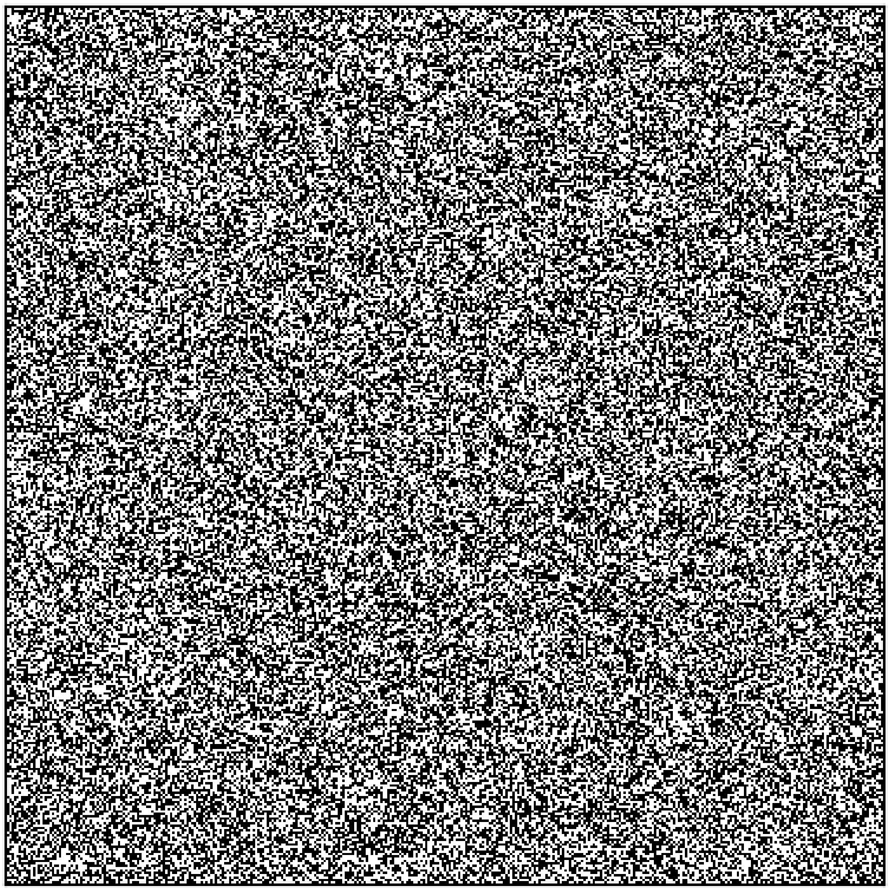


Figure 3.3*: Larger 589x589 binary matrix designed for higher precision and scalability in position tracking, illustrating the system's adaptability for different resolution requirements.*

The choices we make shows that the matrix provides structured representation of the space that is clear and could be detected easily by the camera while we apply it in real-time tracking.

3.1.2 Calibration and Testing for Precision

We have also performed precise calibration and an accurate tracking to both the binary matrix and the laser dot:

- **Calibration of Dot-to-Cell:** The size of the cell we chose has to fit precisely with the single cell of the matrix. The matrix size has been adjusted and we get an appropriate laser dot diameter of around 2.7 mm. It finally gives us an equal match to the laser point which corresponds to a specific position in the matrix.
- **Perspective Calibration:**
  - We get the detected four corners of the matrix by analyzing contours. Most of times we get the matrix tilted due to the camera angle.
  - We maintain the consistency of detected corners which also helps for perspective correction.
  - OpenCV's `cv2.getPerspectiveTransform ()` gives us a transformation matrix and `cv2.warpPerspective ()` by correcting the distortions aligns the matrix with the camera's view.

3.2 Laser Point Detection

We apply different image processing techniques in order to track and detect the laser point accurately. Detecting the laser point is the main component of this system. It allows the laser's position to be tracked and mapped to the binary matrix Image Preprocessing and Colour Filtering.

The laser dot is isolated by filtering the colour ranges specifically.

- **Filtering the Colour:** We covert the camera feed to HSV colour space using `cv2.cvtColor()`. It will do better separation of the background from the laser's red color.
- **Masking**: We use two color ranges for detecting the red hues of the laser (upper red and lower red). These outputs are then combined using `cv2.bitwise_or()` . This creates a comprehensive mask which can isolate the laser dot.
- **Reducing the Noise:**
  - We use morphological operations such as `cv2.MORPH_DILATE` and `cv2.MORPH_OPEN` in order to increase the visibility of the laser by removing the noise.
  - `cv2.GaussianBlur` which is a Gaussian blur technique have been applied in this project to make the image smooth enough and the laser dot's to be more visible.

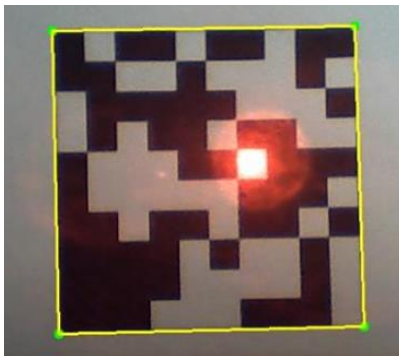


Figure 3.4: *Stabilized laser pointer's red dot after minimizing hand movement fluctuations.*

Figure 3.4 shows us the laser dot is captured clearly after we have applied masking and thresholding.

3.2.1 Detection Algorithm and Position Calculation

Thresholding and Colour filtering separate the laser point. After we do this the next step is getting its position:

- **Detecting the Contour**:
  - We get the contours extracted with the help of `cv2.findContours()`. We can now identify the laser dot.
  - The largest contour is based on the area it covers and it is then calculated by using `cv2.contourArea()`
- **Calculating the Centroid**: We get the centroid of the contour by using `cv2.moments()`. We also get the position of it with `cv2.circle()` to have real-time feedback.
- **Mapping the Coordinate**: The centroid we have got after calculation is then mapped with our binary matrix. It does the transformation matrix from `cv2.getPerspectiveTransform()`.

Even under variable environmental conditions like noise and variable light, the detection algorithm have shown it can track the laser point with high accuracy. **Figure 3.5** confirms the grid overlay in the laser's position within the correct cell and dot mapped onto the matrix.

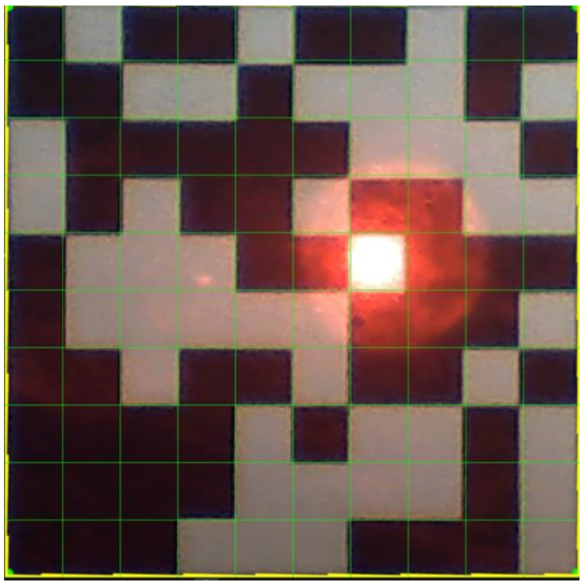


***Figure 3.5**: Laser dot captured on the binary matrix, with grid overlay for mapping.*

3.3 COORDINATE TRANSFORMATION

When we view the matrix at an angle, perspective distortion occurs most of the time. We need an accurate tracking of the position which needs to account for misalignments and distortions that occurs due to the angle of the camera. We then apply coordinate transformation techniques in order to detect the laser point and map it in a good way to correct issues that occur in the binary matrix.

3.3.1 Perspective Correction

The binary matrix most of the times appears distorted mainly because of the angle the matrix is viewed by the camera. Due to these issues, we apply perspective transformation in order to correct it.

- **Detecting the Corners:**
  - It identifies the Corners which are four in our case, it then preprocesses the frame with adaptive thresholding and Gaussian blur.
  - It then approximates the Contours to polygons. This helps us to ensure the accuracy and uses `0.02 * cv2.arcLength()` and `cv2.approxPolyDP()` from OpenCV library.

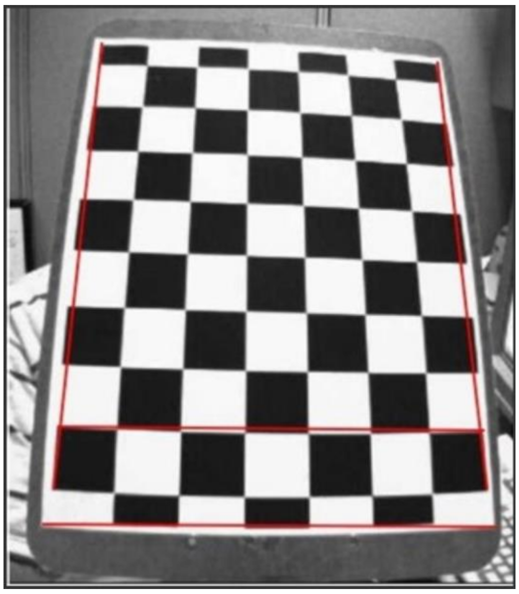


Figure 3.6: *Corner detection on Checkerboard pattern --------- Ref (15)*

- **Homography Transformation:** We compute the perspective transformation matrix using `cv2.getPerspectiveTransform()` tool in order to map the distorted grid to undistorted and flat, view.

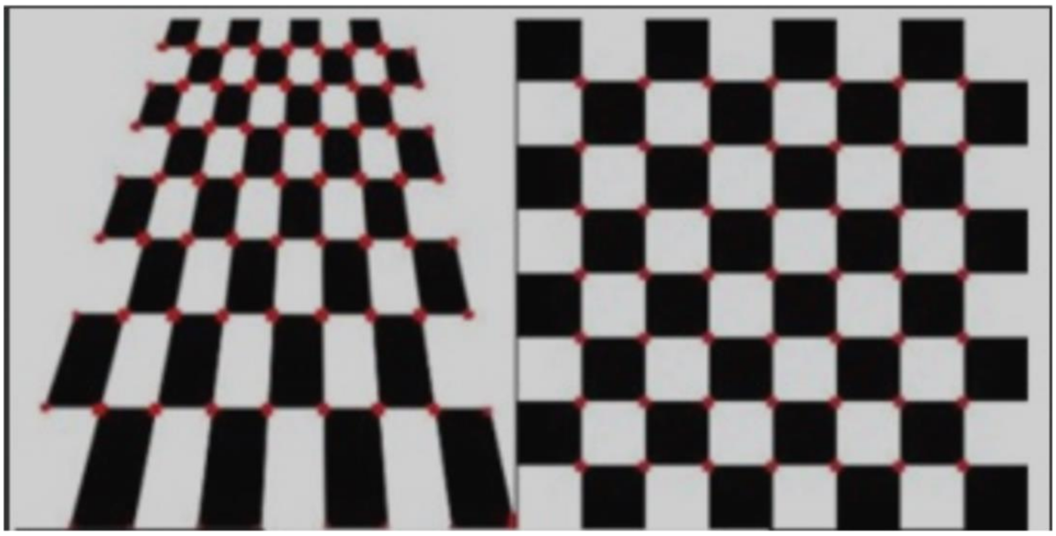


Figure 3.7: *Corrected perspective of a checkerboard --------- Ref (15)*

- **Warping**: We apply transformation matrix using `cv2.warpPerspective()` in order to adjust the matrix alignment to the correct one we need for the next stage.

Perspective distortions are introduced in the raw camera feed which often displays the matrix at an angle. Regardless of tilt and the camera's angle using this approach reduces the effects of perspective distortion by making the matrix to be correctly represented in the camera feed. The distortions can disrupt laser position tracking and misalign the grid.

The transformation from a distorted matrix to a flat are done by using homography from OpenCV. The corners are then mapped to a predefined rectangular region. `cv2.warpPerspective()` then makes the matrix a perfect grid. The accurate mapping of laser positions to grid cells is done by this transformation.

3.3.2 Grid Mapping and Alignment

In this stage the grid lines are overlaid on the warped image to have a proper alignment. We then use this grid to map the laser dot's position accurately to the appropriate cell in the matrix.

- **Overlay of the Grid**: This makes the position of the laser point's to be mapped to the correct matrix cell. It draws the grid lines to the transformed image. This corresponds to the corrected view of the matrix. At this stage we get the grid structure on the warped matrix.
- **Conversion of the Coordinate**: In the warped matrix we have the detected position of the laser. It is then transformed back to its original coordinates by the use of `cv2.perspectiveTransform()`.

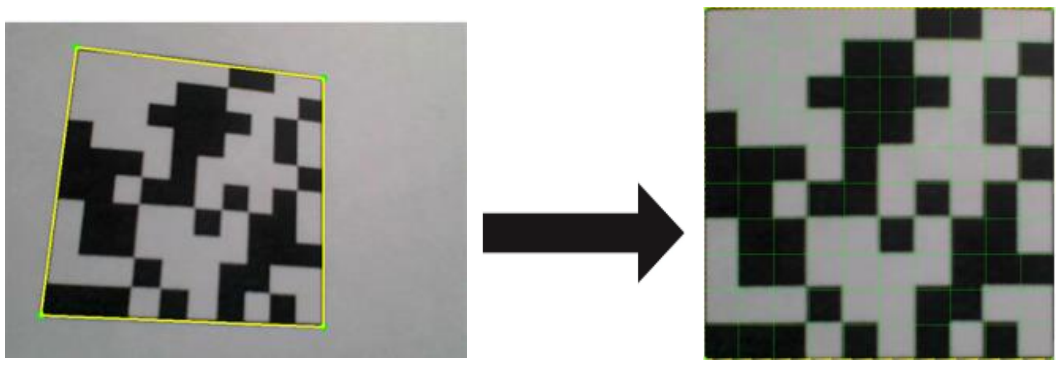


Figure 3.8: *Distorted matrix (left) and corrected perspective with grid overlay (right) for laser mapping.*

Even if the camera's viewpoint was initially misaligned or skewed, this approach makes the laser point to be positioned accurately within the correct matrix cell.

3.4 ERROR DETECTION AND CORRECTION

It is designed to reduce the impacts of environmental changes, noise and other inaccuracies during the tracking process. We implement several correction and error detection mechanisms. This helps us to ensure the reliability and accuracy of the position tracking system.

3.4.1 Noise Reduction Techniques

Noise in the webcam feed such as glare, reflections, or small irrelevant objects can significantly interfere with laser point detection. To address this, various noise reduction techniques are used:

- **Blob Filtering**: It removes any irrelevant objects by measuring the contour sizes with `cv2.contourArea().`
- **Dynamic Thresholding**: It adjusts ambient lighting sensitivity by the use of adaptive thresholding `cv2.adaptiveThreshold`.
- **Gaussian Blurring**: We apply the Gaussian blur `cv2.GaussianBlur()` in order to reduce sharp noise, ensuring that only significant features are detected like the laser dot.
- **Morphological Operations**: In order to get clear image by making it easier to be detected, we apply different morphological techniques such as Closing (filling small holes) or Opening (removing small objects). It makes the system to detect the laser point even in noisy environments while maintaining tracking accuracy.

3.4.2 Adaptation for Environmental Changes

The system incorporates adaptations to have robustness under different variable conditions. Camera positioning, changes in lighting, or other kinds of slight shifts in the printed matrix has an ability to affect the performance of the system.

- **Stabilizing the Laser**: Minor hand movements' influence needs to be reduced. In our system we have introduced a stabilization period `time ()` before processing the position of the laser. This makes to have a more accurate position detection by avoiding any false movements. By doing this, the laser dot remains stable despite fluctuations caused by hand tremors until it settles. When the laser dot stays within the same grid cell for a specified duration (0.5 seconds in our case), the system deems it stable and continues tracking.

- **Recalibration in Real-Time**: This technique makes the system to continue its functionality accurately even under varying environments. The system recalibrates to adapt to environmental changes while in operation in a periodic way. If the matrix or the camera is shifted then it will adjust the parameters of detection, like matrix alignment or the laser dot size to account for this kind of sudden shifts.

  If the matrix is displaced or the camera experiences movement, the system dynamically recalibrates essential parameters such as grid positioning or perspective alignment. This makes the system to operate in diverse settings. It will monitor and adjust to changes to deliver stable and reliable performance, especially during extended operation or under unpredictable conditions. It ensures that the detection and processing remain accurate, even in challenging or high-stress scenarios, solidifying its practical usability in real-world applications.

  Additionally Real-time recalibration guarantees that the system keeps its accuracy and functionality even when exposed to dynamically changing environments. When it has these kinds of situation the system performs periodic adjustments to account for variables like camera position, shifts in lighting or matrix alignment. This process allows the system to respond effectively to sudden disruptions or environmental fluctuations without requiring manual intervention.

  By combining these correction techniques and error detection, the system becomes unchanged in environmental variability. This ensures performance of reliable real-time tracking where the system waits for the laser to remain within a specific cell for 0.5 seconds before confirming the detection.

Camera Setup and Calibration

Proper setup and calibration of the camera Titanium Onyx USB Camera (Model: TC 101) is essential for optimal performance, ensuring accurate image capture. The process involves the following steps:

- **Connecting the Camera:** The camera is connected to the computer via the USB port, and necessary drivers and software are installed to enable functionality.
- **Calibration Settings:** Key camera settings such as **exposure**, **focus**, **white balance**, and **resolution** are adjusted during the calibration process to match specific environmental or application requirements.
- **Optimized Performance:** This calibration ensures that the camera produces clear, high-quality images and performs reliably across varying lighting conditions, making it suitable for a wide range of tasks, from industrial inspections to video conferencing.

To ensure that each 2.7 mm × 2.7 mm cell in the 158 cm × 158 cm matrix is clearly visible in our camera's field of view (FOV), we have some calculation in order to know the exact distance to position the *Titanium Onyx USB Camera (Model: TC 101)* in order capture the specific region of the matrix needed to be worked on.

Camera Specifications:

Webcam Model: Titanium Onyx USB Camera (Model: TC 101)

Focal Length: 4.2 mm (typical webcam specification)

Sensor Size: Approximately 1/1.4 inches, which translates to a sensor width of about 3.6 mm and height of 2.7 mm

Resolution: VGA (640 x 480 pixels)

Calculating the Field of View (FOV):

The horizontal FOV can be approximated using the formula:

$$\mathrm{FOV} = 2 \times \arctan\left(\frac{\text{sensor width}}{2 \times \text{focal length}}\right)$$

Plugging in the values:

$$\mathrm{FOV} = 2 \times \arctan\left(\frac{3.6\ \mathrm{mm}}{2 \times 4.2\ \mathrm{mm}}\right) \approx 46.4°$$

Determining the Optimal Distance:

To capture each 2.7 mm cell distinctly, we need to ensure that the camera's FOV covers a reasonable number of cells. Let's assume we want to capture a 27 cm × 27 cm section of the matrix, which corresponds to 100 × 100 cells.

Using the tangent function:

$$\text{Distance} = \frac{\text{width of section}}{2 \times \tan\left(\frac{\mathrm{FOV}}{2}\right)}$$

$$\text{Distance} = \frac{270\ \text{mm}}{2 \times \tan(\frac{46.4°}{2})} \approx 321\ \text{mm} \approx 32.1\ \text{cm}$$

Therefore, positioning your Titanium Onyx USB Camera (Model: TC 101) approximately 32.1 cm away from the matrix should allow you to capture a 27 cm × 27 cm section, with each 2.7 mm cell clearly visible.

Laser Dot Detection Optimization

- At **32.1 cm**, the laser dot should ideally occupy a significant portion of one cell, ensuring precise detection.
- **Steps to Optimize**:
  1. Confirm the dot's size in pixels relative to a cell. It should ideally cover 50% − 75% of the cell area.
  2. Adjust camera focus or distance slightly if the dot is too small or too large.

FOV (Field of View): Horizontal FOV = 46.4 °.

Field of View in Radians

To convert 46.4° to radians:

$$\text{FOV (in radians)} = \frac{46.4°}{180°} \times \pi \approx 0.810\ \text{radians.}$$

Horizontal Coverage (Matrix Width)

The horizontal width captured by the camera depends on the distance and the FOV:

$$\text{Visible Width} = 2 \times (\text{Distance}) \times \tan(\frac{\text{FOV}}{2})$$

At 32.1 cm distance:

$$\text{Visible Width} = 2 \times 32.1\ \text{cm} \times \tan\left(\frac{0.810}{2}\right) \approx 27\ \text{cm}$$

Vertical Coverage (Matrix Height)

To find the vertical FOV, we use the aspect ratio of the camera feed. For 640 × 480:

$$\text{Vertical FOV} = \text{Horizontal FOV} \times \frac{\text{Height}}{\text{Width}} = 46.4\ ° \times \frac{480}{640} \approx 34.8°$$

Convert to radians:

$$\text{Vertical FOV (in radians)} = \frac{34.8°}{180°} \times \pi \approx 0.607\ \text{radians.}$$

At 32.1 cm distance:

$$\text{Visible Height} = 2 \times 32.1 \text{ cm} \times \tan\left(\frac{0.607}{2}\right) \approx 20.2 \text{ cm}$$

Number of Cells in the Visible Area

Each cell in the matrix is 2.7 mm = 0.27 cm wide.

a. Horizontal Cells:

$$\text{Number of Horizontal Cells} = \frac{\text{Visible Width}}{\text{Cell Size}} = \frac{27}{0.27} \approx 100 \text{ cells.}$$

b. Vertical Cells:

$$\text{Number of Vertical Cells} = \frac{\text{Visible Height}}{\text{Cell Size}} = \frac{20.2}{0.27} \approx 75 \text{ cells.}$$

Pixel Coverage per Cell

The horizontal resolution is 640 pixels, and we see 100 cells horizontally:

$$\text{Pixels per Cell (Horizontal)} = \frac{\text{Horizontal Pixels}}{\text{Horizontal Cells}} = \frac{640}{100} \approx 6.4 \text{ pixels per cell.}$$

The vertical resolution is 480 pixels, and we see 75 cells vertically:

$$\text{Pixels per Cell (Vertical)} = \frac{\text{Vertical Pixels}}{\text{Vertical Cells}} = \frac{480}{75} \approx 6.4 \text{ pixels per cell.}$$

Scaling for webcam

If we want to adjust the visible area (e.g., capture a 27 cm × 27cm section of the matrix):

$$\text{Distance} = \frac{\text{Desired Width}}{2 \times \tan(\frac{\text{FOV}}{2})}$$

For a 27cm width:

$$\text{Distance} = \frac{270 \text{ mm}}{2 \times \tan(\frac{0.810}{2})} \approx 321 \text{ mm} \approx 32.1 \text{ cm}$$

Summary of Results

1. **Visible Area at 32.1 cm**: 27 cm × 20.2 cm.
2. **Number of Cells**: 100 horizontal × 75 verticals.
3. **Pixels per Cell**: 6.4 horizontal × 6.4 vertical.
4. **Distance of webcam for 27 cm × 27 cm Area**: ≈ 32.1 cm.

4 IMPLEMENTATION

This implementation chapter talks about software configuration, the hardware setup, the basic algorithms used to detect the laser dot and map it to a binary matrix. It details the use of webcam to detect and track the position of a laser pointer on a binary matrix. It also addresses the challenges faced during the implementation stage and presents an appropriate and reasonable solutions for each issue.

4.1 HARDWARE AND SOFTWARE SETUP

Hardware setup: Helps to facilitate detection of a laser pointer on a binary matrix. The components used are:

- **Webcam**: A webcam with a maximum video resolution of 1080 pixels (Full HD 1920x1080) has been used to capture the laser pointer and the live feed of the binary matrix. We position the camera at an optimal angle relative to the binary matrix to minimize distortion. This setup allows for capturing the specific section of the matrix from a calculated distance while reducing distortion in the images.
- **Laser Pointer**: We have used a red laser pointer in order to project a small dot onto the binary matrix. A specific red colour emitting laser pointer was chosen because of its ability to be easily detected in the HSV colour space.

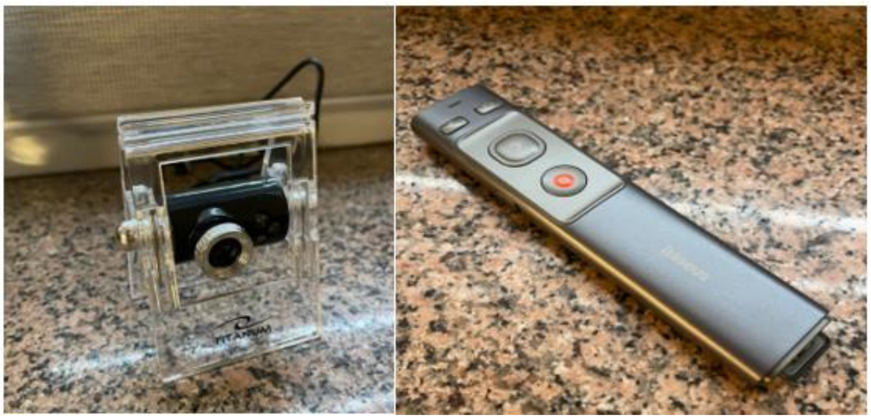


Figure 4.1: *Hardware tools used: Laser Pointer and Webcam*

Software Configuration: This project leverages several libraries for image analysis, system integration, and visualization:

- **OpenCV:** A core library for computer vision and image processing tasks. OpenCV performs contour detection, colour filtering, and perspective transformations. In this project, we have used *OpenCV version 4.10.0.*
  - *Computer vision* tasks involve interpreting and mapping visual elements from input image data. OpenCV detects the laser pointer's position and aligns it with the binary matrix grid.
  - *Image processing* makes raw image data to be manipulated and enhanced in order to get other features like smoothing or filtering which are crucial for laser dot detection.
- **Python**: This is the main programming language used to implement the algorithm. It manages the system flow and integrates all components. In our project, we have used *Python version 3.11.9.*
- **NumPy**: It is used for matrix operations and it handles numerical data related to the grid and laser detection. We used *NumPy version 1.26.2*.
- **Matplotlib:** Used for visualizing and analyzing results like plotting the detected grid and laser position. We used Matplotlib version 3.8.2 to create visualizations and graphs for the project.

4.2 ALGORITHM IMPLEMENTATION

Corner Detection

This step is used for isolating the matrix from the rest of the image. Our system analyzes the video feed from the webcam in order to locate the binary matrix within the frame. It identifies the corners of the matrix using the `detect_corners()` function by detecting the contours and approximates the largest one to a quadrilateral.

Corner Ordering

This stage guarantees that the matrix's orientation is consistent. Since we have a detected corners, then the `order_points()` function arranges them in a specific order which are top-left, top-right, bottom-right, and bottom-left.

Perspective Correction

This stage avoids distortions caused by the camera's angle. It will give us a straight, rectangular image of the binary matrix as shown in **Figure 4.4** below. It uses the ordered corners from the previous stage and then `get_perspective_transform()` function to apply a perspective transformation and align the matrix to a top-down view.

Grid Initialization

It helps us to map the laser pointer's position. The function `capture_baseline_grid()` processes the aligned matrix to extract the binary matrix's structure. Each cell is identified as either black `(0)` or white `(1)`. It then calculates the size of each cell.

Grid Visualization

This visual aid guarantees that the matrix has been divided correctly into cells corresponding to the number or rows and columns detected. It uses the function `draw_grid()`for debugging and verification purpose.

Laser Pointer Detection

Using the `detect_laser_pointer()` function, our system isolates and detects the laser pointer in the camera feed. This involves filtering the red color in the HSV space, then does removal of noise and calculates the centroid of the laser dot.

Stability Check

The system monitors the laser pointer's stability within a cell by using a time-based stabilization mechanism *(makes sure that the system only processes the laser pointer's position once it is confirmed to be steady and not rapidly moving*) in order to keep the accuracy in track.

Laser Position Mapping

The `detect_grid_and_box_size()` function maps the laser's position to the corresponding grid cell and extracts a region of interest (ROI) around the detected cell. Using the transformation matrix obtained earlier the detected laser pointer's position is transformed into the perspective-aligned space of the matrix.

Visualization and Result Output

This stage guarantees the system is accurately mapping the laser to the correct cell by doing real-time feedback for the user. The `debug_laser_pointer()` function finds the laser pointer's detected position on the grid. The system extracts the binary value of the corresponding grid cell when the laser remains stable in a cell for a predefined period. Finally, it will optionally display a 3x3 matrix centered on the laser's position

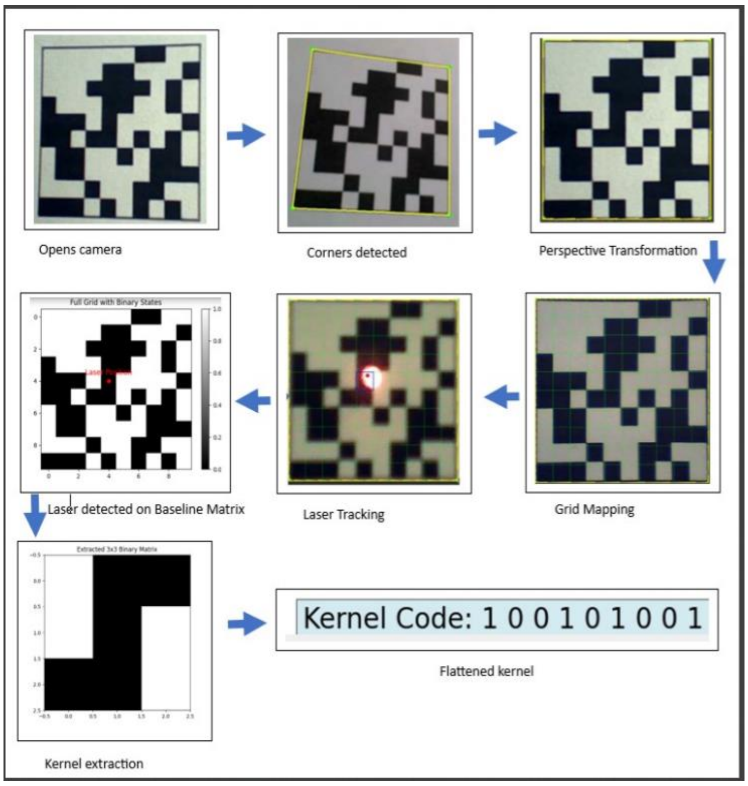

Opens camera
Corners detected
Perspective Transformation
Laser detected on Baseline Matrix
Laser Tracking
Grid Mapping
Kernel Code: 1 0 0 1 0 1 0 0 1
Flattened kernel
Kernel extraction

***Figure 4.2**: Collection of all the stages followed till we get the extracted binary matrix*

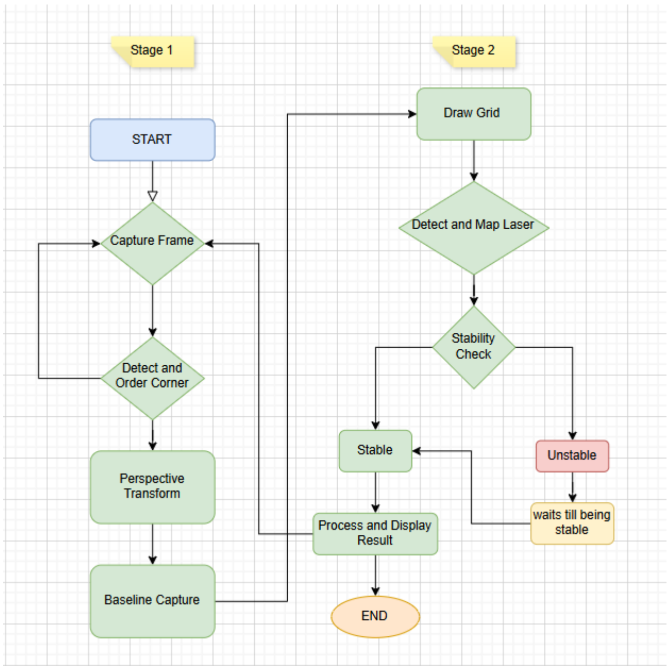


***Figure 4.3:** Flow Chat of the Algorithm Implementation*

The binary matrix is detected and aligned within the video feed. It becomes properly oriented for further processing and the laser pointer also becomes detected easily from the matrix that is aligned properly. It isolates its unique intensity and position within the frame. This information becomes mapped onto the grid and distinguishes the cell in the binary matrix. It does implement a stabilization mechanism for better accuracy which confirms the laser pointer's stability within a cell for a specific duration and decreases the errors caused by noise or sudden movements.

To maintain system reliability, we employ an error-handling measures as it is listed below.

a) **Detecting the Matrix and Aligning:**

- Initially it detects the binary matrix from the camera feed using contour detection techniques provided by OpenCV. The largest contours in the image are detected, which corresponded to the edges of the matrix.
- After it does the above then a perspective transformation was applied using the `cv2.getPerspectiveTransform()` and `cv2.warpPerspective()` functions in order to align the matrix and correct any distortions caused by the angle of the camera.

b) **Laser Detection**:

- It calculates the centroid of the largest detected contour to have the precise position of the laser in the image.
- The red dot from the laser pointer was isolated by filtering the image in the HSV color space. Two separate color ranges called the lower red values and the higher red values were used to identify the red hue.

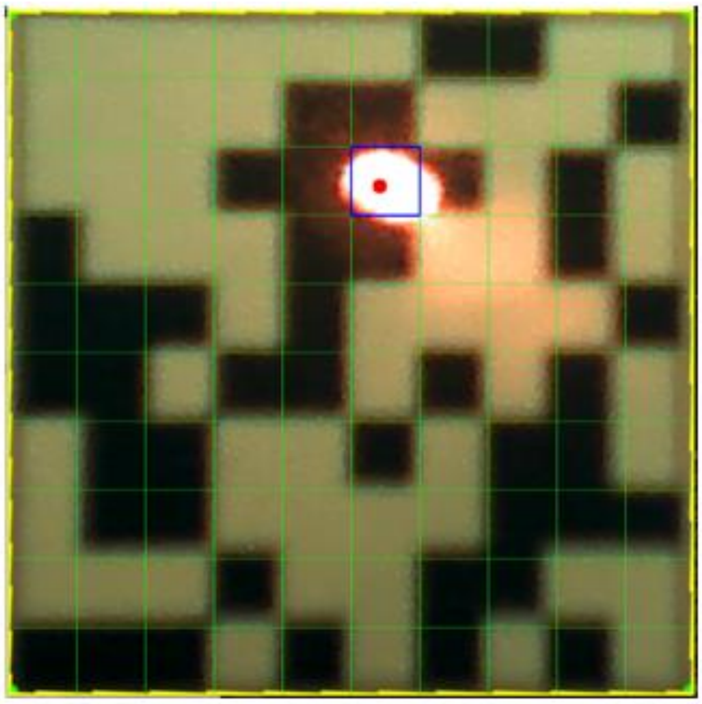


Figure 4.4: *A close-up of the camera feed showing the laser dot detected within a matrix cell of 10X10 Matrix each square with 2.7 mm by 2.7 mm dimensions*

- Morphological operations mainly `cv2.MORPH_OPEN` and `cv2.MORPH_DILATE` are used to reduce noise and enhance the visibility of the laser dot and a logical `OR` operation combines the resulting masks.

❖ **Morphological Transformations**: These are sets of operations that are used to process the structures and shapes within an image, typically applied to binary images. It takes two inputs which is the source image and a structuring element (kernel) that can determine the specific operation's effect.

Erosion and dilation are the primary morphological techniques used along with their derived forms like closing, opening and gradient. All of them have their own functionality and unique purpose. Their functionality can be better understood through illustrative examples.

1- Erosion:

- Erosion in image processing works the same as soil erosion. The process involves a kernel sliding over the image, much like in 2D convolution. It works by reducing the boundaries of the foreground object. The foreground is typically represented in white to clearly observe the effect of erosion. A pixel in the original image remains white with a value of 1 only if all the pixels under the kernel are also white; otherwise, the pixel is eroded or set to 0. This results in reduction of the boundary pixels, thereby making the thickness or size of the white regions or the foreground to decrease.
- Erosion is particularly useful to separate connected objects, avoid small white noise and refines the image's structure.

```
import cv2
import numpy as np
img = cv2.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 1)
```

2- Dilation:

- Dilation is the opposite of erosion. It sets a pixel in the image to '1' if at least one pixel under the kernel is '1'. It causes the white regions in the image to be expanded, effectively increasing the size of the foreground objects.

```
dilation = cv2.dilate(img,kernel,iterations = 1)
```

We use dilation mostly together with erosion for tasks such as noise removal. Erosion will eliminate small white noise that results in the shrinkage of the object's size. Dilation is then performed to restore the object's size without bringing back the noise. We also use dilation for connecting broken parts of an object by making it useful in enhancing image structures.

3- Opening: It is used to get rid of small noise from binary images while preserving the overall size or shape of the foreground objects. In this technique of morphological operation, it first does the erosion before dilation. Erosion will avoid small noises while dilation restores the original size of the main objects. We will use `cv2.morphologyEx()` function in OpenCV to implement this technique**.**

```
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```

4- **Closing:** It is the opposite of Opening. This morphological operation does dilation first, and then erosion. This technique will fill small gaps within the foreground objects, and after that, it will remove small noise on the object.

This operation will remove unwanted imperfections by first expanding the white regions and then refining the boundaries,

```
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

5- **Morphological Gradient:** This operation highlights the boundaries of objects in the image by focusing on the areas where changes in intensity occur. It calculates the difference between the dilation and erosion of the given image. It is useful for shape analysis and edge detection in binary images**.**

```
gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
```

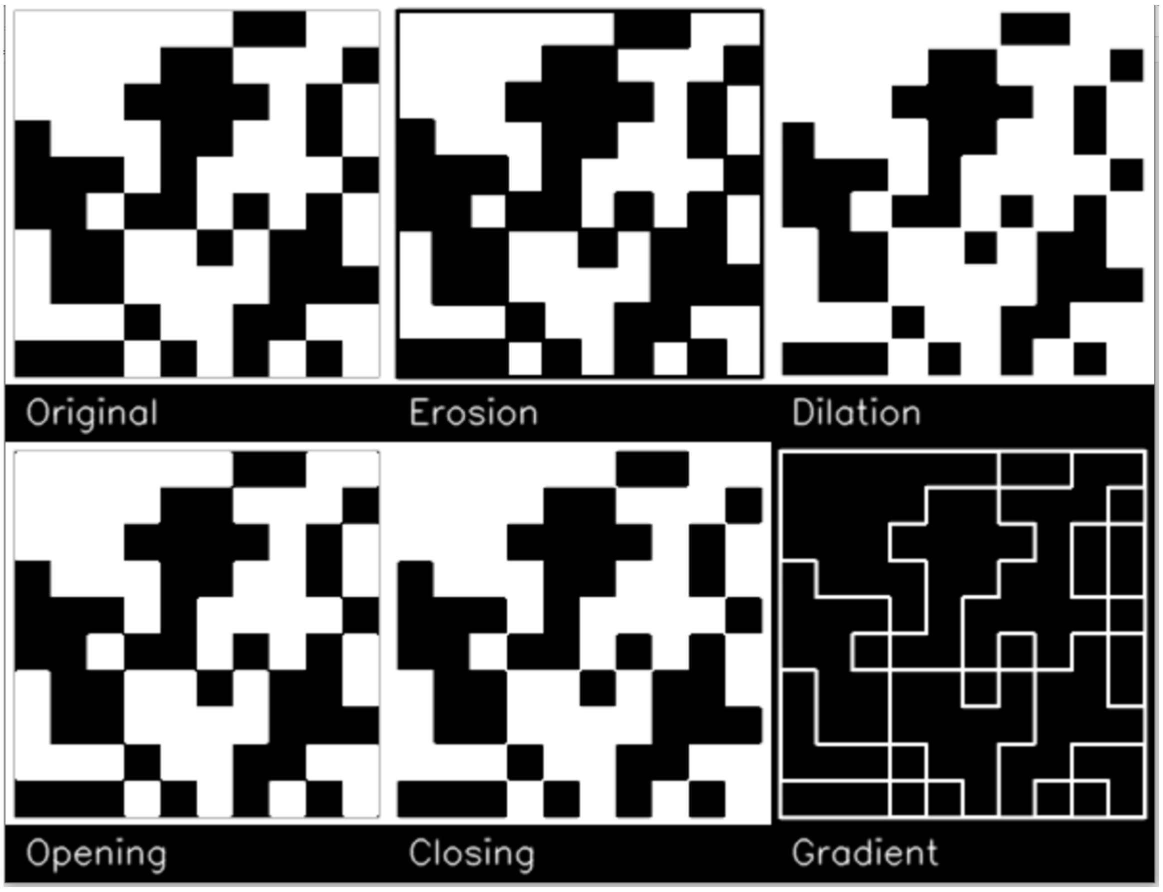


Figure 4.5: *Applying Morphological Transformations in the binary matrix used at our Project*

These techniques are helpful for improving the accuracy and clarity of laser pointer detection and grid corner identification. In this project, we have used `Opening`, `Closing`, and `Dilation` to refine the binary images. `Opening` removes small noises, `Closing` will fill the gaps and small holes, while `Dilation` increases the size of the detected objects to have a better visibility.

c) **Mapping of Grid**:

- Position became mapped onto the binary matrix after the laser dot was detected by dividing the image coordinates according to the cell size of the matrix. The laser dot's coordinates were divided into equal cells, and the corresponding cell it was pointing to was then calculated.
- Based on the cell with the larger portion of the red dot or the centroid's proximity to the cell's edges, the algorithm makes sure that the correct cell was identified.

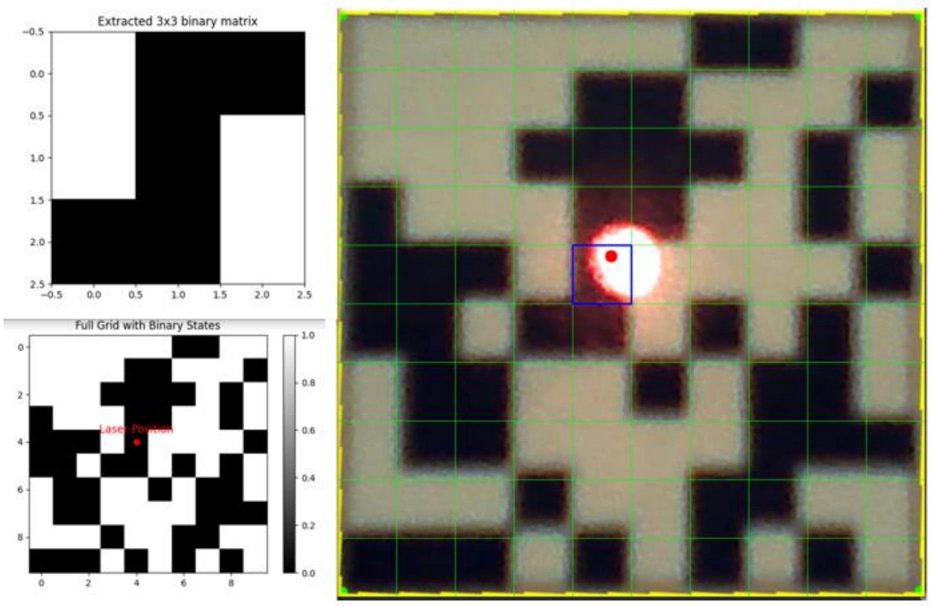
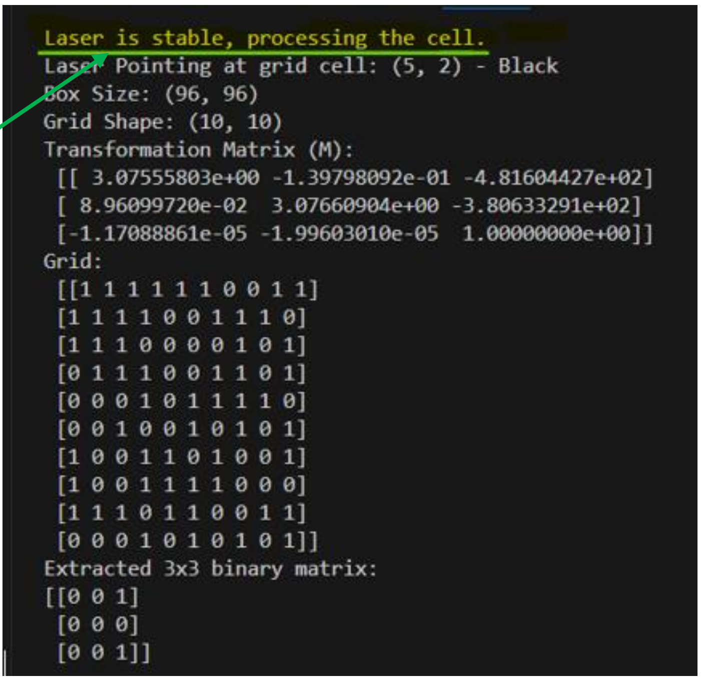


Figure 4.6: *Mapping the laser pointer to the cell with the largest portion of the laser dot when detected in boundary of 2-4 cells*

d) **Error Handling and Stabilization** :

- We did error handling mechanisms to manage issues such as the laser dot being outside the matrix area or varying lighting conditions
- The laser dot's position needs to be stable before processing it. For quick or unstable situations, the system will wait for a brief period to confirm that the laser was pointing to the same position for a specified duration (in our case we used 0.5 seconds).



```
Laser is stable, processing the cell.
Laser Pointing at grid cell: (5, 2) - Black
Box Size: (96, 96)
Grid Shape: (10, 10)
Transformation Matrix (M):
 [[ 3.07555803e+00 -1.39798092e-01 -4.81604427e+02]
 [ 8.96099720e-02  3.07660904e+00 -3.80633291e+02]
 [-1.17088861e-05 -1.99603010e-05  1.00000000e+00]]
Grid:
 [[1 1 1 1 1 1 0 0 1 1]
 [1 1 1 1 0 0 1 1 1 0]
 [1 1 1 0 0 0 0 1 0 1]
 [0 1 1 1 0 0 1 1 0 1]
 [0 0 0 1 0 1 1 1 1 0]
 [0 0 1 0 0 1 0 1 0 1]
 [1 0 0 1 1 0 1 0 0 1]
 [1 0 0 1 1 1 1 0 0 0]
 [1 1 1 0 1 1 0 0 1 1]
 [0 0 0 1 0 1 0 1 0 1]]
Extracted 3x3 binary matrix:
[[0 0 1]
 [0 0 0]
 [0 0 1]]
```

Figure 4.7: *showing stabilization of the system before displaying result*

The transformation matrix M in the above **Figure 4.7** is used for changing data points from one space to another. It is a tool for converting points from one coordinate to another.

This specific matrix is part of a laser processing system used to transform grid coordinates. The grid is a 10×10 shape, and the cell currently being processed is at the coordinates (5, 2). It determines the exact location and scale for the laser's actions on the grid.

Sample Code Overview: The main components of the code that we have used in this project were:

- **Detecting the Corner and Perspective Transformation**: The system has successfully detected the matrix corners using `detect_corners()` function. It has also done the perspective transform using `the get_perspective_transform()` function to get a top-down view of the matrix.
- **Detection of the Laser**: For the identified red laser dot, it calculated the centroid and mapped it to the binary matrix using `detect_laser_pointer()` function.
- **Detecting the Grid**: The system has detected the grid layout and the size of each box using `detect_grid_and_box_size()` function. On the other hand, `draw_grid()` function displayed the grid overlay on the processed image.

4.3 Coordinate Extraction And ROI

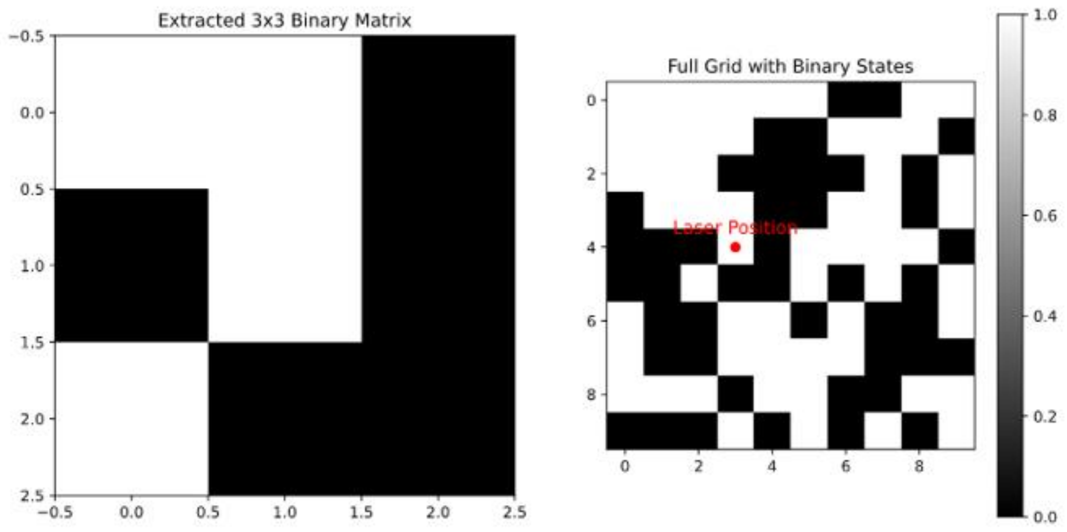
The system identifies the grid cell that the laser was pointing initially then a `3x3` or `5x5` binary matrix around that cell was extracted to have the coordinates of the laser dot using a Region of Interest (`ROI`) approach.

The system does the laser's centroid from the detected contour using `cv2.moments` to get an accurate tracking. The system can now easily identify the laser pointer's precise position in real time and will transform the laser's position using the perspective matrix `cv2.perspectiveTransform()`.

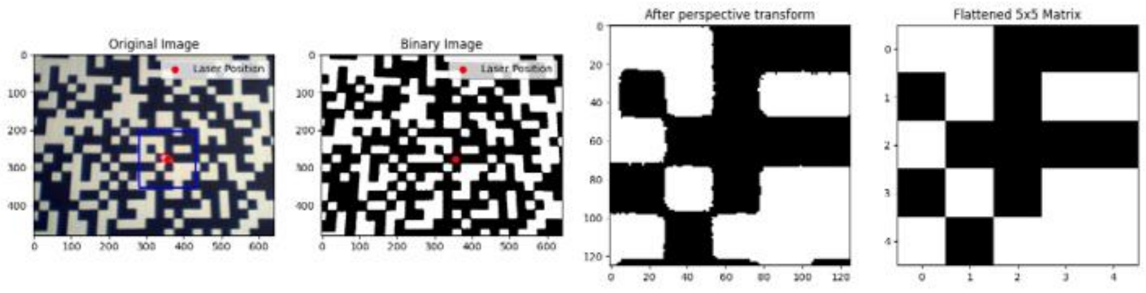
A Region of Interest (ROI) around the laser dot will be analyzed to generate a binary, which will be used for further calculations to ensure a high precision in identifying the grid cell's state.

Example:

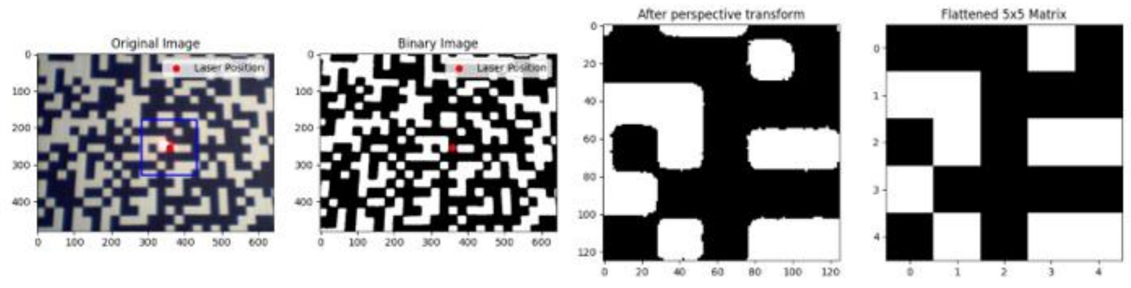
- The system extracted the submatrix (either 3x3 or 5x5 binary matrix) when the laser was directed to a specific cell in the matrix and displayed it for analysis. This step is useful to verify the accuracy of the laser's positioning.



Kernel Code: 1 1 0 0 1 0 1 0 0



Flattened Matrix: [1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1 1 1]



Flattened Matrix: [0 0 0 1 0 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1]

Figure 4.8: *A visual representation of the ROI extraction process, showing a 3x3 and a 5x5 binary matrix extracted from the grid around the laser dot, each with 2.7 mm by 2.7 mm dimensions.*

4.4 Code Highlights and Key Components

We have highlighted the main components of the code that were used in implementation of the laser detection and binary matrix mapping system. It is the backbone of the system's functionality such as grid mapping, error handling and perspective correction or laser dot detection.

a) Detecting of the Corner: It was helpful in this project for detecting the four corners of the binary matrix corners in the camera feed.

These corners are used to do a perspective transformation which aligns the matrix for accurate grid mapping.

```python
def detect_corners(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY_INV, 11, 2)
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
    morph = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel, iterations=2)
    contours, _ = cv2.findContours(morph, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        epsilon = 0.02 * cv2.arcLength(largest_contour, True)
        approx = cv2.approxPolyDP(largest_contour, epsilon, True)
        if len(approx) == 4:
            corners = approx.reshape(4, 2)
            return corners
    return None
```

- **Converting it to a Grayscale**:
  - Grayscale images are easier for processing and analyzing and we have used `gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)` to convert the given input image from color (Blue Green Red) to a grayscale.

    ✓ `cv2.cvtColor()`: This function is from OpenCV and converts an image from one color space to another.
    ✓ `img`: This is the input image in the BGR color format which is a standard in OpenCV.
    ✓ `cv2.COLOR_BGR2GRAY`: Grayscale images have intensity values from black (0) to white (255). This does the conversion from a Blue Green Red color image to a grayscale image.

- **Gaussian Blur**:
  - This step helps to avoid irrelevant contours which can be caused by noise. Gaussian blur of the grayscale image minimizes the noise and makes the image smooth using the function `blurred = cv2.GaussianBlur(gray, (5, 5), 0)`

    ✓ `cv2.GaussianBlur()`: This OpenCV function applies a Gaussian filter to the input image for minimizing the noises.
    ✓ `gray`: is the grayscale image obtained from the previous step.

- ✓ `(5, 5)`: This specifies the size of the kernel used for the Gaussian blur. If we use larger values based on the needs, it can result to more smoothing.
- ✓ `0`: It is the standard deviation of the Gaussian kernel in the x and y directions.

Adaptive Thresholding:

- Adaptive thresholding converts the grayscale image to a binary image using `thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 11, 2).`
- The `THRESH_BINARY_INV` flag will invert the threshold to make white areas become black or black area white.
- This stage isolates the matrix cells which are the black and white regions.
  - ✓ `cv2.adaptiveThreshold()`: Based upon local pixel intensity, it does adaptive thresholding to the image to segment it into black and white regions.
  - ✓ `blurred`: This is the smoothed grayscale image from the previous step.
  - ✓ `255`: It is the maximum intensity value for the binary output image.
  - ✓ `cv2.ADAPTIVE_THRESH_MEAN_C`: In our case the threshold for each pixel is the mean of the pixel values in its neighborhood. So this method specifies the thresholding method.
  - ✓ `cv2.THRESH_BINARY_INV`: It will invert the binary image to make the black to be white or the white to become black.
  - ✓ `11`: It is the size of the neighborhood which is 11x11 pixel block that is used to compute the threshold for each pixel.
  - ✓ `2`: is used to fine-tune the threshold.

Morphological Close:

- Morphological Close will fill small holes in the binary image using `morph = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel, iterations=2)` and makes the edges of the matrix more continuous and solid to easily detect contours.
  - ✓ `cv2.morphologyEx()`: It performs morphological operations on an image like opening or closing, erosion or dilation.
  - ✓ `thresh`: is the input image from the previous step.
  - ✓ `cv2.MORPH_CLOSE`: By first applying dilation then next erosion, this technique will fill small holes in the foreground regions and remove small black noise.
  - ✓ `kernel`: It determines the neighborhood used for the operation by defining the size and shape of the morphological operation.
  - ✓ `iterations=2`: It indicates closing operation is done twice to have a better effect.

Find Contours:

- It is applied to detect the contours in the binary image by using `contours, _ = cv2.findContours(morph, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE).`
- The external contours are retrieved by `RETR_EXTERNAL` and CHAIN_APPROX_SIMPLE does the simplification to the contour representation.
  - ✓ `cv2.findContours()`: is used for detecting the contours in a binary image.

Contours are curves that link continuous points that share the same color or intensity. It is helpful in shape analysis and object detection.

- ✓ `morph`: The input image will now provide a cleaner representation of objects for contour detection.
- ✓ `cv2.RETR_EXTERNAL`: It retrieves the outermost external contours to make the process simpler by ignoring nested contours inside other objects.
- ✓ `cv2.CHAIN_APPROX_SIMPLE`: It will optimize memory usage and processing time by only keeping essential points that define the shape. So, it reduces the number of points in the contour.
- ✓ `_`: is a placeholder for the second return value. It is the hierarchy information of contours.

- **Find the Largest Contour**:
  - `largest_contour = max (contours, key=cv2.contourArea)` finds the largest contour by area.
    - ✓ `contours`: This is a list of all the contours detected in the previous step, where each contour is a set of points that form the boundary of a shape in the binary image.
    - ✓ `max ()`: It will select the largest contour from the contours list.
    - ✓ `key=cv2.contourArea`: It is calculated using `cv2.contourArea()` to guarantee that the largest detected contour is selected.

- **Approximate the Contour**:
  - It calculates an approximation accuracy based on the contour's arc length using `epsilon = 0.02 * cv2.arcLength(largest_contour, True)`. If the epsilon value is smaller, it will give us a more accurate approximation of the contour.
    - ✓ `cv2.arcLength(largest_contour, True)`: It calculates the perimeter or arc length of the largest_contour. 'True' tells that the contour is closed.
    - ✓ `0.02`: is a scaling factor which is a fraction of the contour's perimeter. It determines the level of approximation detail.
  - `approx = cv2.approxPolyDP(largest_contour, epsilon, True)` make the largest contour to a polygon.
    - ✓ `cv2.approxPolyDP()`: It approximates a polygonal curve from a contour by reducing the number of points and also keeps the overall shape.
      - `largest_contour`: The input contour that needs to be simplified.
    - ✓ `epsilon`: is the maximum allowed distance between the approximated curve and the original contour.
    - ✓ `True`: shows that the contour is closed.

- **Check for Four Corners**:
  - The approximated polygon is checked for having exactly four corners using `if len(approx) == 4`. If this condition is met then the detected contour corresponds to the binary matrix.

- **Reshape and Return the Corners**:
  - `corners = approx.reshape(4, 2)` reshapes the corner points into a 4x2 array and each row represents the (x, y) coordinates of a corner . Finally, the points are returned as the result.

The `detect_corners()` function is used for ensuring that the system can map the laser pointer to the correct position on a properly aligned matrix. The process detects the four corners of the binary matrix by converting the image to grayscale, applying Gaussian blur and adaptive thresholding, and then using morphological operations to clean up the image. The largest contour is approximated to a polygon, and if it has four corners, these corners are returned for further processing.

b) **Ordering the Points:** It orders the detected corner points to have them in the correct sequence for the perspective transformation

```
    def order_points(pts):
     rect = np.zeros((4, 2), dtype="float32")

     # Sum the x and y coordinates to get the top-left and bottom-right points
     s = pts.sum(axis=1)
     rect[0] = pts[np.argmin(s)]  # Top-left point
     rect[2] = pts[np.argmax(s)]  # Bottom-right point

     # Calculate the difference between the x and y coordinates to get the other
two points
     diff = np.diff(pts, axis=1)
     rect[1] = pts[np.argmin(diff)]  # Top-right point
     rect[3] = pts[np.argmax(diff)]  # Bottom-left point

     return rect
```

- **Coordinates Sum**: It is used to identify the top-left and bottom-right corners. The top-left corner has the smallest sum, and the bottom-right has the largest sum. The coordinates sum for each point is calculated using (`pts.sum(axis=1)`).
- **Coordinates Difference**: The x and y coordinates difference is done using (`np.diff(pts, axis=1)`) to identify the top-right and bottom-left corners.
- **Point Ordering**: It gives us the `top-left, top-right, bottom-right`, and `bottom-left` in the correct order giving way for the perspective transformation to be applied well.

c) **Perspective Correction**: To have an accurate grid detection it does the perspective distortions. It takes care of the correct alignment of the matrix regardless of camera angle which may be tilted most of the times. This guarantees that the matrix appears correctly aligned in the image in spite of the camera's perspective.
This is done using the function below:

```
def get_perspective_transform(src_img, src_points, dst_size):
    # Order the source points (corners of the detected matrix)
    src_points = order_points(np.array(src_points, dtype='float32'))

    #  Define  the  destination  points  for  perspective  transformation
(straightened grid)
    dst_points = np.array([
        [0, 0],
        [dst_size[0] - 1, 0],
        [dst_size[0] - 1, dst_size[1] - 1],
        [0, dst_size[1] - 1]
    ], dtype="float32")

    # Get the transformation matrix from the source points to the destination
points
    M = cv2.getPerspectiveTransform(src_points, dst_points)

    # Apply the perspective warp to the source image
    dst_img = cv2.warpPerspective(src_img, M, dst_size)
    return dst_img, M
```

- **`order_points()`**:It uses the sum and difference of the coordinates to identify the top-left, top-right, bottom-right, and bottom-left points to order the four corner points of the detected binary matrix for the next step.
- **`cv2.getPerspectiveTransform()`**: It does the perspective transformation matrix to map the detected corners to a rectangular region.
- **`cv2.warpPerspective()`**: It applies the transformation matrix to the original image to have a top-down view of the binary matrix.

d) **Grid Mapping:** It divides the matrix into cells and maps the laser dot's position to the specific cell. This code extracts the binary matrix from the warped image and maps the laser's position onto the correct grid cell. The code below maps the corresponding cell in the binary matrix.

```
def detect_grid_and_box_size(warped_image, grid_shape):
    # Get the height and width of the warped image
    h, w = warped_image.shape[:2]

    # Calculate the size of each box in the grid
    box_size = (w // grid_shape[1], h // grid_shape[0])
    grid = np.zeros(grid_shape, dtype=int)

    # Convert the warped image to grayscale
    gray = cv2.cvtColor(warped_image, cv2.COLOR_BGR2GRAY)

    # Threshold the grayscale image to create a binary image
    _, binary = cv2.threshold(gray, 128, 255, cv2.THRESH_BINARY)

    box_h, box_w = box_size

    # Loop through the grid and check the mean intensity of each cell
    for i in range(grid_shape[0]):
        for j in range(grid_shape[1]):
            cell = binary[i * box_h:(i + 1) * box_h, j * box_w:(j + 1) *
box_w]
            grid[i, j] = 1 if np.mean(cell) > 127 else 0

    return grid, box_size
```

- **Calculating Grid Size**: The `box_size` is done by dividing the width (`w`) and height `(h)` of the warped image by the number of columns using `grid_shape[1]` and rows using `grid_shape[0]`.
- **Binary Thresholding**: `cv2.threshold()` is applied in the grayscale image to differentiate the black and white cells.
- **Mapping the Grid**: The mean intensity of each cell is calculated. If the mean intensity is greater than `127` indicates the white cell and the corresponding grid entry is set to unless it becomes `0`.
- **Grayscale Conversion**: Using `cv2.cvtColor()` the image is changed to grayscale to make the process of thresholding simpler.

e) **Drawing the Grid:** It is used to overlay a grid on the binary matrix. It divides the image to smaller cells based on the cell size. It then can be used for mapping the laser dot to a specific cell.

```
def draw_grid(warped_image, grid_shape, box_size):
    # Draw horizontal lines
    for i in range(1, grid_shape[0]):
        cv2.line(warped_image, (0, i * box_size[1]), (warped_image.shape[1],
i * box_size[1]), (0, 255, 0), 1)

    # Draw vertical lines
    for j in range(1, grid_shape[1]):
        cv2.line(warped_image, (j * box_size[0], 0), (j * box_size[0],
warped_image.shape[0]), (0, 255, 0), 1)
```

- **Inputs**:
  - **`warped_image`**: The one that does the perspective correction of the binary matrix.
  - **`grid_shape`**: is the number of rows and columns in the grid.
  - **`box_size:`** The dimensions which is the width and height of each cell in the grid. It is done using image size and grid shape.
- **Horizontal Lines Drawing**:
  - There is a loop that iterates over the number of rows using `for i in range (1, grid_shape [0]).`
  - `cv2.line()` puts a horizontal line on the given input.

    Both being at height `i * box_size[1]` has the start point (`0, i * box_size[1])` on the left edge, and the end point `(warped_image.shape[1], i * box_size[1])` on the right edge.
  - The colour `(0, 255, 0)` specifies green lines with a thickness of `1`.
- **Vertical Lines Drawing**:
  - The loop here also iterates over the number of columns in the grid using `for j in range (1, grid shape [1]).`
  - `cv2.line()` puts a vertical line down the image.

    Both being at width `j * box_size [0]` has the start point `(j * box size [0], 0)` on the top edge, and the end `point (j * box size [0], warped_image.shape[0])` on the bottom edge,
- **Final Result**:
  - Draws a grid on the binary matrix by dividing it into cells of size `box_size`. Each cell has a specific position in the binary matrix to allow the laser pointer to get its position.

f) **Laser Dot Detection:** It separates and identifies the laser dot, then maps its position. This ensures real-time detection, accurately tracking the laser dot's position on the matrix.

The following function below shows what it does:

```
def detect_laser_pointer(frame):
    # Define the HSV color range for detecting red laser
    lower_red1 = np.array([0, 70, 50])
    upper_red1 = np.array([10, 255, 255])
    lower_red2 = np.array([170, 70, 50])
    upper_red2 = np.array([180, 255, 255])

    # Convert the frame to the HSV color space
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Create masks for the two red color ranges
    mask1 = cv2.inRange(hsv, lower_red1, upper_red1)
    mask2 = cv2.inRange(hsv, lower_red2, upper_red2)

    # Combine the masks to detect the red laser
    mask = cv2.bitwise_or(mask1, mask2)

    # Apply morphological operations to clean up the mask
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((5, 5), np.uint8))
    mask   =   cv2.morphologyEx(mask,   cv2.MORPH_DILATE,   np.ones((5,   5),
np.uint8))

    # Find contours of the detected regions in the mask
    contours,     _     =     cv2.findContours(mask,     cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Get the largest contour, which corresponds to the laser dot
    largest_contour  =  max(contours,  key=cv2.contourArea)  if  contours  else
None

    if largest_contour is not None:
        M = cv2.moments(largest_contour)
        if M["m00"] != 0:  # Check for division by zero
            cX = int(M["m10"] / M["m00"])  # Compute the center x-coordinate
            cY = int(M["m01"] / M["m00"])  # Compute the center y-coordinate
            return (cX, cY)
    return None
```

- **HSV Color Space**: It separates the laser pointer's red color by using its hue values in the HSV color space.

  The `cv2.inRange()` function makes a binary mask after it has successfully detected the specified red ranges.

- **Morphological Operations**: It removes small noise using `cv2.MORPH_OPEN` and dilates the laser spots using `cv2.MORPH_DILATE`.

- **Contour Detection**: It identifies the contours of the detected regions using `cv2.findContours()` function.

- **Centroid Calculation**: Calculating the largest contour moments is achieved using `cv2.moments()` and then the centroid of the laser dot is computed. This will give us exact position of the laser.

g) **Laser Pointer Debugging:** It visualizes the detected laser pointer and its corresponding grid cell by highlighting the laser pointer on the grid and draws a bounding box around it.

```
def debug_laser_pointer(warped_image, laser_pos, box_size, grid_shape):
    # Draw a circle at the laser position for debugging
    cv2.circle(warped_image, laser_pos, 10, (0, 0, 255), -1)

    # Map the laser position to the corresponding grid cell
    grid_x = laser_pos[0] // box_size[0]
    grid_y = laser_pos[1] // box_size[1]

    # If the laser is within the grid, draw a rectangle around the cell
    if 0 <= grid_x < grid_shape[1] and 0 <= grid_y < grid_shape[0]:
        top_left = (grid_x * box_size[0], grid_y * box_size[1])
        bottom_right = ((grid_x + 1) * box_size[0], (grid_y + 1) * box_size[1])
        cv2.rectangle(warped_image, top_left, bottom_right, (255, 0, 0), 2)
```

- **Visualizing the Laser Pointer**: to visualize the laser point on the image by using `cv2.circle()`, a circle is drawn at the laser's position.
- **Mapping of the Grid**: By dividing the laser's coordinates by the size of the cell, the laser’s position is mapped to the corresponding grid cell.
- **Rectangle Drawing**: A rectangle is drawn around the detected cell to highlight the laser's exact location.

h) **Laser Stability Handling:** The stability mechanism ensures that the laser remains in a single grid cell for at least 0.5 seconds before the position is processed.

```
def handle_laser_stability(transformed_laser, grid_shape, box_size,
baseline_grid, M,
                           laser_stable_start_time, previous_laser_cell):
    grid_x = transformed_laser[0] // box_size[0]
    grid_y = transformed_laser[1] // box_size[1]

    if 0 <= grid_x < grid_shape[1] and 0 <= grid_y < grid_shape[0]:
        current_laser_cell = (grid_x, grid_y)
        if current_laser_cell == previous_laser_cell:
            if laser_stable_start_time is None:
                laser_stable_start_time = time.time()
            elif time.time() - laser_stable_start_time >= 0.5:
                print_laser_info(grid_x, grid_y, box_size, baseline_grid, M)
                laser_stable_start_time = None
        else:
            laser_stable_start_time = None
        previous_laser_cell = current_laser_cell

    return laser_stable_start_time, previous_laser_cell
```

- **Laser Stability Check**: The system checks if the laser pointer remains in the same grid cell between consecutive frames.
- **Stability Timer**: If the laser stays in the same cell for 0.5 seconds, it is considered stable and processed further.
- **Handling Laser Movement**: If the laser moves to a new cell before 0.5 seconds, the timer is reset to ensure stable detection before processing.

i) **ROI (Region of Interest) Extraction:** It extracts a region of interest from the binary matrix and gives us a 3x3 or 5x5 binary matrix. This function provides the grid layout to be used for extracting specific regions around the laser's detected position.

```
def capture_baseline_grid(warped_image, grid_shape):
    grid, box_size = detect_grid_and_box_size(warped_image, grid_shape)
    return grid, box_size
```

- **Extraction of the Grid**: It calls the `detect_grid_and_box_size()` function to have the binary grid and the size of grid cell.
- **Return Values**: It returns the grid and the size to be used for extracting specific regions of interest for further analysis.

j) **Laser Information Printing and Visualization:** The system prints out information about the laser position, grid cell, and extracted binary matrix for visualization and debugging.

```
def print_laser_info(grid_x, grid_y, box_size, baseline_grid, M):

        # Extract 3x3 binary matrix around the clicked point
    roi_binary = baseline_grid[max(0, grid_y - 1):min(baseline_grid.shape[0],
grid_y + 2),
                               max(0, grid_x - 1):min(baseline_grid.shape[1],
grid_x + 2)]

        # Flatten the matrix
    flattened = roi_binary.flatten()
    flattened_str = ' '.join(map(str, flattened))
    print("Flattened Binary Matrix:")
    print(flattened_str)

    color = 'White' if baseline_grid[grid_y, grid_x] == 1 else 'Black'

    print("Laser is stable, processing the cell.")
    print(f"Laser Pointing at grid cell: ({grid_x}, {grid_y}) - {color}")
    print("Box Size:", box_size)
    print("Grid Shape:", baseline_grid.shape)
    print("Transformation Matrix (M):\n", M)
    print("Grid:\n", baseline_grid)

    print("Extracted 3x3 binary matrix:")
    print(roi_binary)
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.imshow(roi_binary, cmap='gray')
    plt.title("Extracted 3x3 Binary Matrix")

    plt.subplot(1, 2, 2)
    plt.imshow(baseline_grid, cmap='gray')
    plt.title("Full Grid with Binary States")
    plt.colorbar()
    plt.scatter(grid_x, grid_y, color='red')
    plt.text(grid_x, grid_y - 0.5, 'Laser Position', color='red', fontsize=12,
ha='center', va='center')
    plt.figtext(0.5, 0.01, f"Kernel Code: {flattened_str}", ha='center',
fontsize=20, bbox={"facecolor": "lightblue", "alpha": 0.5, "pad": 5})
    plt.show()
```

- **Information Display**: Prints the grid coordinates of the laser pointer, the binary state of the cell, and the extracted 3x3 or 5x5 binary matrix.
- **Visualization**: It also visualizes the extracted binary matrix and the full grid with the laser position marked.

k) **Main Loop Overview:** The main loop handles webcam frame acquisition, matrix detection, laser detection, stabilization, and visualization.

Frame Acquisition

```
ret, frame = cap.read()
if not ret:
    break
```

The `cap.read()` function captures a frame from the webcam. If no frame is returned then the loop breaks.

Detection and Highlighting of Matrix Corners

```
corners = detect_corners(frame)
if corners is not None:
    for corner in corners:
        cv2.circle(frame, tuple(corner), 5, (0, 255, 0), -1)
    cv2.polylines(frame,  [corners.reshape((-1,  1,  2)).astype(np.int32)],
isClosed=True, color=(0, 255, 255), thickness=2)
```

The corners of the binary matrix are found by `detect corners ()`. If corners are detected then they are highlighted using `cv2.circle()` and `cv2.polylines()` draws the entire contour.

Perspective Transformation

```
ordered_corners = order_points(corners)

warped_image, M = get_perspective_transform(frame, ordered_corners, dst_size)
```

Using `order_points()`, the detected corners are ordered to prepare for perspective transformation. It uses `get_perspective_transform()` to align the matrix to a top-down view and does the transformation matrix.

Initializing the Baseline Grid

```
if baseline_grid is None:
    baseline_grid, box_size = capture_baseline_grid(warped_image, grid_shape)
```

The baseline grid and the size of each cell are initialized in the first iteration using `capture_baseline_grid()`.

Grid Drawing

```
draw_grid(warped_image, grid_shape, box_size)
```

Using `draw_grid()` function, a grid is drawn on the warped image which is used to visualize the structure of the binary matrix.

Laser Dot Detection

```
laser_pos = detect_laser_pointer(frame)
```

The laser pointer's position is detected using `detect_laser_pointer()`, which returns the coordinates `(x, y)` of the laser.

Mapping Laser Position to Matrix

```
transformed_laser      =      cv2.perspectiveTransform(np.array([[laser_pos]],
dtype="float32"), M)[0][0]
transformed_laser = tuple(map(int, transformed_laser))
```

The laser dot's position is transformed from the camera's perspective to the aligned matrix using the perspective transformation matrix `M`.

- **Laser Pointer Debugging**

```
debug_laser_pointer(warped_image, transformed_laser, box_size, grid_shape)
```

The laser pointer's position is visualized on the warped image with a circle using `debug_laser_pointer()`. It also highlights the corresponding grid cell to show the laser's detected position.

Stabilization Logic

```
laser_stable_start_time, previous_laser_cell = handle_laser_stability(
    transformed_laser, grid_shape, box_size, baseline_grid, M,
    laser_stable_start_time, previous_laser_cell)
```

The laser's position is checked for stability: If it remains within the same grid cell for 0.5 seconds, it is considered stable, and the system processes its position.

Display and Exit Condition

```
cv2.imshow('Warped Image', warped_image)
cv2.imshow('Frame', frame)
if cv2.waitKey(1) & 0xFF == ord('q'):
    break
```

It shows the image in process and the original frame. If the user presses the `q` key, it exists the loop. The processed warped image and the original frame are displayed using `cv2.imshow()`. The loop terminates when the "`q`" key is pressed.

5 RESULT AND ANALYSIS

The results obtained from implementing the system are presented in this fifth-chapter. It analyzes the performance in different conditions. Response time, detection accuracy and robustness to environmental factors are used to evaluate and measure the results

5.1 TEST OF PERFORMANCE UNDER CONTROLLED CONDITIONS

Performance testing under controlled conditions ensures the core functionality of the system. A stable environment with ***minimal background noise***, ***uniform lighting*** and ***a fixed binary matrix*** is chosen to do this stage.

While doing the tests:

- **Initializing the Grid and Perspective Correction:** It is used to guarantee precision of cell mapping. It properly identified and detected grid corners and then performed perspective transformations to have a top-down perspective.

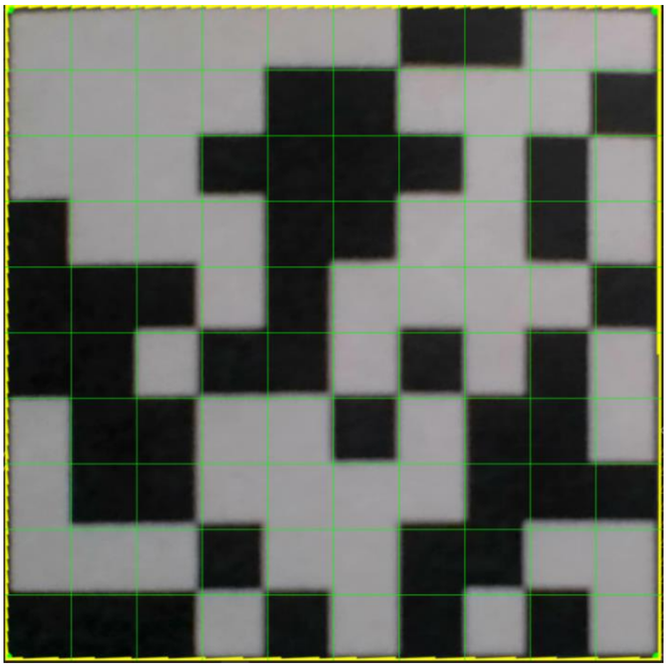


Figure 5.1: *Results obtained after detected grid corners and performed perspective transformations in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions*

The corner detection algorithm has properly identified the edges of the matrix by having over 95% accuracy.

- **Detecting the Laser Pointer**: Morphological transformations has effectively reduced the noise by separating the laser dot in the HSV colour space. Even when it moved rapidly the laser dot was consistently identified.

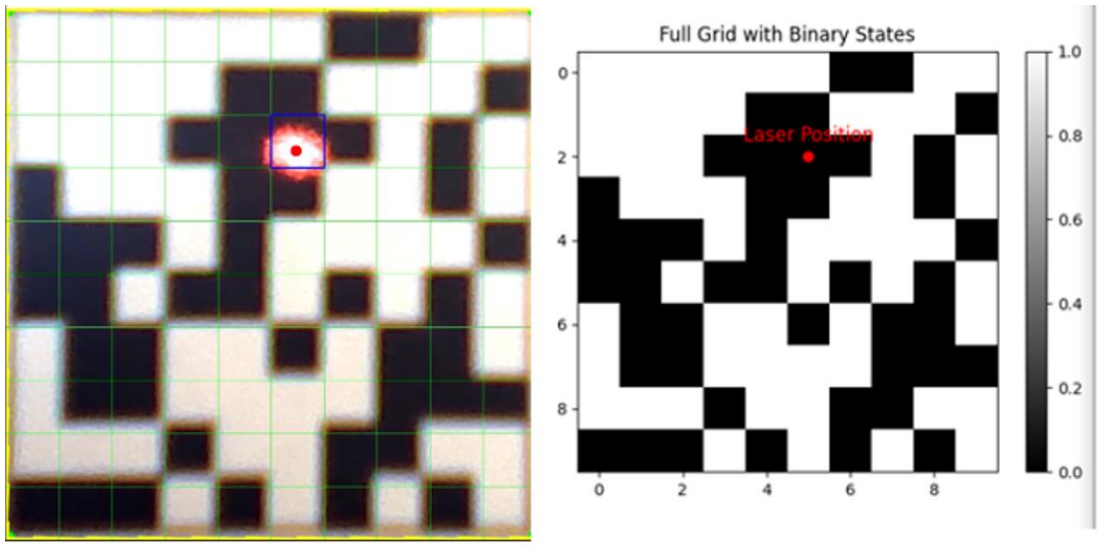
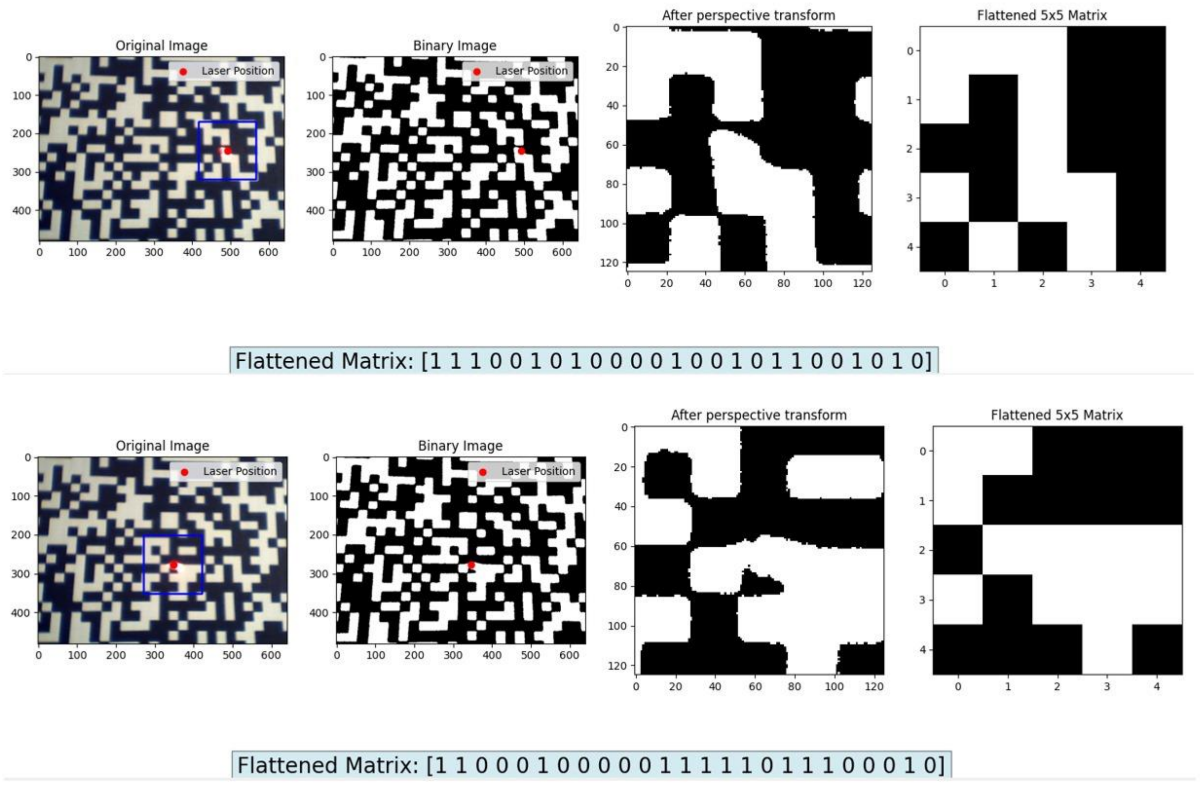


Figure 5.2*: Laser Pointer Detected inside a Grid Cell in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions*

- **Overcoming the challenge of Detecting Black as White with High Laser Intensity:**

The challenge of accurately distinguishing black regions from white regions on a binary matrix when the laser pointer intensity was high was addressed below. The laser's brightness occasionally caused black regions to appear white, introducing errors in the detection process.



Flattened Matrix: [1 1 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0 1 0]

Flattened Matrix: [1 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 1 1 0 0 0 1 0]

Figure 5.3: *A black cell appeared as white in high laser intensity leading to detection challenges*

To resolve this, we adopted a systematic approach involving preprocessing and capturing a stable snapshot.

Capturing a Stable Snapshot:

- To minimize the effects of laser fluctuations, we have first captured approximately 30 frames using the camera to stabilize the initial reference frame. It has shown that the binary matrix and the laser position were correctly identified without transient artifacts caused by laser intensity variations.

```
    for i in range(0, 30):
        _, orig_frame = cap.read()
```

The stabilized snapshot served as the base image for detecting the laser pointer and extracting the matrix, providing a consistent starting point for analysis.

In addition to the reason why the laser pointer does not alter the classification of the region's original binary state (black or white) lies in the **thresholding** and **mean-based grid classification** logic. When detecting the grid's binary state, the code calculates the **mean intensity** of each cell using:

```
grid[i, j] = 1 if np.mean(cell) > 127 else 0
```

Even if a bright red laser spot is introduced into the cell, its impact on the mean intensity depends on its size relative to the cell. The laser spot covers only a small fraction of the cell, and the remaining area's intensity (original black or white) dominates the mean. Thus:

- ✓ In **black regions**, the laser's bright spot is insufficient to raise the mean above the threshold (127), so the cell remains classified as black.
- ✓ In **white regions**, the laser's bright spot adds to an already high mean, keeping the classification as white.

5.2 Accuracy and Response Time Analysis

When the system does the identification of targeted cells, accuracy was tested by aiming the laser pointer at specific grid cells. We did more than 100 trials in a different laser movement speeds and grid sizes.

The results are:

- **Accuracy Analysis:** Detection accuracy of 96–98% is achieved. There were some misdetections at the boundaries of grid cells which has contributed to overlapping contours or slight distortions while doing perspective correction.
- **Testing the Response Time:** Frame processing average time of 20 ms or 50 FPS (Frame per Second) was achieved after testing on a computer with GPU specification of Intel(R) UHD Graphics 620 and processor specification of Intel(R) Core™ i5-8350U CPU @ 1.70GHz 1.90 GHz. The system's response time was analysed across different scenarios:
    - Small grids had an average response time of 150 milliseconds.
    - Larger grids even though it had increased computational load, the processing time still remained under 200 milliseconds.

The system maintained a frame rate of 45-50 FPS on a standard laptop. When the optimization of contour detection algorithms and morphological operations gets advanced and improved more, it can make the system to handle these scenarios effectively.

❖ **Laser Detection and Grid State Mapping with Precise Region Selection:**

- **Fully Inside the Grid:**

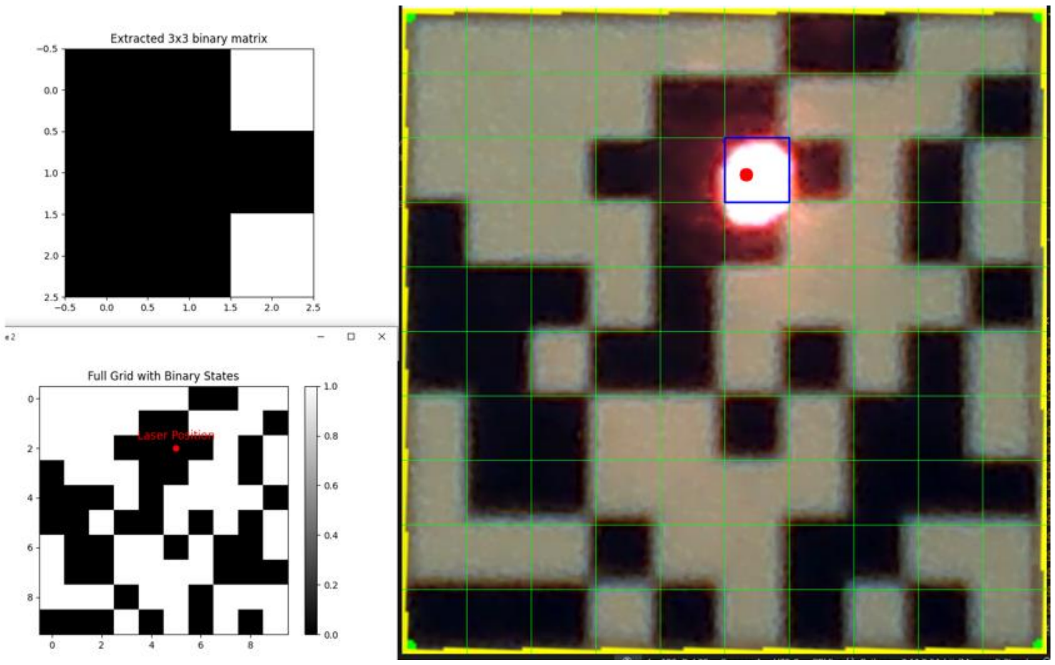


Figure 5.4: *Laser Inside A cell in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions*

- **At the boundary between two or more Grids:**

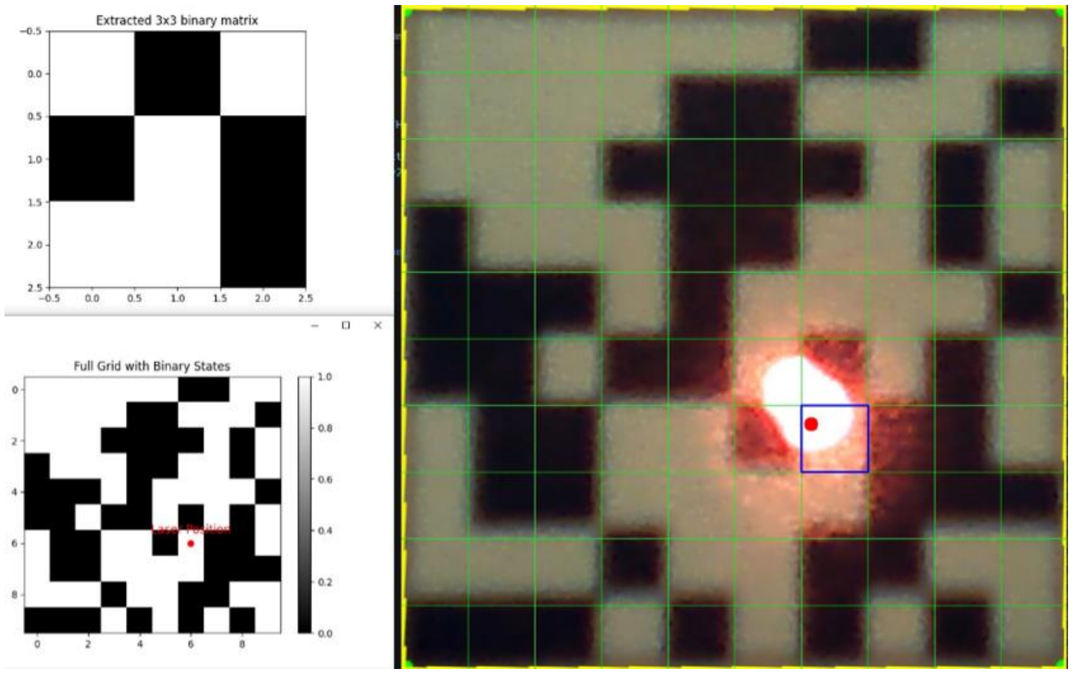


Figure 5.5: *Laser at the Boundary in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions*

- **Top Left or the Extracted 3x3 Binary Matrix:** It shows a localized binary representation of the grid around the laser. The binary states which are black and white cells relates with the grid cells directly around the detected laser.

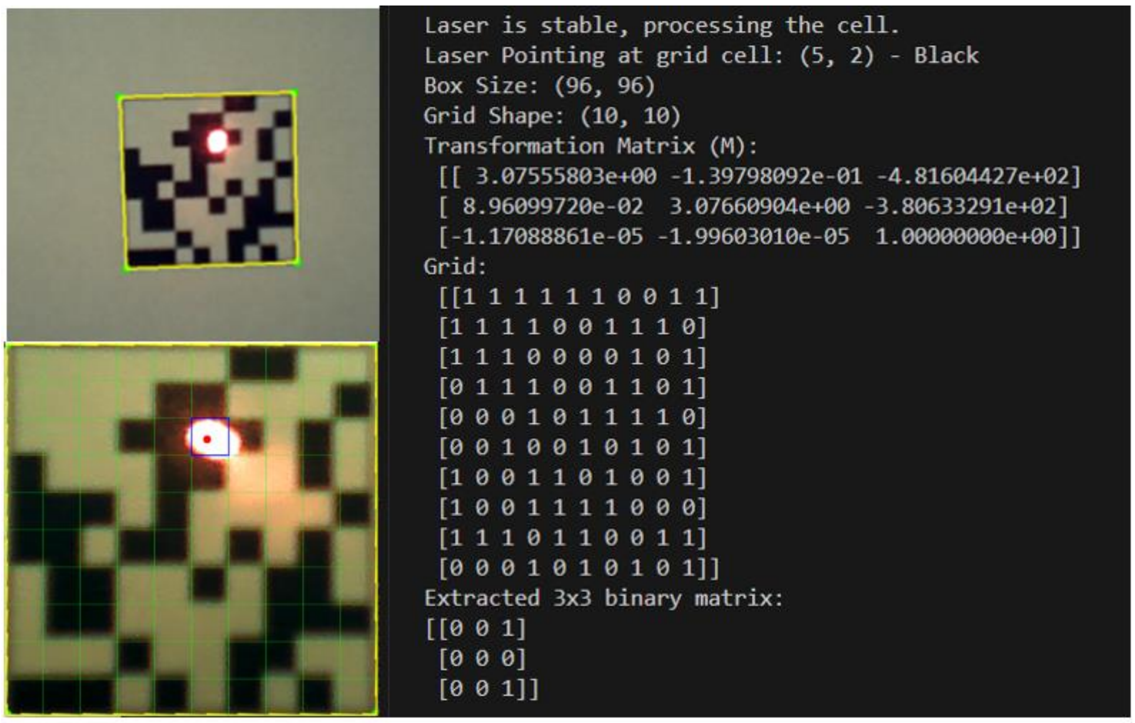
- **Bottom Left or Full Grid with Binary States:** it indicates the entire grid's binary representation with black and white cells. The red dot represents the laser pointer's mapped position in the grid.
- **Right or Warped Image with Detected Laser Dot:** is a perspective-corrected image where the green lines define the grid structure, while the red dot and blue bounding box highlight the detected laser's precise location.

Why That Specific Region Was Chosen:

- **Blue bounding box:** The blue bounding box highlights the grid cell identified as the detected laser position. Despite laser dot being detected at the boundary of four adjacent grid cells, the system selects the cell where the majority of the laser dot area falls. This approach ensures that the chosen cell accurately represents the laser's dominant position, minimizing ambiguity in detection.
- **Red dot (the laser centroid):** Aligns with the centre of the grid cell. It makes sure accurate mapping to the binary grid to be fulfilled and ensures precision in identifying the laser's position relative to the grid.

❖ **Robust Laser Detection and Grid Mapping Under Blurred Conditions:**

Figure 5.6 shows the laser pointer detection process under a blurred environment where grid edges and cell definitions are less sharp. The system has identified the laser dot and gave an accurate result.



```
Laser is stable, processing the cell.
Laser Pointing at grid cell: (5, 2) - Black
Box Size: (96, 96)
Grid Shape: (10, 10)
Transformation Matrix (M):
 [[ 3.07555803e+00 -1.39798092e-01 -4.81604427e+02]
 [ 8.96099720e-02  3.07660904e+00 -3.80633291e+02]
 [-1.17088861e-05 -1.99603010e-05  1.00000000e+00]]
Grid:
 [[1 1 1 1 1 1 0 0 1 1]
 [1 1 1 1 0 0 1 1 1 0]
 [1 1 1 0 0 0 0 1 0 1]
 [0 1 1 1 0 0 1 1 0 1]
 [0 0 0 1 0 1 1 1 1 0]
 [0 0 1 0 0 1 0 1 0 1]
 [1 0 0 1 1 0 1 0 0 1]
 [1 0 0 1 1 1 1 0 0 0]
 [1 1 1 0 1 1 0 0 1 1]
 [0 0 0 1 0 1 0 1 0 1]]
Extracted 3x3 binary matrix:
[[0 0 1]
 [0 0 0]
 [0 0 1]]
```

Figure 5.6: *Result obtained in a blurred image detection in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions*

Components of the Image:

- **Warped Grid at the Top:** It first applies perspective correction then the grid will be displayed. The detected position of the laser pointer is highlighted by the red dot.
- **Information of Laser in the Terminal:**

The terminal output shows:

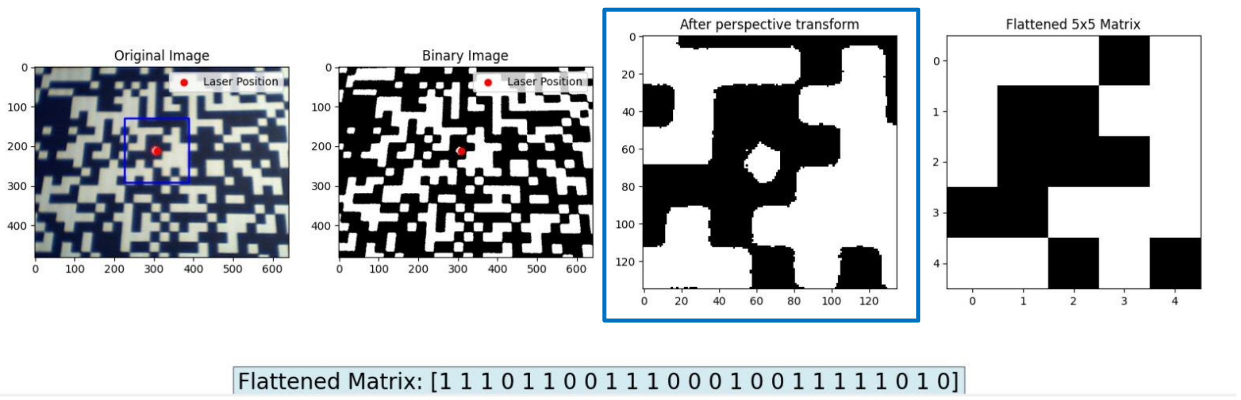
- It shows that the laser is stable enough for grid processing then the laser points at the cell which is marked as black
- Box Size and Grid Shape are parameters that were used for mapping and detecting the laser pointer is being printed. Then Transformation Matrix M, is applied there for perspective correction in order to align the warped image with the binary grid.
- Binary Grid is a binary matrix showing the detected grid states together with the laser's position within the grid. The binary representation of the grid which we got at the end is Binary Matrix Output

Blurred Grid Scenario:

- Lighting condition, camera focus, or motion makes the webcam feed to appear blurred. However, Open CV's **morphological operations** and **HSV-based laser detection algorithm** ensures accurate recognition of grid states and the laser pointer. The laser's position is correctly mapped to the binary grid and is finally displayed in the terminal.

The accuracy and robustness of the laser is affected by several factors:

Detection accuracy is significantly affected by the distance between the webcam and the image. When this distance exceeds the pre-calculated 32.1 cm, the perspective transformation results in more rows and columns than expected. As shown in **Figure 5.7**, the "After Perspective Transform" which is the 3rd image expands to 6×6 instead of the intended 5×5, causing errors in matrix flattening. This discrepancy makes it difficult to extract a properly structured 5×5 binary matrix.



```
Flattened Matrix: [1 1 1 0 1 1 0 0 1 1 1 0 0 0 1 0 0 1 1 1 1 1 0 1 0]
```

Figure 5.7: *Effect of incorrect camera distance on Matrix Extraction.*

Complexity of morphological operations, captured frame resolution, and the grid size affects the speed of the algorithm. In this project, real-time performance was achieved with an average processing rate of **50 frames per second (FPS),** which is fast enough to ensure minimal latency and allowing the system to respond immediately to laser movements.

The determinant of processing depends on hardware capabilities (e.g., CPU and GPU speed), the resolution of the captured frame and kernel size used to process the image. Our system finishes its operation in less than **20 milliseconds per frame**, indicating high efficiency and suitability for real-time applications.

5.3 Environmental Variability: Lighting and Noise Effects

We have done several testing under diverse environmental conditions such as variable light levels and with background noise. It guarantees the system's ability to keep the track of accuracy and efficiency despite external challenges.

- Lighting Variability:
  - **Bright Lighting:** Overexposure enlarges the laser dot leading to false positives. It was reduced by adjusting camera exposure and applying Gaussian blurring. Our system addresses reflections and glare by using a combination of morphological operations mainly opening and closing to filter out irrelevant artefacts.

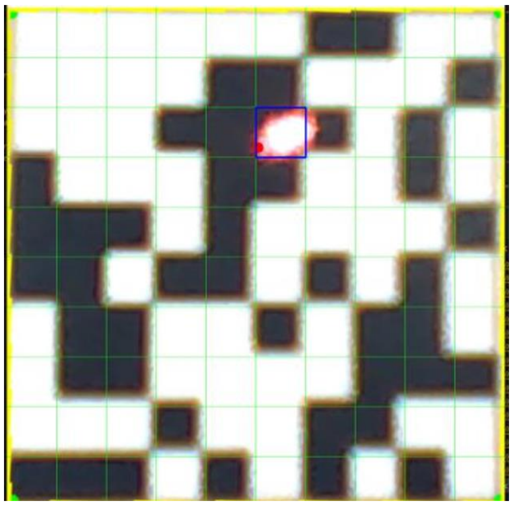


Figure 5.8: *Laser Detection in Overexposed Lighting in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions*

  - **Dim Lighting**: Even in a dark background, the laser pointer was easily distinguished. Due to this, the detection accuracy has dropped by 15% in dim lighting. To avoid and reduce these gaps we have used an external additional light source and tuned HSV thresholds.

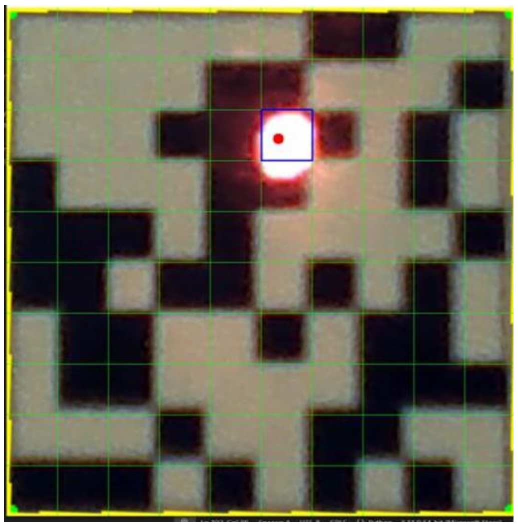
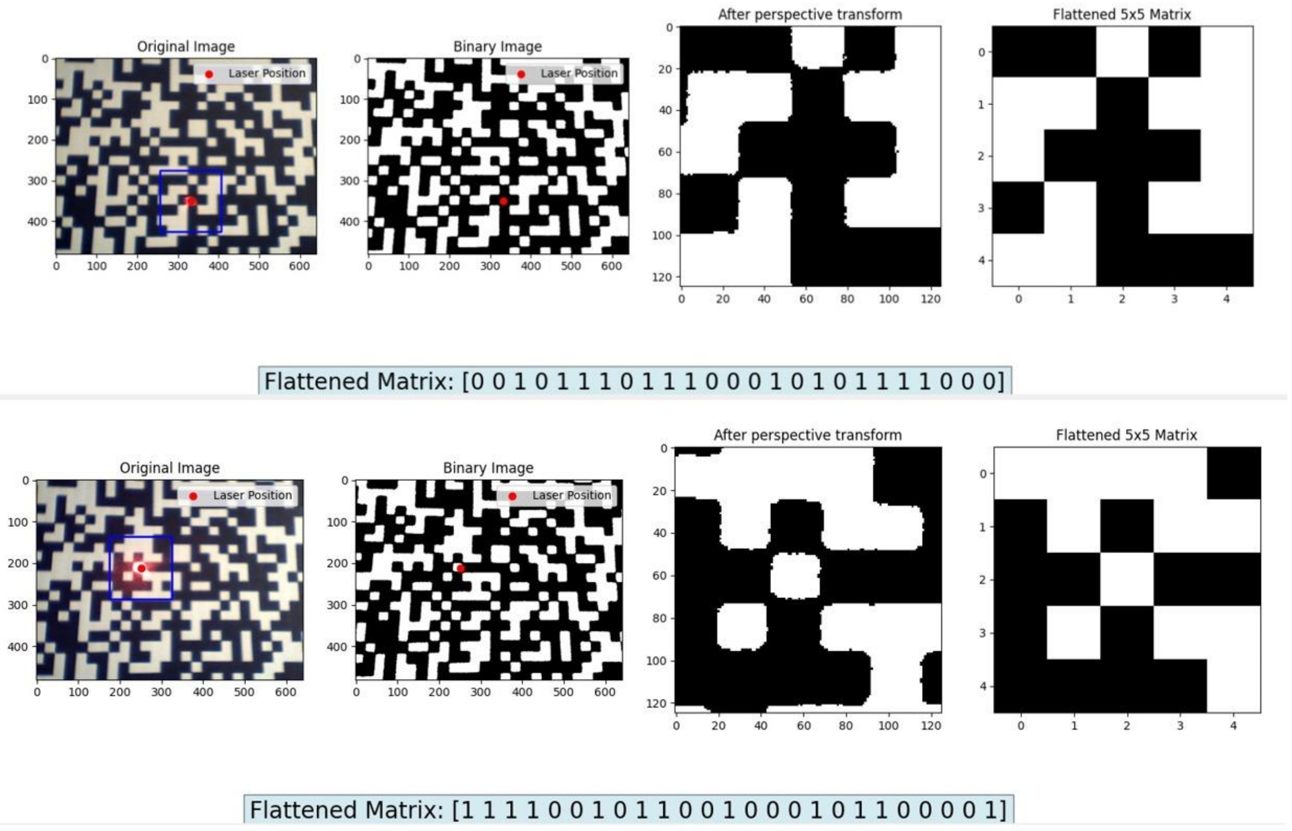


Figure 5.9: *Laser Detection in Low-Light Conditions in 10X10 matrix each cell with 2.7 mm by 2.7 mm dimensions.*

- **Background Noise**:
    - Morphological operations have effectively filtered out irrelevant false detections caused by reflective surfaces or red objects.
    - The system could effectively isolate the laser pointer even in presence of random objects and dynamic elements by applying robust morphological filters and refining HSV thresholds.

The experiments demonstrated the system's adaptability:

- Accuracy has been dropped a bit in highly reflective or noisy conditions but remained above 90%.
- The processing speed stayed the same in the presence of additional noise showcasing the efficiency of the image processing pipeline.



```
Flattened Matrix: [0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 0 1 0 1 1 1 1 0 0 0]
```

```
Flattened Matrix: [1 1 1 1 0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 0 0 1]
```

***Figure 5.10**: Final Results obtained* when *implementing on the bigger matrix* and extracting *5x5 binary matrix*

6 CHALLENGES AND SOLUTIONS

This chapter discusses the challenges encountered during the implementation and testing phases of the system and the solutions applied to overcome them by addressing technical, environmental, and usability issues.

6.1 VARIABILITY OF LIGHTING

One of the most significant challenges was Lighting conditions. Reflections, shadows and variations in brightness interfere the detection of the laser dot. Ambient light or light reflections overwhelm the laser's intensity by leading to false positives.

The laser detection system faced significant issues under variable lighting conditions:

- Glare from the laser in overexposed environments can cause false positives.
- The laser dot became faint in dim environments and reduces detection accuracy.

Solution:

Instead of working with RGB channels we have used the HSV (Hue Saturation Value) color space to mitigate lighting variability. HSV has a readymade ability to separates the color information (hue) from intensity (value), allowing the system to isolate the laser dot effectively despite varying brightness levels.

- **Dynamic Thresholding**: ambient lighting changes were dynamically adjusted using adaptive thresholding techniques.
- **Adjustment of the light**: Ring Light were externally added to provide consistent illumination.
- **Setting the Camera**: to enhance contrast and minimize the glare, the camera's exposure and brightness were fine-tuned.

6.2 MATRIX TILTING AND ALIGNMENT

Proper alignment of the matrix is crucial in implementation stage and if the matrix was skewed or tilted relative to the camera's field of view, the perspective transformation could fail to produce a reliable top-down projection.

Solution:

From Open CV tools, we used corner detection and perspective correction to identify the four corners of the matrix, even if the matrix was tilted, by the use of contour approximation and ordered them consistently as top-left, top-right, bottom-right and bottom-left. The `cv2.getPerspectiveTransform` function was then applied to generate a perfect top-down view. The camera's position relative to the matrix during initialization can also improve the performance better than the previous.

- **Perspective Transformation**: The matrix's perspective is corrected by applying a homography transformation.
- **Corner Detection**: Even if the matrix is tilted, advanced and better corner detection techniques guarantees that the matrix was accurately identified.

6.3 Laser Stability Management And Boundary Cases

Hand movements caused the laser pointer to be in an unstable situation. Quick movements of the laser caused loss of track of its position.

Boundary cases like the laser near the edges of a grid cell or in between two cells can lead to ambiguity in the grid's binary state interpretation.

Solution:

A combination of contour tracking and morphological operations eliminated the transient noise. The laser's centroid was averaged over several frames to account for minor flickers.

The system used a Region-Of-Interest around the laser dot to evaluate its proximity to cell boundaries when there is an ambiguity in boundaries. The laser's position was then assigned to the cell that covered the majority of laser dot area.

- **Stabilization Period**: A time-based mechanism was introduced, requiring the laser to remain in a single cell for 0.5 seconds before confirming its position.
- **Enhancing the Logic of Boundary**: The grid mapping algorithm was refined to analyze the laser dot's centroid and proximity to grid lines to enhance boundary detection accuracy. The system selects the cell where the majority of the laser dot area falls.

6.4 Reducing Noise And False Positives

Isolation of the laser dot can be greatly affected by unwanted light reflections and sensor interference. Noise and false positives may occur due to background elements such as reflective surfaces or red objects within the camera’s field of view.

Solution:

Gaussian blurring and morphological operations especially opening and closing are the main noise reduction techniques often used. Where the Gaussian blurring has decreased high-frequency noise in the image, opening gets rid of small white artifacts, while closing filled in small black gaps within the binary grid. Iterations of these operations and tuning the size of kernel has helped us to have a clean binary grid and robust laser dot detection.
Contour filtering helps in removing objects below a specific size threshold which guarantees that only significant contours were considered during processing.

- **Filtering the Color**: The Hue Saturation Value range for red colour is narrowed and morphological operations were applied to filter out irrelevant regions.
- **Gaussian Blurring**: Blurring smoothens the image and reduce the impact of small, bright spots unrelated to the laser pointer.

6.5 Real-Time Processing

Having a good real-time performance is helpful for the system’s usability mainly when given the computational requirements of image processing tasks such as laser tracking, contour detection and perspective correction.

Real-time processing is essential to detect the laser dot, process the grid, and produce the results we wanted to achieve. To achieve real-time performance, we optimized computational steps, focusing on operations such as perspective correction, morphological transformations, and contour detection.

Solution:

Morphological transformations, contour analysis and thresholding were implemented with minimal iterations to reduce processing time without compromising accuracy. This OpenCV functions were used to solve these issues and we have limited the resolution of the input video feed to a level that balanced computational efficiency with sufficient detail for grid and laser detection. Multi-threading was also considered to parallelize tasks such as image preprocessing and laser tracking.

- **Code Optimized**: Intensive operations like contour approximation and matrix division were streamlined using efficient OpenCV and NumPy functions.
- **Considerations of the Hardware's Used**: Testing on different systems ensured the software could achieve a minimum frame rate of 30 FPS on standard hardware.

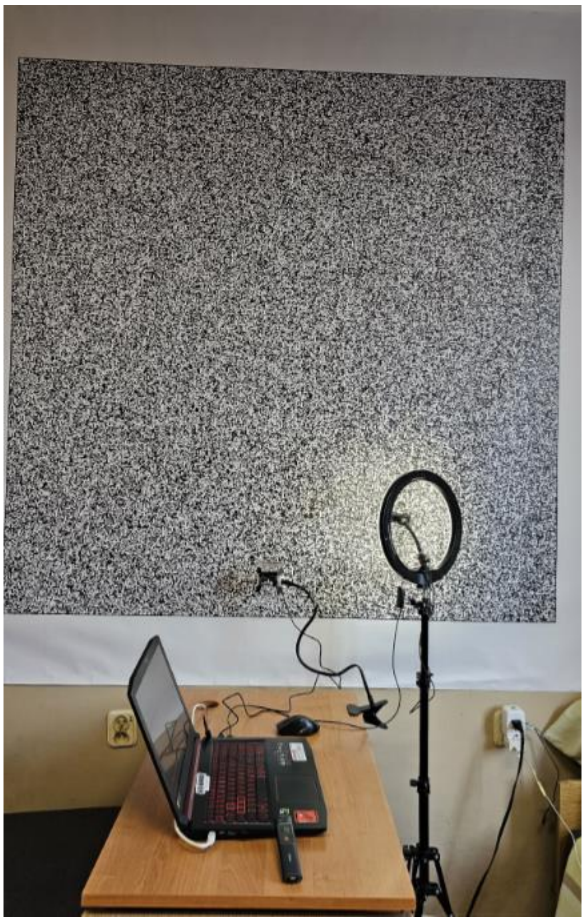


Figure 6.1: *Photo of the lab stand where the implementations and testing were performed*

7 CONCLUSIONS

This chapter concludes the project by highlighting its contributions, acknowledges limitations, and proposes directions for future research and development.

7.1 SUMMARY OF FINDINGS

The project we did has successfully implemented the detecting and tracking of a laser pointer on a binary matrix in real-time. It shows its potential for applications in interactive systems, educational tools, and robotics, where cost-effective and precise position tracking is required.

Key accomplishments we have achieved are listed below:

- The laser pointer's position is mapped to specific grid cells even under varying environmental conditions.
- Detection and perspective alignments are done accurately by the use of OpenCV-based image processing techniques.
- A stabilization mechanism is used to handle hand movements to have a precise positional tracking.
- A robust real-time performance is achieved with a frame rate of up to 45 FPS on standard hardware.

7.2 LIMITATIONS AND LESSONS LEARNED

These limitations provided valuable insights into areas where further research and optimization are necessary. Even if the project achieved its primary objectives there were also some limitations which we have observed:

- **Challenges in Boundary Detection**: Ambiguities arose when the laser pointer was near the boundaries of two or more grid cells that needed fine-tuned algorithms.
- **Dynamic Environments**: Matrix tilting or movement introduced errors which were not fully addressed within the project scope.
- **Hardware Constraints**: Camera’s resolution and processing power has been influenced the performance greatly, limiting the system’s portability on lower-end devices.
- **Dependency in Lighting**: In extreme lighting conditions, the performance has dropped slightly requiring external light sources or manual adjustments.

7.3 CONTRIBUTION TO THE FIELD

The approach can be applied in interactive displays and educational tools. This project makes the following contributions:

- **Open-Source Framework**: The implementation is built on widely used libraries like OpenCV and NumPy, making it adaptable and extensible for future applications.
- **Accessible Technology**: We have adopted cost-effective alternative to traditional tracking methods by using basic tools like a webcam and laser pointer.

7.4 FUTURE DIRECTIONS

To improve system’s capabilities and broaden its applications, the following future directions are proposed:

- **Adaptation to Lighting:** Advanced lighting compensation methods can be implemented, such as HDR imaging or machine learning-based adjustments.
- **Enhanced Noise Filtering**: Additional filtering techniques can be added to reduce false positives from background noise and reflections.
- **Detection of Dynamic Matrix**: Real-time tracking of a moving or rotating matrix can be enabled to make the system more adaptable in dynamic environments.
- **Handling Boundary:** A sophisticated boundary detection algorithms can also be developed using sub-pixel precision or deep learning techniques.
- **Integration with Other Systems**: Integration with robotic systems or augmented reality platforms for collaborative applications.

7.5 CLOSING REMARKS

Implementation and testing is done successfully and this demonstrate the feasibility of using simple, cost-effective tools for accurate position tracking. By doing further improvements and optimizations, it holds significant potential for innovation in fields requiring interactive and real-time tracking solutions.

REFERENCES

1. Khurana, M., & Ahuja, R. (2019). *Real-Time Object Detection and Tracking Using Laser Pointers.* Journal of Computer Science and Engineering
2. Szeliski, R. (2021). *Computer Vision: Algorithms and Applications* (2nd ed.). Springer.
3. Bradski, G., & Kaehler, A. (2008). *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media.
4. Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing* (4th ed.). Pearson Education.
5. Shi, J., & Tomasi, C. (1994). Good Features to Track. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
6. https://physics.stackexchange.com/questions/464866/determining-focal-length-of-converging-lens
7. https://opencv24-python-tutorials.readthedocs.io/_/downloads/en/stable/pdf/
8. https://docs.opencv.org/4.x/d9/df8/tutorial_root.html
9. https://www.indiamart.com/proddetail/bluetooth-wireless-barcode-scanner-23259273697.html
10. https://www.vecteezy.com/vector-art/511079-barcode-scanner-stock-vector-illustration
11. https://pl.shein.com/Cat-Toy-Laser-Pointer-Pen,-Teaching-Presentation-Pointer-Pen-With-Red/purple/green-Laser-And-Infrared-Light-p-25822285-cat-4491.html
12. https://pl.dreamstime.com/kod-kreskowy-i-zestaw-kod%C3%B3w-qr-etykieta-na-pasku-skanowania-dla-przemys%C5%82u-lub-supermarket%C3%B3w-znak-kodu-kreskowego-wektorowego-image171982959
13. https://www.researchgate.net/figure/A-view-of-99-D-yy-and-D-xy-boxfilters_fig2_371911542
14. https://physics.stackexchange.com/questions/464866/determining-focal-length-of-converging-lens
15. https://opencv24-python-tutorials.readthedocs.io/_/downloads/en/stable/pdf/
16. https://www.camerafv5.com/devices/manufacturers/huawei/clt-l09_hwclt_3/?utm_source=chatgpt.com
17. https://www.anandtech.com/show/12676/the-huawei-p20-p20-pro-review/7?utm_source=chatgpt.com